Abstract

Most deep learning compilers use a greedy approach to optimizing computational graphs in that every successive graph substitution must result in a strict performance increase. This narrows the space of possible optimizations and motivates the notion of relaxed graph substitutions: substitutions that don't themselves necessarily result in a more optimal computation graph, but facilitate further transformations down the line that ultimately yield a net improvement in performance. In order to determine the sequence of transformations to be applied such that the net performance increase is guaranteed, we search the space of potential optimizations using a backtracking algorithm which, at a given iteration, determines the cost of the corresponding intermediate graph using programmer-defined heuristics in the form of a cost-model. To test our optimization strategy, we plan to implement our own graph transforms in Open Neural Network Exchange (ONNX) and we limit our performance heuristics to simply measuring inference time. So far we have created arbitrary computation graphs in ONNX and furthermore successfully implemented the 'enlarge kernel size', and 'fuse conv layers' transforms on it. We have implemented a Floating Point Operations Per Second (FLOPS)-based cost-model to benchmark our model without the need to actually run it. It is fairly accurate in estimating real inference times as we detail in our results. All this is integrated into a cost-limited backtracking algorithm which programmatically discovers new relaxed graph substitutions. Further we have unpacked the theoretical the details of two more transforms - fusing two convolutional layers and fusing an add and convolutional layer.

Contents

List of Figures

Chapter 1

Introduction

In most Deep Learning (DL) frameworks, training and inference are both represented using *Computation Graphs*, where nodes represent operations, and edges represent the flow of data through those operations. Particularly, we can think of each incoming edge for some node N representing arguments to N, each argument being the result of computing the respective subgraphs they are the roots of. Computation graphs, like other programs, can usually be optimized for some metric such as optimal memory usage or efficiency, and has motivated a vast body of literature surrounding the same. That being said, most, if not all, optimizations being worked on so far have been *greedy* in nature. i.e. They optimize a program in *passes*, where each pass, i, goes over the program in order to make some transformation T_i, such that T_i *must* lead to a strict performance increase.

The described constraint restricts the space of transformations an algorithm must explore, and, presumably, sacrifices a hard-to-discern, but "globally optimal" solution for a possibly locally optimal option that is guaranteed to improve performance within some time constraint. The idea behind [13] is that it's very feasible to search in a much bigger search space of what are called *relaxed graph substitutions*, leading to a discovery of complex substitutions that go through potentially sub-optimal intermediate steps (hence, "greedy") to reach an ultimate resultant computation graph that is better than what we would have achieved using greedy strategies.

The approach presented here is a backtracking algorithm that searches the exhaustive space of intermediate graphs obtained by applying all possible transformations on them (regardless of reducing performance, assuming the reduction in performance is below a set tolerance criteria). To quantify the performance of these intermediate graphs, so as to compare it with its predecessor, successor and other interemdiate transformations, we also need a *cost model* that allows us to statically estimate the same.

1.1 Objectives

Our objectives with this project are two fold:

1. Replicate results obtained by [13], but without the graph splitting algorithm (used to scale

the aforementioned non-greedy approach for large graphs)

2. Flesh out the *cost model* which statically estimates performance metrics on intermediate graphs.

The report is organized as follows: Chapter 2 gives some theoretical background on deep learning compilers and our motivation for exploring this particular thread of research. Chapter 3 gives some detail about our algorithm, including an in-depth explanation on a running example that we plan to impement as a demonstration of the algorithm. Chapter 4 provides some insight into possible challenges one can face when implementing such a scheme, along with future directions we plan to follow. Chapter 5 summarizes and concludes our exposition.

Chapter 2

Theoretical Background and Literature Review

2.1 Deep Learning Compilers and Motivation

In this section, we give an overview of the general landscape of deep learning compilers, and motivate the need for the endeavor we choose to embark on. Deep Learning compilers essentially aim to specialize existing, but hitherto general, compilation techniques, so as to perform optimizations specific to deep learning and machine learning workloads. Here by *general*, we mean techniques that can be applied across the board for any language as long as it's first converted to some suitable intermediate representation (assuming the transforms need such a requirement, such as the Intermediate Representation (IR) being LLVM IR[16]). Since deep learning computational graphs are usually very uniform and limited in terms of their basic operations, optimizing specifically for them becomes not only reasonably tractable but also beneficent.

