Optimizing DNN Computation with Relaxed Graph Substitutions

Anonymous Authors[1]

Abstract

Existing deep learning frameworks optimize the computation graph of a DNN model by performing greedy rule-based graph transformations, which generally only consider transformations that improve runtime performance. As a result, complex graph optimizations that involve temporarily decreasing the runtime performance as an intermediate step are not considered in existing frameworks.

To address this limitation, we propose *relaxed graph substitutions* that enable the exploration of complex graph optimizations by relaxing the performance improvement constraint of existing systems. Relaxing this constraint greatly increases the space of computation graphs that can be found by repeatedly applying substitution rules to an input computation graph, ultimately including more efficient graphs in the search space. To effectively explore this large search space, we introduce a *backtracking search algorithm* over a set of relaxed graph substitutions to find optimized networks and use a *flow-based graph split algorithm* to recursively split a computation graph into smaller subgraphs to allow efficient search. We implement MetaFlow, to the best of our knowledge, the first relaxed graph substitution optimizer for DNNs and show that MetaFlow improves the inference and training performance by up to 1.6× and 1.2× respectively over existing deep learning frameworks.

1 Introduction

Deep neural networks (DNNs) have driven advances in many practical problems, such as image classification (Krizhevsky et al., 2012; He et al., 2016), machine translation (Wu et al., 2016; Bahdanau et al., 2014), and game playing (Silver et al., 2016). Over time, state-of-the-art DNNs have gotten substantially larger and deeper, resulting in greatly increased computational requirements.

To mitigate the increasing computational requirements it is standard to optimize the DNN computation, which is defined by a *computation graph* of mathematical operators (e.g., matrix multiplication, convolution, etc). Existing deep learning systems such as TensorFlow, PyTorch, and TVM optimize an input computation graph by performing *greedy rule-based substitutions* on the graph (Abadi et al. (2016); pyt (2017); Chen et al. (2018)). Each substitution replaces a subgraph satisfying a specific pattern with a new subgraph that computes the same result. For example, operator fusion combines a few operators into a single operator, which can eliminate intermediate results and increases the granularity of the operators, thereby reducing system overheads.

Existing deep learning optimizers only consider performance improving substitutions, which they *greedily* apply to an input computation graph until no further substitutions can be applied. More involved sequences of transformations where not all intermediate states are strict improvements are not considered. Unfortunately, this means that current optimizers miss many more complex optimization opportunities. Instead, we show that exploring a larger space of substitutions can improve the performance of widely used DNNs by up to 1.6× over existing rule-based optimizers.

In this paper, we propose *relaxed graph substitutions* with a cost-based backtracking search to address this limitation. We increase the space of computation graphs considered by *relaxing* the performance constraint of existing systems and allowing substitutions that may decrease runtime performance in addition to using the widely used performance improving substitutions. Although seemingly counter-intuitive, we observe that these "downgrading" graph substitutions are useful as intermediate steps in transforming graph architectures and discovering new graphs with significantly better runtime performance. To efficiently explore this larger space of computation graphs, we introduce a backtracking search over a set of relaxed graph substitutions to find improved networks after multiple transformation steps.

As a motivating example, we show how we can optimize a real DNN using relaxed graph substitutions. Figure 1 shows a sequence of relaxed graph substitutions on a ResNet module (He et al., 2016). The leftmost graph shows an optimized graph after greedy operator fusions, which combine a convo-

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Systems and Machine Learning (SysML) Conference. Do not distribute.

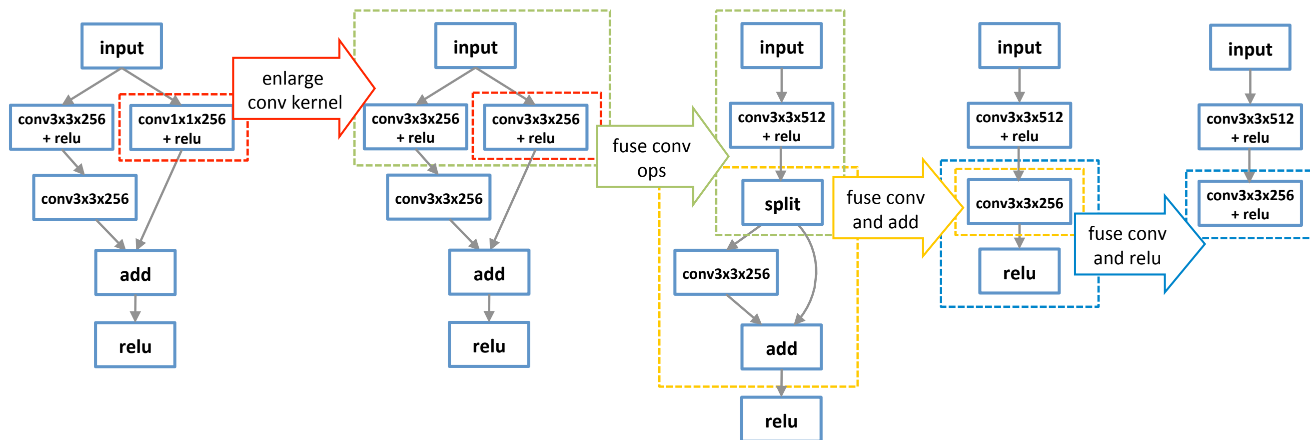


Figure 1. A sequence of relaxed graph substitutions on a ResNet module (He et al., 2016). Each arrow is a graph substitution, and the dotted subgraphs in the same color indicate the source and target graph of a substitution. "conv axbxc" indicates a convolution with kernel size $a \times b$ and c output channels. The final graph (right-most) is 1.5x faster than the original graph (left-most) on a NVIDIA P100 GPU.

lution and a following activation (i.e., relu) to a "convolution with activation". However, by adaptively applying relaxed graph substitutions (shown as the arrows in the figure), it is possible to generate a final graph (rightmost) that is 1.5x faster than the original graph (leftmost) on a NVIDIA P100 GPU. Note that the first graph substitution increases a convolution's kernel size from 1x1 to 3x3 by padding the kernel with extra 0's. This temporarily downgrades runtime performance (since a convolution with a larger kernel runs slower) but enables potential graph substitutions to further improve runtime performance. Section 3 describes other graph substitutions in Figure 1 in detail.

