Summary of JANUS: Fast and Flexible Deep Learningvia Symbolic Graph Execution of Imperative Programs

Runyu Zheng (runyuz), Shangquan Sun (sunsean)

Problem and Motivation

Deep Learning (DL) frameworks have been created by scientists to improve the performance and also facilitate the use of Deep Neural Networks (NN). There are two families of existing frameworks. One family of frameworks is based on Symbolic graph execution, such as Tensorflow, Caffe2, and MXNet. The other one is based on imperative program execution, e.g. PyTorch, TensorFlow Eager, and MXNet Imperative. The previous one is easy to optimize through parallel execution and compiler optimization methods such as subexpression elimination and constant folding. But it requires users to construct a graph when defining an NN, which complicates the user experience. On the other hand, the latter one is more user-friendly but less possible to be optimized.

Hypothesis

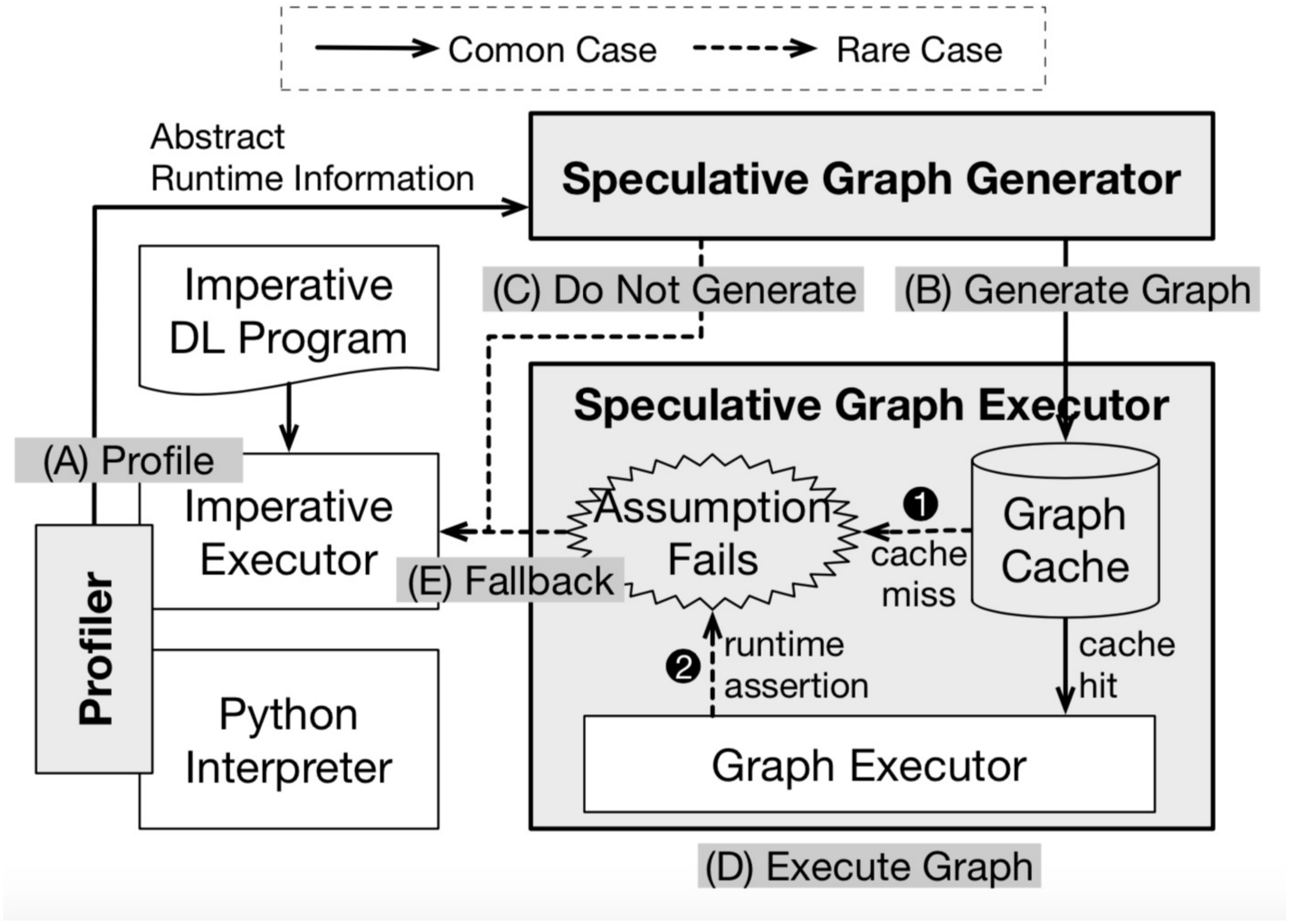
Therefore, if a system can convert an imperative program into a symbolic dataflow graph transparently, users can easily write a NN program and the system can generate and execute the corresponding optimized dataflow graph, which will lead to a best trade-off between a high performance and user experience. But the existing related works cannot do that successfully. They either fails to correctly generate graphs or cannot support Python. The difficulty lies in how to capture the three dynamic semantics of Python, e.g. dynamic control flow, dynamic types and impure functions, and convert them to restrictive set of operations.