2.1.1 Intermediate Representations and The current Optimization Scenario

According to [18], a rather generalized pipeline for compilation frameworks, which are much more amenable to such specializations as deep learning, is emerging in the form of MLIR [17]. The notion behind such a pipeline can be described in the following way:

- Computation graphs are usually defined at a coarse granularity consisting of abstract operations such as convolutions and pooling. While improvements can be made by altering graphs on this level, the abstraction makes doing so quite restricting.
- On the other hand, when compiled to intermediate representations such as LLVM IR, the operations not only lose their abstraction, but they plummet to a level no longer reminiscent of the domain they are actually specific too, and so nothing can be done beyond general optimizations whicha are anyways done on general purpose languages like C++ or Java. The idea is that different sized "chunks" of this IR represent algorithms or structures that are (i) used exclusively in our domain, such as a convolution algorithm (ii) are at an abstraction level lower than or equal to that of computation graphs. If we could somehow expose these "chunks" so that they were explicit operations at a level of

abstraction below computation graphs, but above machine code, opportunities exist for much more optimization!

- MLIR does exactly this, by allowing users to specify multiple, custom, intermediate representations, called *dialects*, which can be deployed seemlessly into a pipeline where one can iteratively lower from one IR to another, doing any necessary processing on the way.
- Having these different IRs, exposes potential for optimizations at different levels.

Many deep learning compilers [24] [21] exploits one of these "levels" of interemediate representations to do domain specific optimizations. If done at a lower level, they could be optimized for, say, deep learning hardware like Google's Tensor Processing Units (TPUs) [15], and if done at a higher level, they may represent, say, operations such as fusing two convolutions.

2.1.2 Motivation for Non-Greedy Approaches

Research in deep learning compilers is vast and rapidly growing. The following are some directions in which recent work has been done:

- **Optimizing for domain-specific hardware, like TPUs.** This includes the creation of IRs specific to an application or hardware. For example, the work by Hu et al.[12] defines two new MLIR dialects, TOP and TPU, that successively perform different kind of optimizing passes specific to a TPU. Given their passes are atomic and by definition "optimizing", their approach is also a greedy one. Furthermore, OpenVINO [6] by Intel optimizes models specifically for Intel hardware.
- **Evolving algorithms that specialize on particular kinds of inputs.** For example, Graph Neural Networks [27], which are build specifically to make explicit, graph-like relations in a given problem. These techniques have optimizations inherent in the actual Deep Neural Network (DNN) rather than doing them as some sort of post processing, which means greedy or non-greedy algorithms don't come into the picture.
- **Optimizing Distributed DNN training.** Recent advancements have introduced deep learning frameworks designed to automatically identify efficient parallelization strategies for distributed DNN training. For instance, ColocRL [19] employs reinforcement learning to determine optimal device assignments for model parallelism across multiple GPUs. Similarly, FlexFlow [14] explores an extensive range of parallelization strategies and utilizes randomized search to identify effective solutions within this space. However, these works all assume a fixed computation graph, and find ways of distributing said graph better, which doesn't consider a reduction in the inherent computational cost of deep neural networks, and hence doesn't fit into the direction we wish to explore.
- **Verification and Repair of Deep Neural Networks**. By its nature, verification involves giving theoretical guarantees on certain optimizations, which means the prospect of "al-

lowing degradation" doesn't come into the picture, unless it is a part of the proof that proves the improvement. An example is [23].

With this literature survey, however, we found that the assumption relating to greedy transformations held true for many cases, and little information was available on non-greedy approaches to optimizing programs. Furthermore, most, if not all, of the aforementioned threads of research already have working implementations on well-reputed libraries such as TensorFlow [4], PyTorch [8], and ONNX Runtime [9] and hence, wouldn't benefit much from another nudge in that same direction. With that in mind, we aim to explore this space of greedy optimizations and replicate results given in [13].

2.1.3 Estimating Computational Graph Costs

One of the primary components of our work is a program that takes a DL computational graph as input, and gives us, as output, an estimation of its execution time, without actually running the graph. This component is called the *cost model*. While the cost model is essential to this project (as we'll soon see), estimating computational graph costs is a problem that has been explored many times before (as we shall see in a moment) for the following reason: When neural network architectures have a large inference time, testing various architectures for any given application, so as to select the optimal architecture, becomes a slow and tedious task. Being able to estimate graph costs without running the actual inference, hence, is a valuable tool to have at one's disposal. We can categorize the approaches taken for estimating neural network inference times into two broad categories: (i) Hardware agnostic and (ii) Hardware sensitive

1. The tools that are hardware agnostic estimate graph costs without considering hardware-level intricacies that play a role in the performance of certain algorithms. For instance, the Very Large Instruction Word (VLIW) length or specific Instruction Set Architecture (ISA) instructions might make an algorithm run faster on one architecture compared to the other. If we were to consider these, the developer using this tool would have to specify the required hardware parameters such as bandwidth, cores, etc. before using the tool to estimate costs. Our approach in this paper has been majorly derived from the work done in [22] and [20]. We consider aspects that may affect performance at a granularity coarser than architectural details. This has an advantage in that the tool may be used for all platforms with minimal overhead.