Adding relaxed graph substitutions to existing systems and applying them greedily could easily result in degraded performance. For example, the *enlarge operator* substitution in Figure 1 will likely degrade performance if the resulting convolution cannot be fused with another operator. While one could attempt to address this by adding special case rules and heuristics to an existing system, we believe such an approach would be error prone and brittle in the face of new architectures and new substitution rules. Instead we used a cost-based backtracking search to effectively explore the large space of computation graphs generated by applying relaxed graph substitutions.

First we introduce a cost model that incorporates multiple cost dimensions (e.g., FLOPs, execution time, memory usage, etc) and can accurately estimate the performance of different computation graphs. The cost model allows us to quickly compare different graphs.

Second, we propose a *backtracking search algorithm* that quickly finds efficient solutions for small graphs. However, the computation graphs of state-of-the-art DNNs are too large to efficiently search directly. Therefore, we use a *flow-based recursive graph split* algorithm that splits an original computation graph into individual subgraphs with reasonable sizes. The graph is split in a way that minimizes the number of graph substitutions spanning different subgraphs and is computed by solving a *max-flow problem* (Cormen et al., 2009). These subgraphs are optimized by the backtracking search and then stitched back together to form the final optimized graph.

We design, implement, and evaluate MetaFlow, to the best of our knowledge, the first relaxed graph substitution optimizer for DNNs. In addition to automatically generating high-performance computation graphs for generic DNN models, MetaFlow is also a framework-oblivious optimizer. We show that existing deep learning frameworks such as TensorFlow and TensorRT can directly use MetaFlow's optimized graphs, which improve the inference and training performance by up to $1.3\times$ and $1.2\times$, respectively.

We evaluate MetaFlow on five real-world DNNs, including Inception-v3 (Szegedy et al., 2016), SqueezeNet (Iandola et al., 2016), ResNet-34 (He et al., 2016) for image classification, RNN Text Classification (Kim, 2014), and Neural Machine Translation (Wu et al., 2016). MetaFlow is able to optimize each of these DNNs in under 5 minutes We show that MetaFlow outperforms existing deep learning frameworks with speedups ranging from $1.1\times$ to $1.6\times$. The performance improvement is achieved by discovering efficient computation graphs that decrease the overall memory usage by up to $1.7\times$ and the total number of kernel launches by up to $3.7\times$.

To summarize, our contributions are:

- We introduce relaxed graph substitutions, which enables the exploration of complex graph optimizations

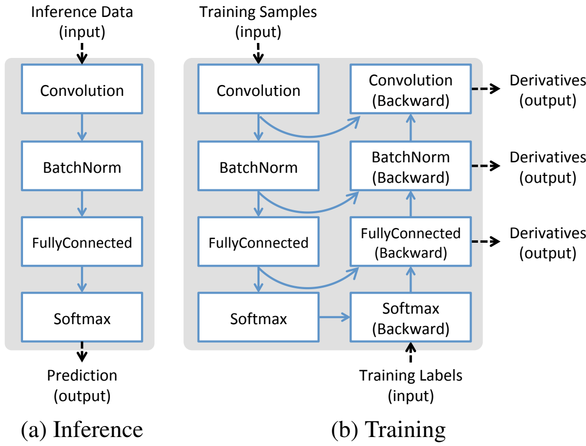


Figure 2. The inference and training graphs of an 4-layer example CNN model. Dotted edges are the inputs and outputs of each computation graph.

that are missing in existing deep learning frameworks.

- We propose a cost-based search algorithm that uses a max-flow algorithm to recursively split a large computation graph into smaller subgraphs, and uses a backtracking search algorithm to optimize individual subgraphs.
- We implement MetaFlow, the first relaxed graph substitution optimizer for DNNs. We show that MetaFlow can increase the inference and training performance by up to 1.6× and 1.2× over existing deep learning frameworks.

2 Overview

Similar to existing deep learning systems (Abadi et al., 2016; Chen et al., 2018; pyt, 2017), MetaFlow uses a *computation graph* $\mathcal{G}$ to define computation and state in a DNN model. Each node is a mathematical operator (e.g., matrix multiplication, convlution, etc), and each edge is a tensor (i.e., a n-dimensional array) between two operators. MetaFlow labels the inputs $\mathcal{I}$ and outputs $\mathcal{O}$ of a computation graph and defines computation in a DNN model as $\mathcal{O} = \mathcal{G}(\mathcal{I})$.

We define two computation graphs $\mathcal{G}$ and $\mathcal{G}'$ to be *equivalent* if $\mathcal{G}$ and $\mathcal{G}'$ compute mathematically equivalent outputs for any inputs (i.e., $\mathcal{G}(\mathcal{I}) = \mathcal{G}'(\mathcal{I})$). For a given computation graph $\mathcal{G}$, MetaFlow automatically finds an equivalent computation graph $\mathcal{G}'$ with optimized runtime performance.