To deal with the problem, the authors of the paper proposed JANUS, a DL framework that speculatively converts imperative Python DL program into symbolic graphs accordingly.

Solution Overview

In common cases, a DL program written in Python is sent to JANUS. The program is first executed by the imperative executor. At the same time, the Profiler collects abstract runtime information that is necessary for making reasonable assumptions. "After a sufficient amount of information has been gathered, the Speculative

Graph Generator then tries to convert the program into a symbolic dataflow graph with the assumptions based on the runtime information". Finally, the Speculative Graph Executor executes the generated symbolic graph.

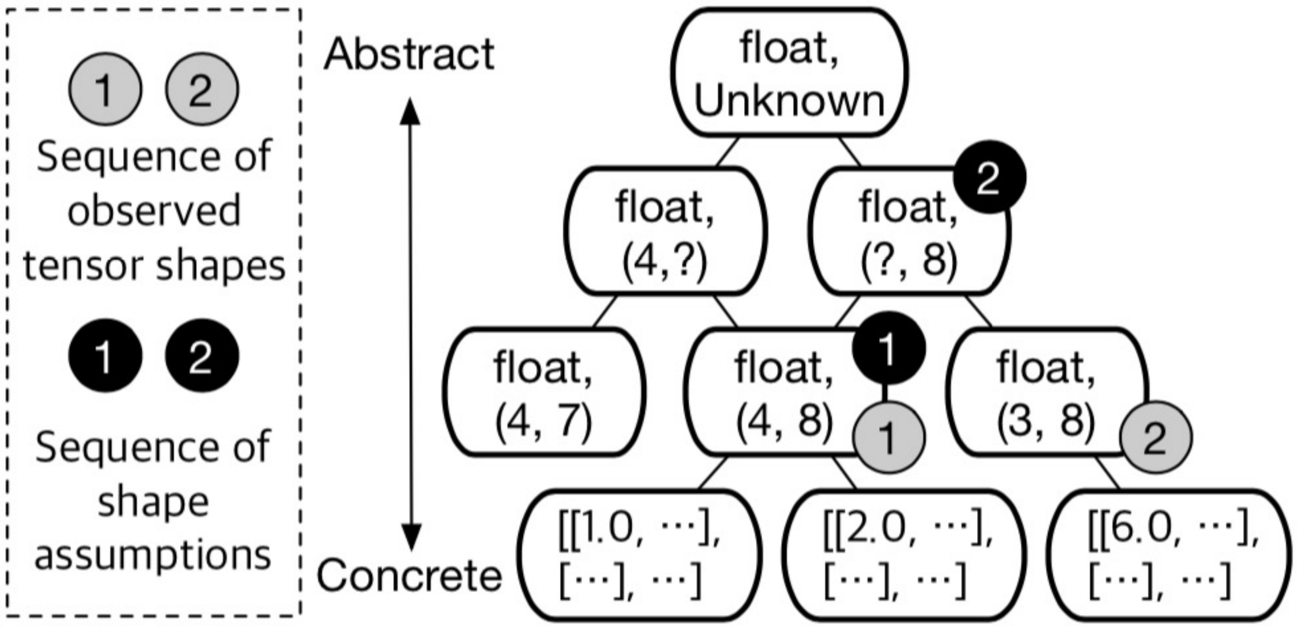


But in some rare cases, an assumption might fail. Then speculative graph executor will raise an assertion, which makes JANUS adopt the fallback and eventually goes back to the imperative executor. Also for some imperative programs, such as coroutine, the Speculative Graph Generator cannot generate a graph, since "they do not have any clear graph representations". Thus they can only be executed by imperative executor.

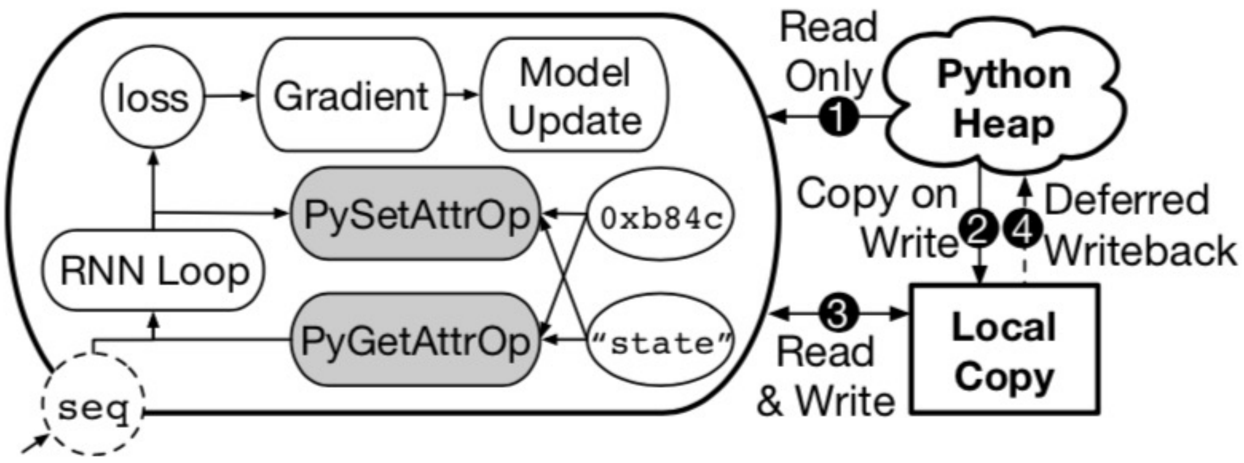
JANUS deals with the challenge of capturing three kinds of dynamic semantics.

- For conditional statements, if, and the iterative statements of Python, while and for, can be handled by using $\mathsf{SwitchOp}$, $\mathsf{MergeOp}$, and $\mathsf{NextIterationOp}$. For function calls, a function invocation operation $\mathsf{InvokeOp}$ is used to point to the generated graph of the callee function.
- The types of a Python expression cannot be determined explicitly, but some of them can be infered from types of other expression. For those expressions whose types cannot be inferred from other expressions,

for example, the types of function inputs or object attribute, Profiler observes the types of them during imperative executions. Some preliminary assumptions based on a few runs of profiling, might be relaxed and generalized after running several more iterations.



- JANUS allows the execution to read from and write to global states. It "handles global state accesses and updates alongside symbolic graph execution. To make things simpler and also faster, JANUS does not mutate globally accessible states in place on the fly. Instead, JANUS creates some local copies of global states, and mutates only the local copies during symbolic graph execution," and updates back to global states after execution, as shown in the following figure



Limitations and Possible Improvements

But JANUS still has some limitations:

- JANUS fails to convert many Python feature into a symbolic graph operation, for example, "programs that include invisible state mutations" and some complicated expression including coroutines and generators. Also "to keep the implementation of local copies of global state simple, Python object with custom accessor functions, e.g., __getattr__ or __setattr__, are not supported by the JANUS graph executor."
- JANUS can convert functions that are provided by framework or written in other languages, by looking up a separate white-list. But the white-list is limited.