2. The lines between hardware agnostic and sensitive are a little blurred. ANNETTE [26] uses an approach, that, while hardware agnostic at its core, needs us to specify a lot of metadata about the device being used. It then takes about 3-5 days for the tool to benchmark appropriate hardware parameters which aid graph cost estimation. PerfNet [25] uses a Neural Network to estimate graph costs, and requires one to feed many details

about the device to run inference. The advantage of using a more hardware sensitive approach is, of course, that the finer granularity lends us greater precision in most cases.

That being said, the above approaches rightfully try to get their cost estimates as close to the benchmarked cost as possible. This is not a constraint for our application, because, as we shall see later in this report, we only care about the relative graph costs between a set of candidate graphs. This encouraged us to use a hardware agnostic, rather than hardware sensitive approach in our work.

Chapter 3

Proposed Work

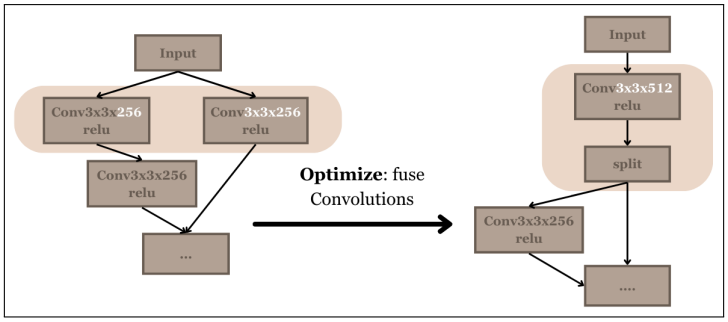
There are essentially three major facets involved in coming up with an implementation for relaxed graph substitutions: (i) introducing "downgrading" graph substitutions that are likely to reduce the graph's performance, (ii) a backtracking search algorithm that searches the set of all possible semantically correct graph transformation sequences, and (iii) the cost function we will use to guide our backtracking search. We now describe the motivation for each of these steps in detail.

3.1 Motivating our Approach Using an Example

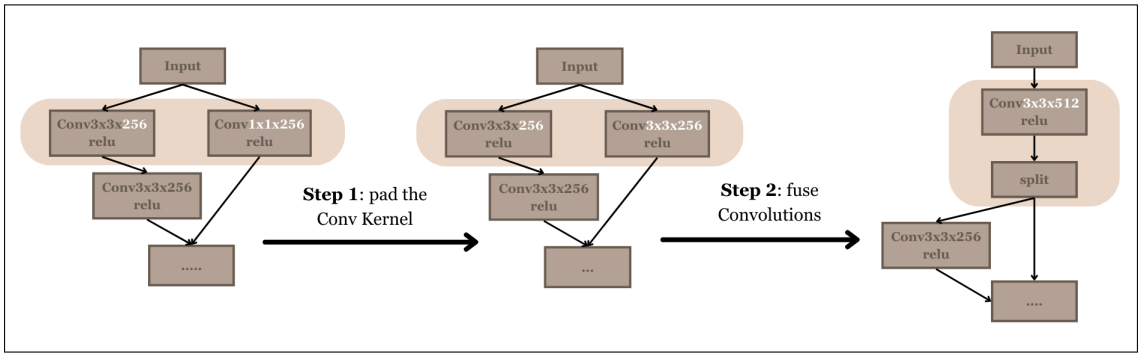
We use an example from a ResNet [11] module (Figure 3.1) to illustrate our strategy. Consider feeding the the first graph in Figure 3.1a to a state-of-the-art deep learning compiler such as TensorFlow RT. Such a compiler will detect that the two initial convolutions (say *Conv1* and *Conv2*) have the same attributes (kernel size, channels, input shape etc.), and can be *fused* into a single convolution combining the channels from both those convolutions.

However, what if the kernel sizes of *Conv1* and *Conv2* didn't match? In Figure 3.1b, the convolutions in the leftmost graph won't be fused because their kernel sizes aren't compatible (*Conv2* now has a 1x1 kernel instead of the earlier 3x3). That being said, there does exist a two-step method to optimize this graph: As the first step, we can pad the 1x1 kernel to form an *equivalent* 3x3 kernel — a step that causes a performance degradation due to increased computations. Doing so now allows us to fuse these kernels and save us the memory latency being spent on reading/writing tensor inputs/outputs. It so turns out, that the benefit obtained in Step 2 outweighs the loss incurred by step 1, resulting in a graph more optimal than our starting point.