For a DNN model, the inference and training procedures are defined by different computation graphs, as shown in Figure 2. An inference graph includes a single input and one or more outputs, while a training graph generally has two inputs (i.e., training samples and labels) and multiple

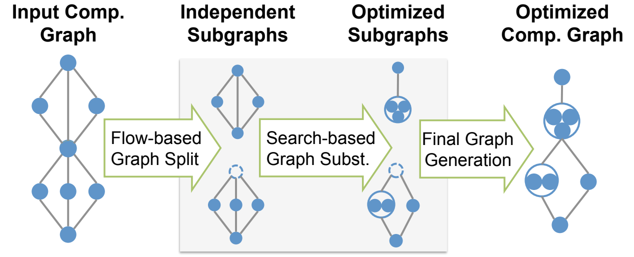


Figure 3. MetaFlow Overview.

outputs (i.e., derivatives for trainable parameters in each operator). MetaFlow merely treats inference and training as different input graphs to optimize and uses the same techniques to optimize both graphs.

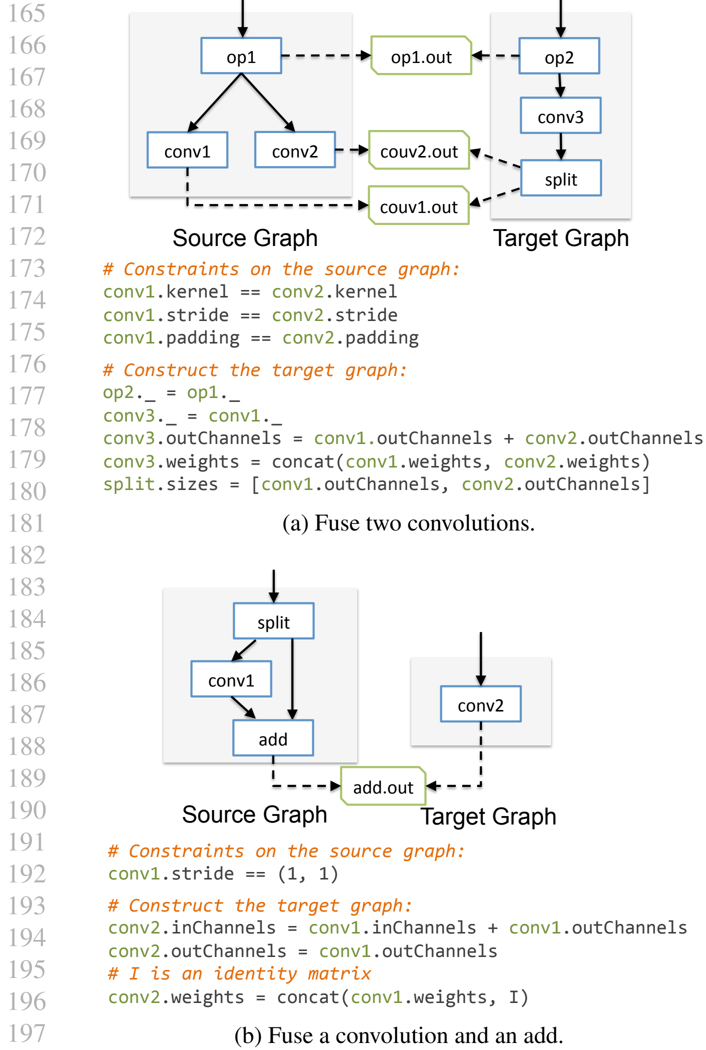
The main components of MetaFlow are shown in Figure 3. MetaFlow first divides an input computation graph into smaller individual subgraphs by using a *flow-based recursive graph split algorithm*. After that, each subgraph is optimized by performing a backtracking search on the search space defined by repeated application of relaxed graph substitutions to the subgraph. Finally, MetaFlow generates an optimized computation graph of the input graph by using the optimized subgraphs as building blocks. MetaFlow also serves as a framework-independent optimizer: an optimized computation graph can be executed on various deep learning runtimes, including the MetaFlow runtime, TensorFlow (Abadi et al., 2016), and TensorRT (trt, 2017).

3 Relaxed Graph Substitutions

This section introduces relaxed graph substitutions, each of which consists of a *source graph* that can map to particular subgraphs in a DNN and a *target graph* that defines how to create a new subgraph to replace a mapped subgraph.

Source graph. The source graph describes which subgraphs are valid candidates for this substitution. Each node in a source graph is associated with a type and can only be mapped to an operator of the same type. A source graph can also include *wildcard nodes* that can be mapped to any single operator. The wildcard nodes are useful when the type of an operator does not affect the substitution procedure and allow us to use a single source graph to describe many substitution scenarios. In addition to the type constraints, a graph substitution also defines additional constraints on one or multiple operators to further restrict mapping. Figure 4a shows a substitution to fuse two convolutions, which defines constraints on `conv1` and `conv2` to check they have the same kernel size, stride, and padding.

Edges in a source graph describe data dependencies between operators. A graph substitution requires the mapped subgraph to have the same data dependencies as the source



(a) Fuse two convolutions.

(b) Fuse a convolution and an add.

Figure 4. The source and target graphs of the relaxed graph substitutions used in Figure 1.

graph. Each operator can optionally have a *hyper edge* (shown as dotted edges in Figure 4) that can map to zero, one, or multiple edges connecting to external operators of the source graph. A hyper edge indicates that the operator's output can be accessed by external operators and must be preserved in the substitution.