Their possible improvements are: * Python object with custom accessor functions could be converted into graph with a more comlicated schema of accessing local and global states. But for those do not have a clear graph representations, it might be hard to generate graph for them. * The authors could further expand the white-list and cover more functions in the standard Python library.

Summary of Class Discussion

Q: Why does JANUS result outperform Tensorflow on LM models (1B) when number of GPUs is around 6, as shown below?

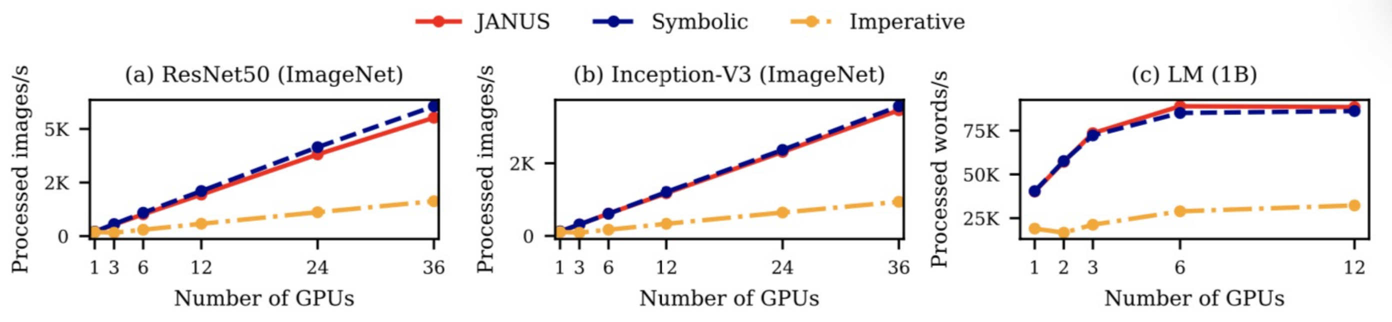


Figure 8: Training throughput for the ResNet50, Inception-V3, and LM models on JANUS, TensorFlow, TensorFlow Eager, using varying numbers of GPUs. The scale factor of JANUS is 0.77, 0.81, 0.18 for each applications, which is much bigger than the scale factor of TensorFlow Eager (0.24, 0.24, 0.14 each), and similar with the scale factor of TensorFlow (0.81, 0.80, 0.18).

There is no direct answer about this question in Section 6.3.2 where the authors are discussing about the figure. But the authors mention that "JANUS achieves bigger performance gain on recurrent and recursive neural networks than convolutional neural networks since recurrent and recursive neural networks have many concurrently executable operations". This might explain why only in the third subgraph for LM model, JANUS outperforms Tensorflow, but not for the previous two CNNs.

Q: What part of the paper is useful for other languages?

I guess the techniques used for capturing Impure Functions and Dynamic Control Flow are still useful for other languages, since almost all languages have impure functions and dynamic control flow. But not all languages need to handle dynamic type.

Q: How large is a graph in real world? It should be really large. The simple examples are only for demonstration.

Q: Why not generate multiple graphs for the same program with different assumptions (if one assumption fails) and pick a correct one?

The authors did not discuss why they did not do this. But my understanding is as following. Speculative Graph Generator is only responsible for generating graph. If multiple graphs sent to graph executor, it cost more computation and storage. Also the trial over assumptions might reduce overall performance. Besides, the authors emphasize "correctness" when discussing assumption failures. So I guess there might be some cases that an assumption makes graph run but actually turns out to be a wrong graph representation and thus wrong results are obtained.