Optimization algorithms in modern compilers won't do the aforementioned two-step process because they do not have the "foresight" to look beyond a single transformation step: If optimality in the next step is not a theoretical guarantee, that transform is not done even if though it may lead to a better globally optimal solution as discussed (the latter may or may not be a theoretical guarantee). In other words, these algorithms search for optimizing *transformations*, and not *sequences of transformations*. By definition, this is a *greedy* approach to



(a) Most Deep Learning compilers will detect pairs of convolutions having the same attributes, and fuse them into a single convolution as an optimization step.



(b) An example from [13] that demonstrates a relaxed graph substitution. If given the left graph, most compilers cannot perform the shown 2-step process to reach the optimized (right) graph, because the first step requires a "downgrading" graph transform.

Figure 3.1: Difference between regular and relaxed graph substitutions.

optimization, and severely restricts the space of opportunities as shown.

The above examples show two out of four total steps in this particular optimizing sequence. In the interest of brevity, we defer an explanation of the sequence to Appendix A.

3.2 Defining our Goals

In this study, we're particularly interested in the kind of optimizations facilitated by an initial reduction in performance, because removing this property leaves us with cases most compilers will optimize anyways. Therefore, our **first goal** is to introduce "deprecating" transformations in our list of optimizations. Once we have such a list of transformations, we can define our **second goal** as coming up with an algorithm that iterates through the space of transformation sequences with an aim of finding one that leads to a more optimal graph. That being said, knowing whether or not a given sequence is optimizing will require a comparison between initial and final computational graphs. Inference time cannot be used for such a comparison because the number of sequences to test will be very large. There needs to be a fast way to compare the efficiency of two computational graphs without running them. This brings us to our **third and most important goal** of defining a *cost model* that approximates the cost of a given computational graph. These goals are summarized in Table 3.1.

Description	Solution
Introduce "downgrading" graph transformations that may lead to a better optimizing sequence compared to the "regular" strategy. (These are the only kind of cases we're interested in investigating)	TensorFlow transformations + Custom ones
Explore the space of possible sequences with the complete (optimizing+downgrading) set of transforms	Backtracking Algorithm
Find a way to estimate one sequence is better than the other. (We can never know for sure without running)	Cost Model (our primary goal)

Table 3.1: Goals of This Study

3.3 The backtracking Algorithm

We take the following approach to search through the space of possible graph transformations:

1. Consider a list of possible graph transformations, T. At any given point in the algorithm, let $\mathcal{G}$ be the current intermediate graph version. We maintain a priority queue of intermediate graphs, ordered by their *cost*, defined by a heuristic (in our case, this is simply the execution time).

2. In each iteration:

   (a) We dequeue to get the next candidate intermediate representation $\mathcal{G}$.

   (b) For each optimization t_i in T, we apply t_i on $\mathcal{G}$ to obtain some equivalent graph $\mathcal{G}'$. If $cost(\mathcal{G}') < cost(\mathcal{G})$, we set $\mathcal{G} = \mathcal{G}'$. Else if $cost(\mathcal{G}') < \alpha \cdot cost(\mathcal{G})$, where α is a hyperparameter defined by the user (greater than one), then $\mathcal{G}'$ is considered to be a valid candidate for further search, and is added to the priority queue.

   (c) α helps define how much of a performance decrease we can tolerate. This value can't be too high so as to avoid blowing up the search space, and hence, the time it takes for our algorithm to find a beneficial sequence of graph transformations.

3. The above is repeated until there are still graphs to process in the queue.

Here, in 2(b), the factor α can be thought of as the amount of performance degradation we can tolerate for any given transform. This limit is important to cut down the space of possible sequences to a pragmatic value. The complete process can be seen in Algorithm 1

Algorithm 2 The backtracking algorithm from [13] used to find an optimizing sequence.

Require: An initial model G_0, a cost function $\text{Cost}(\cdot)$, a set of valid operations $\{S_1, \ldots, S_m\}$, and a hyperparameter α.

Ensure: An optimized model G_{opt}.