Target graph. The target graph describes how to create a new graph to substitute for the mapped subgraph. For each newly created operator, a target graph defines how to set parameters and compute weights of the operator by using parameters and weights of the source graph. For each hyper edge in the source graph, there is a corresponding hyper edge in the target graph (shown as a pair of dotted edges). Any external operator originally connecting to a mapped operator in the source graph should now connect to the corresponding operator in the target graph.

Correctness. We define a graph substitution to be *valid* if its source and target graphs compute the same output for each hyper edge. This definition is similar to our definition of equivalent computation graphs if each hyper edge is considered as an output of the graph. It is easy to prove that performing valid graph substitutions preserves equivalence among generated computation graphs.

Composition. Our search procedure (described in Section 4.2) is able to perform complex optimizations on computation graphs by repeatedly applying a fairly small set of simple substitutions. For example, the following simplification of the computation in a recurrent unit (Equation 2 and 4 in (Lei et al., 2017)) can be achieved by repeatedly applying substitution rules that distribute multiplications, reorder commutative operators, and factor out common terms.

$$
\begin{aligned}
& \vec{x} \otimes \vec{y} + (\vec{1} - \vec{x}) \otimes \vec{z} && \text{(4 operators)} \\
\Rightarrow\; & \vec{x} \otimes \vec{y} + \vec{1} \otimes \vec{z} - \vec{x} \otimes \vec{z} && \text{(5 operators)} \\
\Rightarrow\; & \vec{x} \otimes \vec{y} - \vec{x} \otimes \vec{z} + \vec{z} && \text{(4 operators)} \\
\Rightarrow\; & \vec{x} \otimes (\vec{y} - \vec{z}) + \vec{z} && \text{(3 operators)}
\end{aligned}
$$

Some of these substitutions may temporarily reduce performance, but they will still be explored.

4 Search Algorithm

Relaxed graph substitutions can describe a comprehensive search space of potential computation graphs that are equivalent to an initial computation graph but have different runtime performance. Finding efficient graphs in the search space is challenging, since the search space can be infinite depending on which substitution rules are used. This makes it infeasible to exhaustively enumerate the search space for today's DNN models.

This section describes a number of techniques to prune the search space and efficiently find high-performance graphs. In particular, Section 4.1 introduces a cost model that incorporates multiple cost dimensions (e.g., FLOPs, execution time, memory usage, etc) and can accurately predict runtime performance of various computation graphs. Section 4.2 introduces a *backtracking search algorithm* that effectively finds an optimized candidate graph in the search space under the cost model. Because the computation graphs of state-of-the-art DNNs are too large to efficiently search directly, we use a *flow-based recursive graph split algorithm* (Section 4.3) that divides a computation graph into smaller individual subgraphs while maximizing the graph substitution opportunities.

4.1 Cost Model

We introduce a cost model that incorporates multiple dimensions to evaluate the runtime performance of a computation

220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274

Algorithm 1 A Backtracking Search Algorithm

1: **Input**: An initial computation graph $\mathcal{G}_0$, a cost model *Cost*$(\cdot)$, and a list of valid graph substitutions $\{S_1, ..., S_m\}$
2:
3: $\mathcal{Q} = \{\mathcal{G}_0\}$ *// a priority queue of graphs sorted by Cost*$(\cdot)$.
4: **while** $\mathcal{Q} \neq \{\}$ **do**
5: $\quad\mathcal{G} = \mathcal{Q}$.`dequeue()`
6: $\quad$**for** i = 1 to m **do**
7: $\quad\quad\mathcal{G}' = S_i(\mathcal{G})$
8: $\quad\quad$**if** *Cost*$(\mathcal{G}') <$ *Cost*$(\mathcal{G}_{opt})$ **then**
9: $\quad\quad\quad\mathcal{G}_{opt} = \mathcal{G}'$
10: $\quad\quad$**end if**
11: $\quad\quad$**if** *Cost*$(\mathcal{G}') < \alpha \times$ *Cost*$(\mathcal{G}_{opt})$ **then**
12: $\quad\quad\quad\mathcal{Q}$.`enqueue`$(\mathcal{G}')$
13: $\quad\quad$**end if**
14: $\quad$**end for**
15: **end while**
16: **return** $\mathcal{G}_{opt}$

graph. The cost model computes metrics for each operator in a graph and combines them appropriately to obtain a total cost. This includes both metrics that can be computed statically (e.g., FLOPs, memory usage, and number of kernel launches) as well as dynamic metrics that usually require measurements on specific hardware (e.g., execution time on a particular GPU or CPU). For dynamic metrics, previous work (Jia et al., 2018) shows that it is possible to accurately predict the execution time of a computation graph by only measuring a few representative operators on the hardware. For example, once we have measured and stored the execution time for a convolution with particular kernel size, stride, padding, input channels, and output channels, we can use that execution time for any other convolution with the same parameters.

Our cost model can optimize a single cost dimension (e.g., minimizing overall FLOPs) as well as incorporate multiple cost dimensions, such as minimizing execution time while maintaining a memory usage limit (by returning an infinite cost if the memory usage limit is exceeded). We observe that many graph substitutions result in a tradeoff among several cost dimensions instead of improving all of them. For example, the graph substitution in Figure 4b reduces memory accesses and kernel launches at the cost of increasing FLOPs.

4.2 Backtracking Search

We now describe a backtracking search algorithm to find an efficient graph using the cost model and a set of relaxed graph substitutions. Algorithm 1 shows the pseudocode. The search algorithm uses a parameter α (line 11 in the algorithm) to tradeoff between the search time and the best-discovered solution. By setting $\alpha = 1$, the search algorithm becomes a simple greedy algorithm and only considers graph substitutions that strictly reduce cost. As α increases, the search algorithm explores a larger part of the search space.