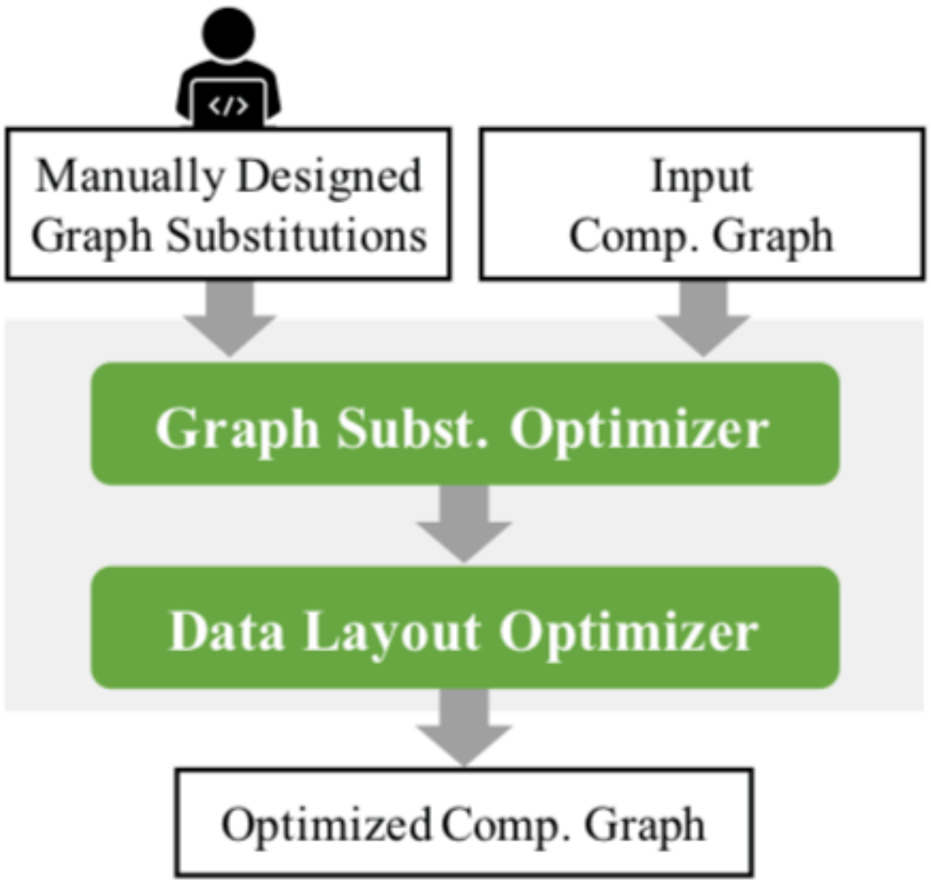
Summary of TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions

Runyu Zheng (runyuz), Shangquan Sun (sunsean)

Problem and Motivation

DNN frameworks represent a neural architecture as a computation graph. Existing DNN frameworks optimize the computation graph by applying graph substitution. Existing DNN frameworks optimize a computation graph by applying graph substitutions that are manually designed by domain experts. There are 3 problems for manually designed substitution:

- Maintainability: Hand-written codes require significant engineering effort. It is said there are 53K lines in Tensorflow for 155 substitutions. However, as deep learning keeps developing, new operator and new substitution strategies could be invented. So more engineering efforts are needed.
- Data layout: Current frameworks avoid this complexity of combining data layout and graph substitution by separating optimization problems and solving them sequentially.
- Correctness: Hand-written graph substitutions are error- prone, and a bug in graph substitutions can lead to incorrect computation graphs.