Initialize a priority queue Q, where models are sorted by $\text{Cost}(\cdot)$.
$Q \leftarrow \{G_0\}$
$G_{\text{opt}} \leftarrow G_0$
$C_{\text{opt}} \leftarrow \text{Cost}(G_0)$
while $Q \neq \emptyset$ **do**
  $G \leftarrow Q.\text{dequeue}()$
  for each operation $S_i \in \{S_1, \ldots, S_m\}$ **do**
    Apply S_i to G to generate a new model G'
    if $\text{Cost}(G') < \text{Cost}(G_{\text{opt}})$ **then**
      $G_{\text{opt}} \leftarrow G'$
      $C_{\text{opt}} \leftarrow \text{Cost}(G')$
    end if
    if $\text{Cost}(G') < \alpha \cdot C_{\text{opt}}$ **then**
      $Q.\text{enqueue}(G')$
    end if
  end for
end while
return G_{opt}

3.4 The Cost Model

When comparing the expected performance of intermediate graphs, $\mathcal{G}'$, we cannot simply run the inference on them and compare execution times because an exhaustive search will go through a vast number of potential graphs. Running them will not only result in an enormous amount of computation overhead, but also might involve convoluted problems related to memory latencies and slow I/O operations, making such a strategy quite impracticable. Furthermore, it might be better to have an algorithm that is more amenable to a flexible cost function, that optimizes for not necessarily execution time, but also other metrics such as Floating Point Operations (FLOPS) or Memory accesses. As a result, a flexible, static cost estimation strategy must be developed.

Estimating the cost would ideally have to be done by taking many hardware factors into consideration, since the Instruction Set Architecture (ISA) and the nature of the processors would greatly affect what kind of computations can be done faster, say. However, fine-tuning algorithms to specific hardware is not only tedious but also time consuming, and may require the user of the cost model to provide a lot of architectural data for it to run, which makes

it inconvenient to use (albeit, more accurate). Efforts have been made recently to use both Deep Learning techniques [25], as well as hardware agnostic, benchmark-based techniques [22][26][20] to estimate the cost model. Due to their simplicity and deterministic nature, we chose to explore a hardware-agnostic and benchmark-based technique to do the same.

Our approach for the cost model can be divided into three facets:

1. Estimating the costs for individual operations, such as convolution or pooling layers.
2. Estimating the memory latency incurred when reading and writing intermediate results
3. Estimating how the individual costs add in the presence of model parallelism

3.4.1 Estimating Layer Cost

An ideal but impractical method to estimate layer costs would be the following: for every possible layer (Conv, Pooling etc.), we store a table that enumerates all possible attribute values to that operator, along with their corresponding benchmarks. Estimating the cost of a layer would be a simple lookup in this table. While this is not possible, it is indeed possible to benchmark each layer with some set of attributes, and extract some trend that emerges. With that in mind, we describe our method of estimating layer costs, inspired from [22]

We can simplify this estimation by first taking (i) the best possible runtime for that layer given it's computational FLOPS and the peak CPU Floating Point Operations Per Second (FLOPs) [1], (ii) multiplying that with *the percentage of peak performance it usually runs at*. The latter was called the *Platform Percentage of Peak* (PPP) by [22]. We first estimate PPP by benchmarking that layer over a range of attributes, and then use its value for estimating the cost for an unknown set of attributes. This can be summed up in the following equations:

$$PPP = \frac{bestPossibleTime(layer)}{benchmarkedTime(layer)} \tag{3.1}$$

$$bestPossibleTime(layer) = \frac{FLOPS(layer)}{PEAK\_CPU\_FLOPS} \tag{3.2}$$

$$estimatedTime(layer') = \frac{FLOPS(layer')}{PEAK\_CPU\_FLOPS \cdot PPP} \tag{3.3}$$

Here, Equation 3.1 is curated per layer, and calculated over many iterations, with different attributes in each iteration.

Benchmarking and Relative Costs

One thing to keep in mind, is that our cost model only has to be accurate in a relative sense, which is to say:

[1] Note that we use FLOPS, with a capital S, to represent just the total floating point operations in some computation, and use FLOPs, with a small s, to represents floating point operations per second.

This approach is different from a linear interpolation-based approach used in [20], in that the latter tries to *directly* estimate the layer cost, given an extensive set of benchmarks. According to their work, a major limitation was the fact that benchmarks when integrated within a model gave a different cost compared to benchmarks that measured that layer in isolation; giving an inaccurate cost estimation. Since our cost model is dependant not on an absolute measure of cost, but rather costs of graphs when compared to others, this inaccuracy can be tolerated. In other words, as long as the following inequality is satisfied, our cost model will perform well:

$$estimatedCost(\mathcal{G}_1) < estimatedCost(\mathcal{G}_2) \implies actualCost(\mathcal{G}_1) < actualCost(\mathcal{G}_2) \quad (3.4)$$

With this in mind, we adopted the benchmarking strategy used in [20] for the operations we did run one on by embedding the layer to be benchmarked within a three-layer model. We used the ONNX Profiler [10] for extracting the time taken for a particular layer to run.