Algorithm 2 A Recursive Graph Split Algorithm.

1: **Input**: An initial computation graph $\mathcal{G}$
2: **function** GRAPHSPLIT($\mathcal{G}$)
3: $\quad$**if** $|\mathcal{G}| \leq$ threshold **then**
4: $\quad\quad$**return** $\mathcal{G}$
5: $\quad$**else**
6: $\quad\quad$*// MIN-CUT$(\cdot)$ returns a minimum vertex cut.*
7: $\quad\quad\mathcal{C}$ = MIN-CUT($\mathcal{G}$)
8: $\quad\quad\mathcal{G}_1 = \{o_i \in \mathcal{G} | o_i$ is reachable from $\mathcal{C}\}$
9: $\quad\quad\mathcal{G}_2 = \mathcal{G} - \mathcal{G}_1$
10: $\quad\quad$**return** GRAPHSPLIT($\mathcal{G}_1$) || GRAPHSPLIT($\mathcal{G}_2$)
11: $\quad$**end if**
12: **end function**

4.3 Flow-Based Recursive Graph Split

The backtracking approach described in Section 4.2 is too slow to run on the entire computation graph of state-of-the-art DNNs. Since graph substitutions are generally performed on a few locally connected operators, splitting the computation graph of a DNN model into smaller individual subgraphs can preserve most graph substitutions. Based on this observation, we propose a *flow-based recursive graph split algorithm* to divide a computation graph into smaller individual subgraphs that are amenable to backtracking search.

To split a graph into two disjoint subgraphs, we minimize the number of graph substitutions spanning the subgraphs, since these graph substitutions cannot be performed on either subgraph. For each operator $o_i \in \mathcal{G}$, we define its capacity $Cap(o_i)$ to be the number of graph substitutions that map to at least one in-edge and one out-edge of operator o_i. These graph substitutions are disabled if operator o_i is used to split the graph. By using $Cap(o_i)$ as the weight for each operator, we map the graph split problem to a *minimum vertex cut* problem (Cormen et al., 2009) and can use any *max-flow* algorithm to find a minimum cut.

The above flow-based algorithm splits an arbitrary graph into two subgraphs by minimizing spanning graph substitutions. Algorithm 2 shows a recursive algorithm that uses the max-flow algorithm as a subroutine to split an entire computation graph into individual subgraphs smaller than a threshold.

After running the backtracking search on each subgraph individually, we stitch the subgraphs back together to get an optimized computation graph. Finally, we run a small search around the points where the subgraphs are joined together for substitutions we may have missed at the boundary between subgraphs.

275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329

Table 1. DNNs used in our experiments.

DNN	Description
Convolutional Neural Networks (CNNs)	
Inception-v3	A 102-layer CNN with Inception modules
SqueezeNet	A 42-layer CNN with fire modules
ResNet34	A 34-layer CNN with residual modules
Recurrent Neural Networks (RNNs)	
RNNTC	A 3-layer RNN for text classification
NMT	A 4-layer RNN for neural machine translation

5 IMPLEMENTATION

MetaFlow is implemented as a framework-independent optimizer for arbitrary computation graphs. The MetaFlow cost model and runtime use existing deep learning libraries (e.g., cuDNN (Chetlur et al., 2014) and cuBLAS (cub, 2016)) to estimate the execution time of a computation graph and perform real executions. MetaFlow accepts a user-defined cost function that incorporates one or multiple cost dimensions and finds a computation graph optimizing the cost function. An optimized graph can be transformed to formats that are accepted by existing deep learning frameworks, including TensorFlow and TensorRT. This allows existing deep learning frameworks to directly use MetaFlow's optimized graphs as inputs to improve runtime performance. In particular, we show that MetaFlow can further improve the runtime performance of TensorFlow and TensorRT by up to 1.3×, though these systems internally perform rule-based graph transformations before executing an input computation graph.

6 EVALUATION

This section evaluates the performance of MetaFlow by answering the following questions:

- How does MetaFlow compare to existing deep learning frameworks that rely on rule-based graph transformations?
- Can MetaFlow's graph optimization be used to improve the runtime performance of these deep learning frameworks?
- Can MetaFlow improve both the inference and training performance of different real-world DNNs?

6.1 Experimental Setup

Table 1 summarizes the DNNs used in our experiments. We use three representative CNNs (i.e., Inception-v3 (Szegedy et al., 2016), SqueezeNet (Iandola et al., 2016) and ResNet34 (He et al., 2016)) for image classification. They use different DNN modules to improve model accuracy and exhibit different graph architectures. RNNTC and NMT are sequence-to-sequence RNN models from (Lei et al., 2017) for text classification and neural machine translation, respectively. RNNTC uses an embedding layer, a recurrent layer with a hidden size of 1024, and a softmax layer. NMT includes an encoder and a decoder, both of which consist of an embedding layer and two recurrent layers each with a hidden size of 1024. We follow previous work and use SRU (Lei et al., 2017) as the recurrent units for RNNTC and NMT. All experiments were performed on a GPU node with a 10-core Intel E5-2600 CPU and 4 NVIDIA Tesla P100 GPUs.

In all experiments, MetaFlow considers all key graph substitutions in TensorFlow XLA as well as the graph substitutions described in Section 3 and Figure 4. Unless otherwise stated, we use $\alpha = 1.05$ as the pruning parameter for our backtracking search algorithm. The graph split algorithm recursively splits subgraphs with more than 30 operators. This allows MetaFlow's search procedure to complete in less than 5 minutes for all the experiments.