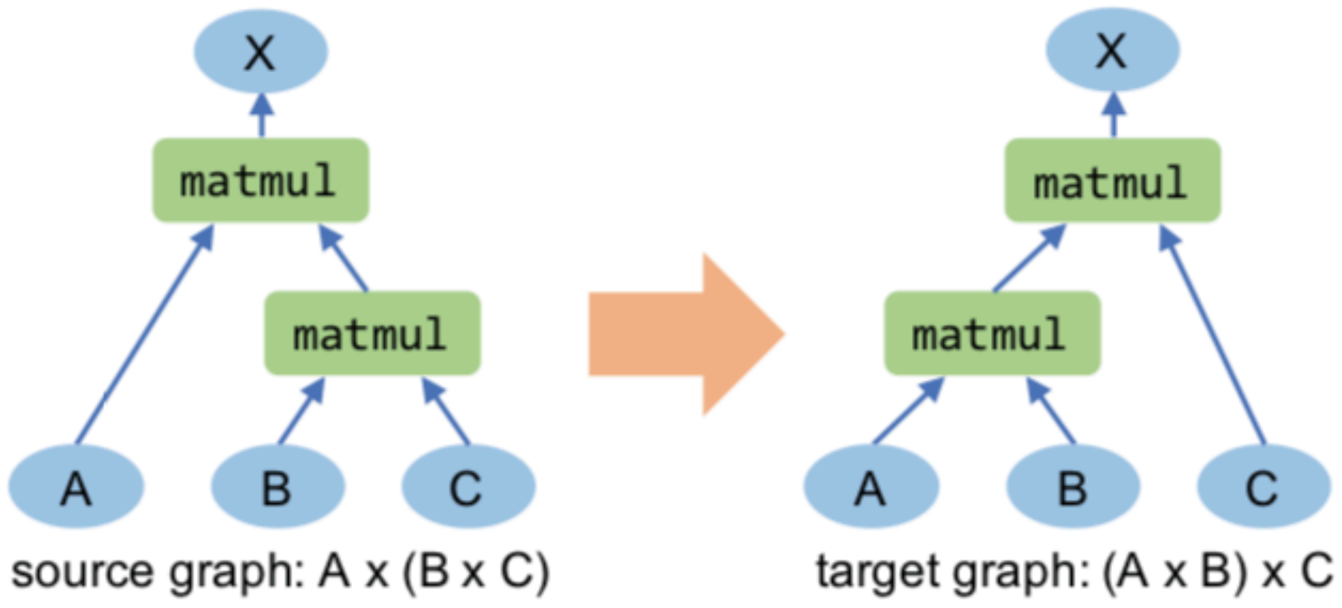


(a) Existing DNN frameworks.