3.4.2 Estimating Memory Latency

We use here, a very simple heuristic to model writes and reads to memory in between layers. We assume that the models are being run on the same device (perhaps, on different cores), and estimated the cost using the following equation:

$$time = \frac{sizeOfTensorInBytes}{Bandwidth} \quad (3.5)$$

Here, we estimated the bandwidth using the mbw [7] linux utility, averaging it over three possible algorithms to copy data to and from memory. Improvements to this memory model, potentially in the direction of modelling cache misses, is a promising future direction.

3.4.3 Estimating Model Cost

With parallelism being the norm, most Deep Learning models run over multiple threads, with either data or the layers themselves distributed over different threads. The former is referred to as Data Parallelism, and the latter as Model Parallelism. While it is obvious that modeling such a property would be desirable, it is not very obvious why it could be crucial. The reason for the same can be summed up using the following inequality:

$$costSerial(\mathcal{G}_1) < costSerial(\mathcal{G}_2) \not\Longrightarrow costParallel(\mathcal{G}_1) < costParallel(\mathcal{G}_2) \quad (3.6)$$

Some algorithms could perform better when parallelized, and others could perform better serially. Furthermore, the extent to which something is parallelized could further modulate how the algorithm behaves. Such fine grained modeling is out of the scope of this study, and has been left as a future direction.

Our estimation of model cost has been inspired from [22] and can be shown using Figure

3.2. It is a general truth that sequential layer costs can simply be added to get a reasonable estimate for total cost [20]. We hence, show through the beige background, regions that have been grouped together so as to form a sequence of *supernodes* whose costs can simply be added to get the total cost. Within such supernodes, for every branch encountered, we assume that the multiple branches will be run on separate threads. In this case, the minimum possible execution time will be max(thread1, thread2), and the maximum possible execution time will be a sum of both threads' execution times.

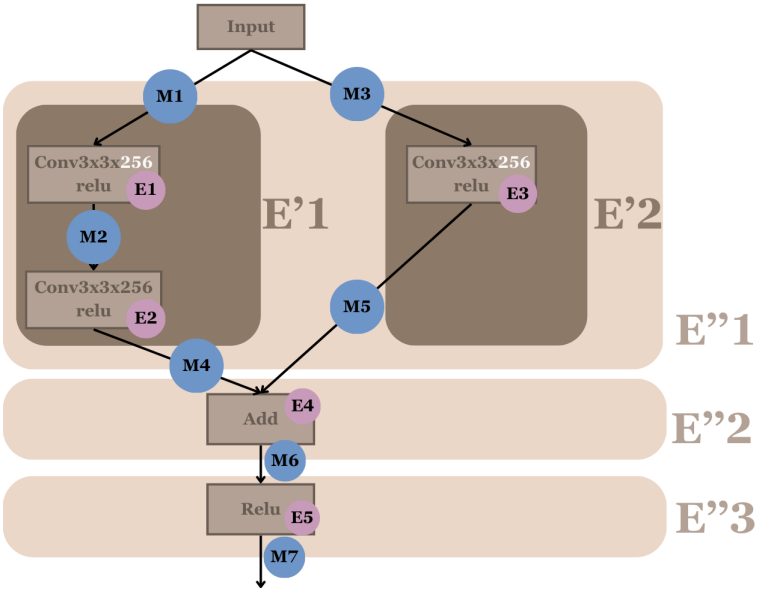


Figure 3.2: The ResNet example, with pink squares showing super-nodes

So far, our analysis is not fine-grained enough to determine where in this range the actual execution time lies, so we're currently taking an average of both values. Therefore, the execution time for this example can be summed up using the following equations (we've left out the trivial ones):

$$E_{total} = E_1'' + E_2'' + E_3'' \tag{3.7}$$

$$E_1'' = Avg(max(E_1', E_2'), sum(E_1', E_2')) \tag{3.8}$$

$$E_1' = E1 + M2 + E2 + M4 \tag{3.9}$$

$$E_2' = E3 + M5 \tag{3.10}$$

Algorithm 3 gives the entire process for consolidating layer (computation) and memory costs into the total cost of the graph.