6.2 Inference Performance

End-to-end performance. We first compare the end-to-end inference performance between MetaFlow and existing deep learning frameworks, including TensorFlow, TensorFlow XLA, and TensorRT, on a NVIIDA P100 GPU. MetaFlow can automatically transform optimized computation graphs to standard formats accepted by TensorFlow and TensorRT. Therefore, we also evaluate the performance of TensorFlow, TensorFlow XLA and TensorRT with MetaFlow's optimized graphs.

The results are shown in Figure 5. The blue lines show the best performance achieved among the three existing frameworks without using MetaFlow's optimized graphs, and the red lines show the MetaFlow performance. MetaFlow outperforms existing deep learning inference engines with speedups ranging from 1.1× to 1.6×. In addition, when applied the optimized graphs to those frameworks, MetaFlow also improves the inference performance of TensorFlow, TensorFlow XLA and TensorRT by up to 1.3×. Note that all existing systems internally perform rule-based graph transformations before executing a computation graph, therefore the performance improvement comes from other graph optimizations beyond rule-based graph transformations.

We further study the performance difference between MetaFlow and existing rule-based deep learning frameworks on multiple cost dimensions, including the overall memory accesses, the total amount of FLOPs, the number of kernel launches and the device utilization. For this experiment, we use TensorRT as the baseline as it has the best performance among existing deep learning systems. For TensorRT, the cost metrics are collected through its `IProfiler` interface.

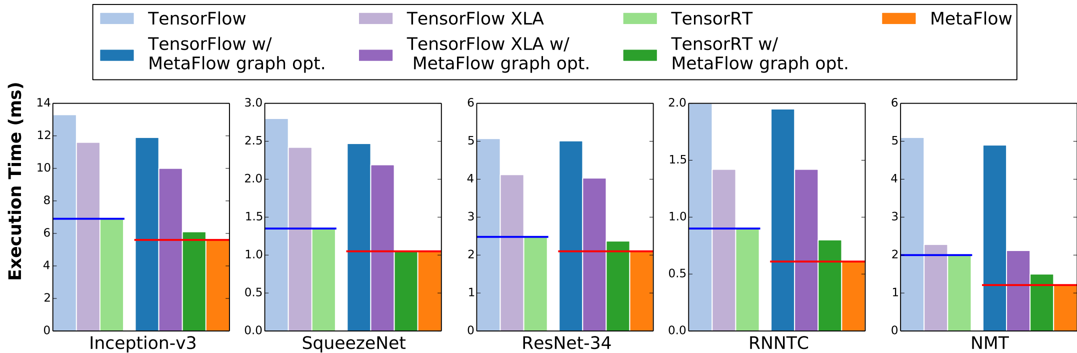


Figure 5. End-to-end inference performance comparison among MetaFlow, TensorFlow, TensorFlow XLA, TensorRT, and TVM. For TensorFlow, TensorFlow XLA and TensorRT, we also measure the performance with MetaFlow's optimized graphs. The experiments were performed on a NVIDIA P100 GPU. For each DNN model, the blue and red lines indicate the performance achieved by the best existing systems and MetaFlow, respectively.

Table 2. Performance comparison between MetaFlow and TensorRT on multiple cost dimensions. For TensorRT, the cost metrics are collected through its `Profiler` interface. The device utilization is computed by normalizing the FLOPs by the execution time (TFLOPs per second). For each cost dimension, a number in bold shows the one with better performance.

DNN	**Execution Time** (ms)		**Memory Accesses** (GB)		**Launched Kernels**		**FLOPs** (GFLOPs)		**Device Utilization**	
	TensorRT	MetaFlow	TensorRT	MetaFlow	TensorRT	MetaFlow	TensorRT	MetaFlow	TensorRT	MetaFlow
Inception-v3	6.9	**6.1**	103.4	**70.2**	138	**115**	**5.7**	6.1	0.82	**0.90**
SqueezeNet	1.35	**1.06**	69.7	**45.4**	54	**38**	**0.67**	1.00	0.49	**0.92**
ResNet34	2.48	**2.37**	32.3	**21.8**	46	**38**	**0.86**	1.24	0.35	**0.52**
RNNTC	0.9	**0.61**	2.65	**2.34**	220	**83**	0.22	**0.2**	0.16	**0.40**
NMT	1.9	**1.21**	10.07	**7.7**	440	**135**	0.84	**0.78**	0.44	**0.64**

Tables 2 compares the cost dimensions between TensorRT and MetaFlow. Compared to TensorRT, MetaFlow reduces the overall memory accesses by 1.6× and the number of kernel launches by up to 3.7×. For the CNNs in our experiments, MetaFlow achieves performance improvement at the cost of the increasing computation (i.e., FLOPs) in a computation graph. This provides opportunities to potentially fuse multiple operators to reduce memory accesses and kernel launches. Figure 6 shows how MetaFlow optimizes the computation graph of an Inception module. MetaFlow enlarges a `conv1x3` and a `conv3x1` operator both to `conv3x3` operators to fuse them to a single `conv3x3` operator. This reduces both memory accesses and kernel launches.