Hypothesis

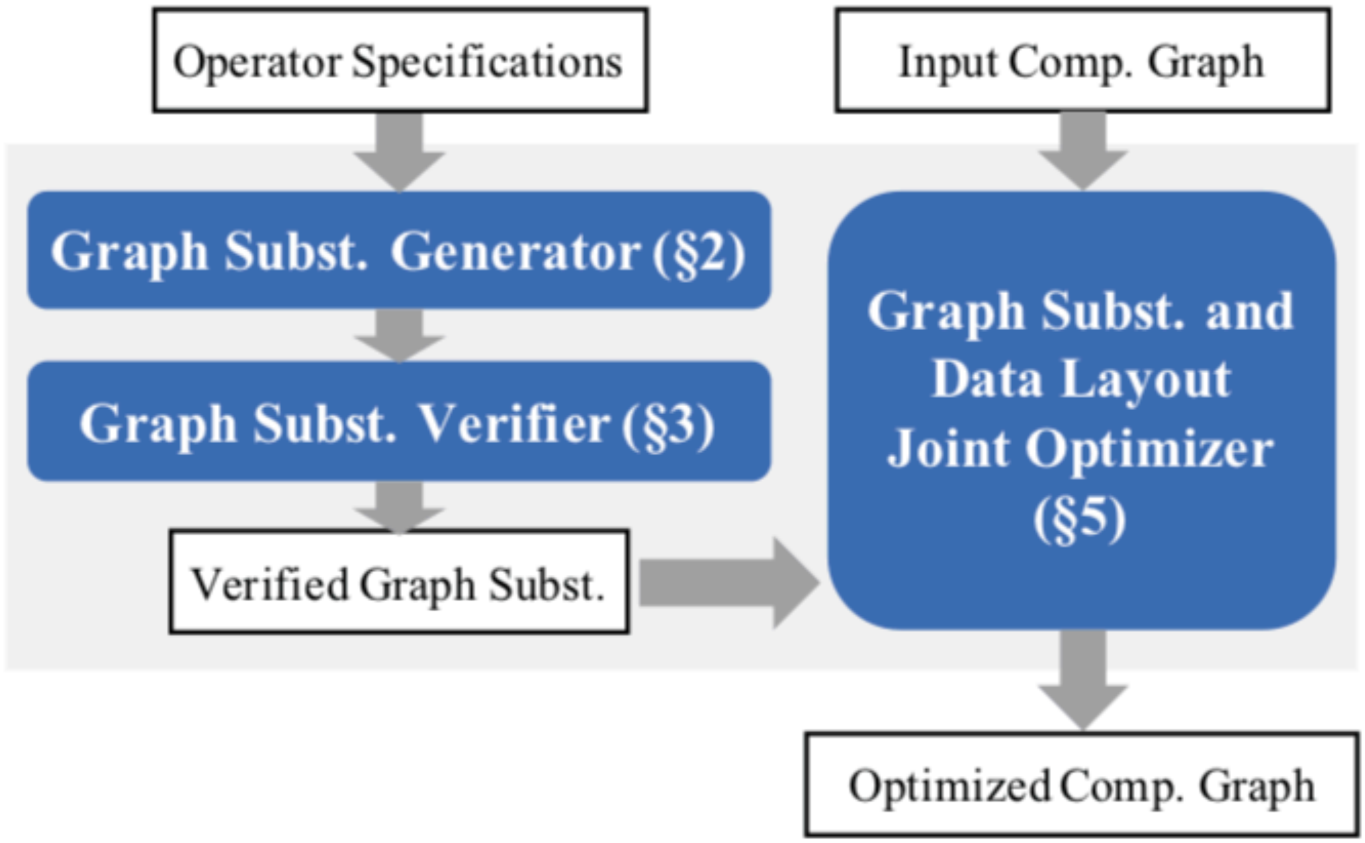
Graph Substitution

A graph substitution consists of three components: * source graph: matched to subgraphs in a compiutation graph * target graph: functionally equivalent new subgraph to replace the matched subgraph * a mapping relation between input/output tensors in the source and target graphs.



(a) Associativity of matrix multiplication.

Solution 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

(b) TASO.

TASO has three features: * automatic graph substitutions generation * automatic graph substitutions verification * joint optimization of graph substitution and data layout.

Graph Substitution generator

Step 1. Enumerating potential graphs and collecting their fingerprints. TASO first enumerates all potential graphs up to a certain size by using a given set of operators. To construct a graph, TASO iteratively adds an operator in the current graph by enumerating the type of the operator and the input tensors to the operator. For each graph, its fingerprint is collected, which is a hash of the output tensors obtained by evaluating the graph on some input tensors. Graph with same fingerprint is considered as substitutions for each other.