3.5 Implementation Details

We selected Open Neural Network Exchange (ONNX) as our tool-of-choice for defining, modifying, and benchmarking computation graphs. Following are the reasons why:

Algorithm 3 Compute Total Graph Cost

Require: Number of nodes n, adjacency list $graph$, parent mapping $parent$, initial indegree $indegree$, node computation costs $comp\_cost$, node memory costs mem_cost
Ensure: Total cost $total\_cost$

Initialize $total\_cost \leftarrow 0$
Copy $initial\_indegree \leftarrow indegree$
Initialize $cost\_store[node]$ for all nodes, with fields `max_cost` and `sum_cost`
Initialize a queue $queue$
for each node i in $graph$ **do**
  if $indegree[i] = 0$ **then**
    $total\_cost \leftarrow total\_cost + comp\_cost[i] + mem\_cost[i]$
    Enqueue i into $queue$
  end if
end for
while $queue$ is not empty **do**
  if $indegree[current] = 0$ **then**
    Dequeue $current$ from $queue$
    $current\_cost \leftarrow comp\_cost[current] + mem\_cost[current]$
    $parent\_sum \leftarrow 0$, $parent\_max \leftarrow 0$
    for each $p \in parent[current]$ **do**
      $parent\_sum \leftarrow parent\_sum + cost\_store[p].sum\_cost$
      $parent\_max \leftarrow \max(parent\_max, cost\_store[p].sum\_cost)$
    end for
    $current\_cost \leftarrow current\_cost + \frac{parent\_sum + parent\_max}{2}$
    Update $cost\_store[current].max\_cost \leftarrow \max(cost\_store[current].max\_cost, current\_cost)$
    Update $cost\_store[current].sum\_cost \leftarrow cost\_store[current].sum\_cost + current\_cost$
  end if
  for each $child \in graph[current]$ **do**
    $indegree[child] \leftarrow indegree[child] - 1$
    if $indegree[child] = 0$ **then**
      if $initial\_indegree[child] > 1$ **then**
        $ancestor \leftarrow LCA(parent[child])$
        for each $par \in parent[child]$ **do**
          $cost\_store[par].max\_cost \leftarrow 0$
          $cost\_store[par].sum\_cost \leftarrow cost\_store[par].sum\_cost - \frac{(cost\_store[par].sum\_cost + cost\_store[par].max\_cost)}{2.0}$
        end for
      end if
      Enqueue $child$ into $queue$
    end if
  end for
end while
for each node i in $graph$ **do**
  if $indegree[i] > 1$ **then**
    $total\_cost \leftarrow total\_cost + \frac{cost\_store[i].max\_cost + cost\_store[i].sum\_cost}{2} + mem\_cost[i]$
  end if
end for
return $total\_cost$

- **Standardization**: ONNX is an **open standard** that defines a common set of operators and a common file format to represent deep learning models.
- **Cross-framework compatibility**: A computation graph in ONNX can easily be converted to TensorFlow [3] and PyTorch and vice-versa [8][1].
- **User-Friendly API**: ONNX has an easy-to-use Python library featuring convenient APIs for creating and modifying computation graphs.

3.5.1 Initial Graph Construction

The initial phase of our implementation involves constructing a computation graph starting with the definition of an input tensor. This input tensor serves as the data that flows through the computation graph. ONNX provides the **TensorProto** class, which supports the specification of a generic tensor of an arbitrary shape.

Following the tensor definition, we define each node-type that will serve as a component in our graph. For our purposes, we make use of Convolution (Conv), Rectified Linear Unit (ReLU), and Addition (Add) nodes. ONNX simplifies this process by offering predefined operator types, referred to as "op-types" [2], which encapsulate these operations in a standardized and efficient manner.

The directed acyclic graph (DAG) is constructed by designating the appropriate nodes as inputs and outputs for other nodes. ONNX subsequently verifies that the data flow adheres to a topological order before producing the final graph. This modular approach to graph construction makes our implementation flexible and amenable to further changes. The output of our construction is shown in Fig 3.3.