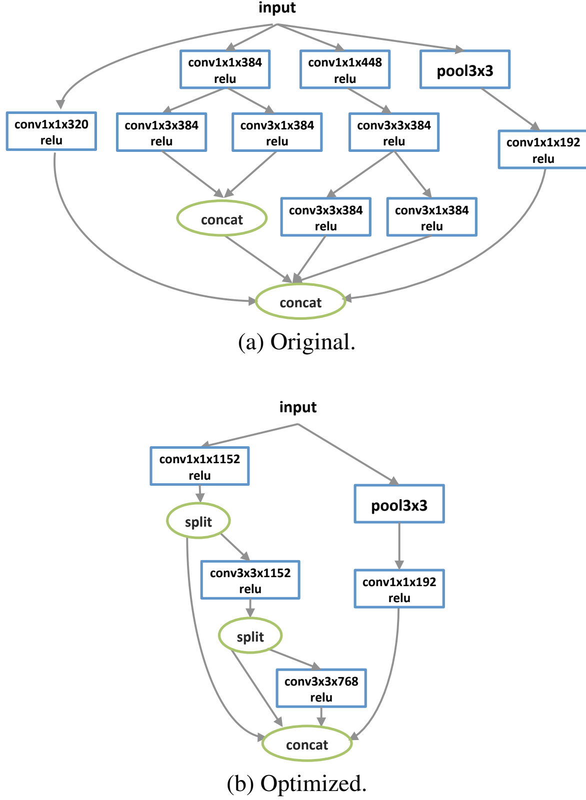
For RNNs, MetaFlow can also decrease the FLOPs compared TensorRT. Section 3 shows how MetaFlow transforms the computation in a recurrent unit from 4 element-wise operators to 3 by composing a few simple substitutions. This is a potential but currently missing optimization in TensorRT (v4.0.1, the latest version as of Sep 2018).

Subgraph performance. We evaluate whether MetaFlow can improve the performance of individual subgraphs in a DNN. Figure 7 compares the performance of TensorRT and MetaFlow on individual subgraphs in Inception-v3. The figure shows that MetaFlow can consistently find faster computation graphs than TensorRT, which leads to an end-to-end performance improvement of 25%.

Comparison with kernel code generation. In Figure 8, we compare MetaFlow-optimized graphs (which use cuDNN operator kernels) against graphs with operator kernels generated by TVM (Chen et al., 2018). TVM is able to generate high-performance kernels, especially for convolutions, making it competitive on some benchmarks despite its lack of the higher-level graph optimizations MetaFlow provides. The optimizations in TVM operate at a lower level than the optimizations in MetaFlow, so they could easily be composed. In the future, we plan to integrate TVM as a backend for MetaFlow so that we can achieve maximum performance via both graph optimization and individual kernel code generation.

6.3 Training Performance

The idea of relaxed graph substitutions of MetaFlow is designed for generic computation graphs including both inference and training. We also evaluate how MetaFlow improves the training performance of different DNNs. We use TensorFlow to run both the baseline computation graphs and the optimized graphs from MetaFlow. We follow the suggestions in TensorFlow Benchmarks (ten, 2017) and use synthetic data to benchmark the training performance. The



(a) Original.

(b) Optimized.

Figure 6. The original and the MetaFlow optimized computation graphs of an Inception module (Szegedy et al., 2016). The optimized computation graph increases the FLOPs by 3% while reduces the overall memory accesses by 1.6× and the number of kernel launches by 2×.

experiments were performed on four NVIDIA P100 GPUs of a single compute node, with data parallelism and a global batch size of 64.

Figure 9 shows the comparison of the training throughput. A training graph generally involves more outputs and data dependencies than an inference graph, as shown in Figure 2. As a result, the gains are relatively smaller. However, relaxed graph substitutions still discover computation graphs that are up to 1.2 × faster than the original graphs.

6.4 Search Algorithm

We first compare our backtracking search algorithm (described in Section 4.2) with a baseline exhaustive search algorithm that enumerates all potential computation graphs in the search space. To allow the exhaustive search to complete in reasonable time, we use small DNN models including AlexNet (Krizhevsky et al., 2012), VGG16 (Simonyan & Zisserman, 2014), ResNet18 and an Inception module shown in Figure 6a.

Table 3 compares the execution time of the two algorithms.

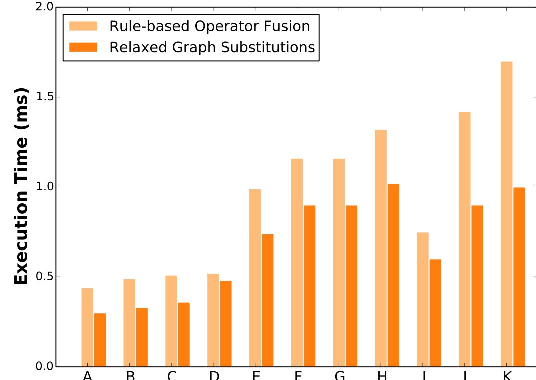


Figure 7. Performance comparison between MetaFlow and TensorRT on individual subgraphs in Inception-v3 (Szegedy et al., 2016). The experiments were performed on a NVIDIA P100 GPU.

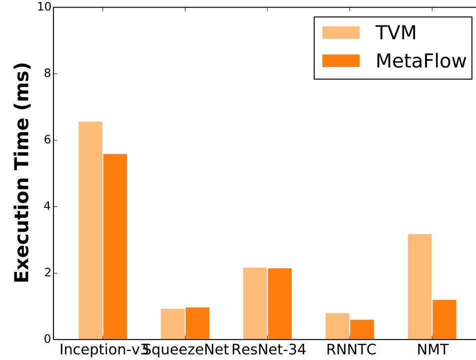


Figure 8. Performance comparison between MetaFlow and TVM on a NVIDIA P100 GPU.