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
    $\mathcal{D} = \mathcal{D} + (\text{FINGERPRINT}(\mathcal{G}), \mathcal{G})$
    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 $\mathcal{S}$
```

Step 2: Testing graphs with identical fingerprint. For graphs with the same fingerprint, TASO further examines their equivalence on a set of test cases.

Graph Substitution Verifer

The key idea is to use a small set of operator properties expressed in first-order logic to formally verify substitutions.

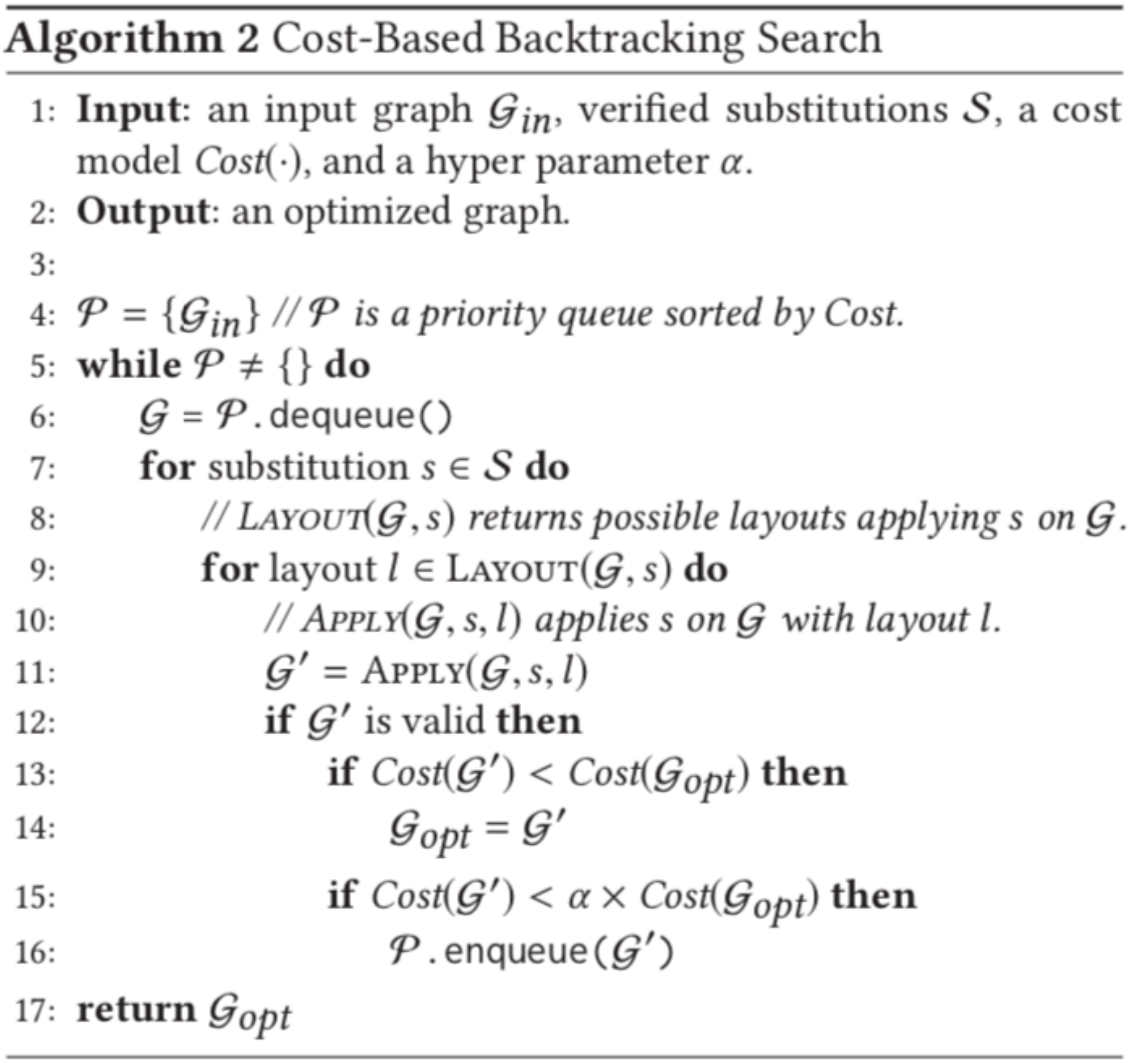
Pruning Redundant Substitutions

A graph substitution is redundant if it is subsumed by a more general valid substitution. There are two kinds of pruning considered in this paper:

- **Input tensor renaming.** TASO eliminates graph substitutions identical to other substitutions modulo input tensor renaming.
- **Common subgraph.** TASO also tries to eliminate substi- tutions whose source and target graphs have a common subgraph. Such case can be replaced with a more common graph.

Joint Optimizer

The author has proposed a cost-based backtracking search algorithm. The substitutions are applied to a matched subgraph with different data layout. The backtracking algorithm is used to find the new graph with smallest cost. The cost of the graph is an estimate for its execution time. TASO measures the execution time of a DNN operator once for each configuration and data lay- out, and estimates the performance of a graph by summing up the measured execution time of its operators.



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 $\mathcal{G}_{opt}$
```

Limitations and Possible Improvements

- **Reliance on user provided operator properties.** The verification step with SMT model is semi-automatic since it depends on user defined operator properties.
- **Scalability of the generator.** Current system can only scale up to graph substitution with 4 nodes.
- **Combination of graph-level and operator-level optimizations.** This joint optimization is challenging as both problems involve large and complex search spaces, and optimizations at one level affect the search space of the other.

Summary of Class Discussion

We have discussed some specific design choices of the system: * The graph generator will generate graph with up to 4 nodes in the paper. Distributed algorithm should be used to scale up the system. * If each part of the graph is run on different devices, the execution time should be evaluated on each device separately. * If no new operators are used in the framework, the previously generated graph substitutions could be reused.