Compared to the baseline exhaustive search, our backtracking search finds the same optimal results for the four DNNs and reduces the execution time by orders of magnitude over the baseline.

Second, we evaluate the performance of our backtracking search algorithm with different pruning parameters α. Figure 10 shows the performance of the best discovered graphs and the end-to-end search time for different α. The figure show that using a relatively small α allows us to find an optimized computation graph for the Inception model within a few seconds.

7 Related Work

Greedy rule-based graph transformation has been widely used by existing deep learning systems (e.g., TensorFlow XLA (xla, 2017), TensorRT (trt, 2017)) to improve the runtime performance of a computation graph. Existing

440
441
...

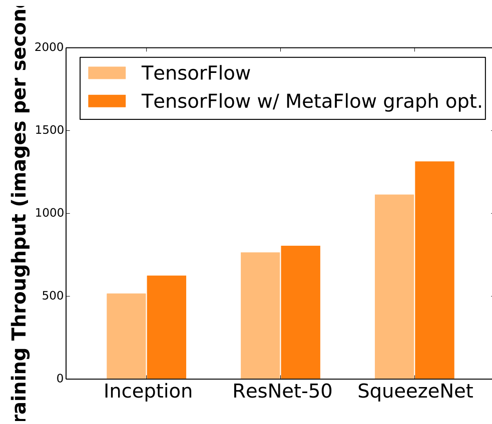


Figure 9. Training performance comparison between TensorFlow and TensorFlow w/ MetaFlow's optimized computation graphs. The experiments were performed on 4 NVIDIA P100 GPUs with data parallelism and a global batch size of 64.

Table 3. Performance comparison between MetaFlow's backtracking search (with $\alpha = 1.05$) and a baseline exhaustive search on AlexNet, VGG16, ResNet18 and an Inception module shown in Figure 6a. A check mark indicates the backtracking search find the same optimal result as the exhaustive search under the cost model.

Graph	Exhaustive Search	Backtracking Search	Same Result?
AlexNet	5.0 seconds	0.1 seconds	✓
VGG16	2.3 minutes	0.2 seconds	✓
InceptionE	12.8 minutes	0.29 seconds	✓
ResNet18	3.1 hours	0.99 seconds	✓

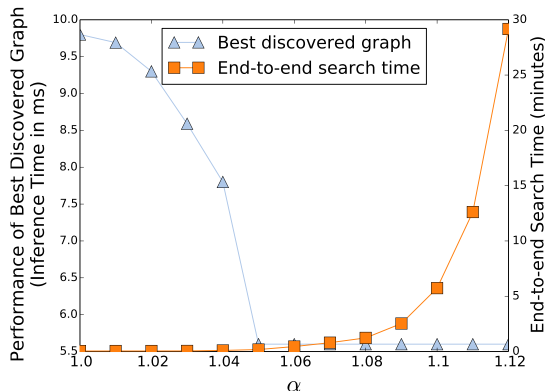


Figure 10. The performance of the best discovered graphs (shown as the red line) and the end-to-end search time for different α.

systems typically require each rule to improve the runtime performance, which prevents a large number of potential graph substitutions from being considered by rule-based operator fusion. The key different between existing DL frameworks and MetaFlow is that MetaFlow considers relaxed graph substitutions and uses a novel search algorithm to discover efficient computation graphs in the search space.

Automatic kernel generation. Recent work has proposed deep learning frameworks (e.g., Tensor Comprehension (Vasilache et al., 2018), TVM (Chen et al., 2018), Halide (Ragan-Kelley et al., 2013)) that automatically generate high performance kernels for dedicated hardware. These kernel generation techniques solve an orthogonal problem of how to improve performance of individual operators, while MetaFlow aims at optimizing computation graphs using relaxed graph substitutions. We believe it is possible to combine relaxed graph substitutions with automatic code generation and leave this as future work.

8 Conclusion

Existing deep learning optimizers use a greedy algorithm to optimize computation graphs by applying graph substitutions that are strictly performance increasing. This approach misses potential performance gains from more complex transformations where some intermediate states are not improvements. We identify the potential of performing such transformations, and propose relaxed graph substitutions to achieve them. We provide a system, MetaFlow, for optimizing DNN computation graphs using relaxed graph substitutions, and show that MetaFlow can achieve up to 1.6× performance improvements on DNNs. Finally, we demonstrate that relaxed graph substitutions are widely applicable as we show that adding them to existing optimizers such as TensorFlow XLA and TensorRT results in further performance improvements.

References

Dense Linear Algebra on GPUs. `https://developer.nvidia.com/cublas`, 2016.

Tensors and Dynamic neural networks in Python with strong GPU acceleration. `https://pytorch.org`, 2017.

TensorFlow Benchmarks. `https://www.tensorflow.org/performance/benchmarks`, 2017.

NVIDIA TensorRT: Programmable inference accelerator. `https://developer.nvidia.com/tensorrt`, 2017.

Tensorflow xla overview. `https://www.tensorflow.org/performance/xla`, 2017.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G.,

495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549

Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E. Q., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. URL http://arxiv.org/abs/1802.04799.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. URL http://arxiv.org/abs/1410.0759.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.

Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. Exploring hidden dimensions in parallelizing convolutional neural networks. *CoRR*, abs/1802.04924, 2018. URL http://arxiv.org/abs/1802.04924.

Kim, Y. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014. URL http://arxiv.org/abs/1408.5882.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS, 2012.

Lei, T., Zhang, Y., and Artzi, Y. Training rnns as fast as cnns. *CoRR*, abs/1709.02755, 2017. URL http://arxiv.org/abs/1709.02755.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.