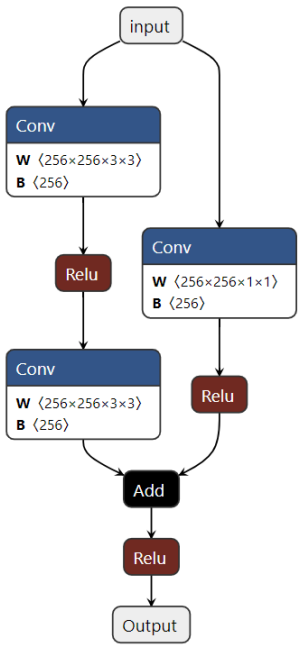


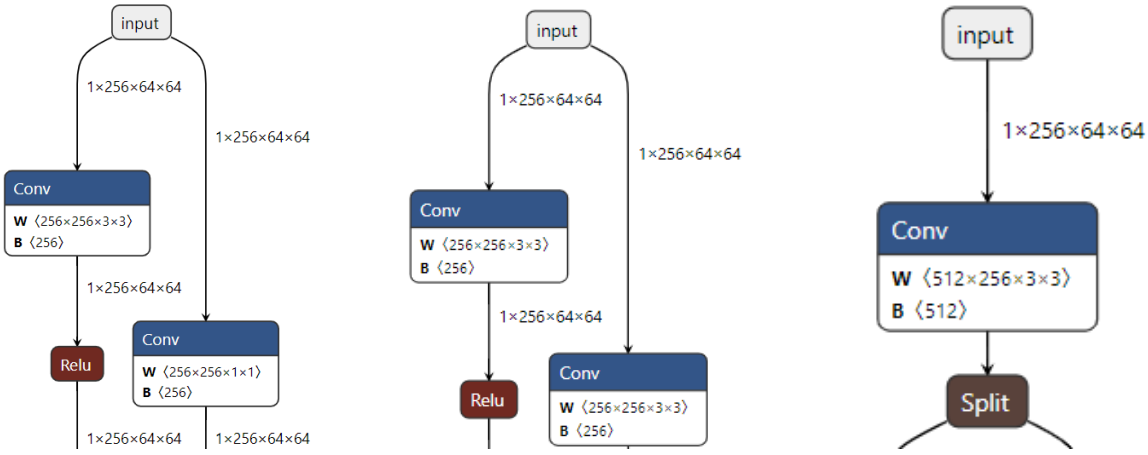
Figure 3.3: Initial computation graph as constructed using ONNX

3.5.2 Backtracking and Visualizing the Optimization Strategy Chosen

We now highlight some key implementation factors associated with the backtracking algorithm shown in Algorithm 1 along with an approach to (optionally) visualize the optimization steps chosen by the model.

- **Initialization**: The algorithm begins by initializing a priority queue (**heapq**) with the initial model and its cost.
- Whenever a new instance of the model is initialized with the values of an already existing one, a **deep copy** is used.
- We use deep copying because, in languages like Python, when we instantiate an object and assign it to a variable, that variable is simply a pointer to that object in the heap. If we then initialize a new variable with this object, both variables will point to the same region in memory. Thus, to avoid unintended modifications, we use deep copying.
- If visualisation is enabled, each transformation step within the threshold is saved and rendered using Netron. The images are annotated to indicate the applied operation and then combined into a single timeline for visual representation.
- A list called **path** is maintained storing the source model, the model obtained after transformation, and the name of the transform applied. This list is later iterated through to get the optimization steps taken up the model.

The output visualising the optimization steps taken up the model is shown in Fig 3.4.



(a) Initial Graph (b) Enlarge Conv Kernel (c) Fuse Conv Layers

Figure 3.4: Optimization Steps Showing Key Changes

3.5.3 Cost Model Implementation Details

Since the computation graph is represented as a Directed Acyclic Graph (DAG), it has certain dependencies that need to be computed before the current node. Therefore, the algorithm utilizes topological sorting along with the Lowest Common Ancestor (LCA) technique to achieve the correct computing order of the nodes.

For a Directed Acyclic Graph, its topological sort gives a linear ordering of the nodes such that, for a directed edge $u \rightarrow v$, u should appear before v. It is commonly used in scheduling tasks where certain tasks must be done before others.

To estimate the parallelism of the model in the algorithm 3, we assume that the system supports an infinite number of threads. Based on this assumption, the algorithm determines the Lowest Common Ancestor (LCA) of all the predecessors of the current node. This helps identify possible paths from the common ancestor node to the current node, which can then be parallelized.

The algorithm uses a *structure* to store both the maximum cost (complete parallelism) and the total cost (no parallelism). As an initialization step for the algorithm, all nodes that do not have any predecessors, and thus have no dependencies, are inserted into the queue for computation. Additionally, a variable $cost\_store$ is used to store all the intermediate costs that are calculated.

The algorithm dequeues a node from the queue and checks if its in-degree is 0. If so, it updates the values of the current node in $cost\_store$. Furthermore, the in-degree of all child nodes is reduced by 1. If the in-degree of a child node becomes zero, the child is inserted into the queue for its computation. If the initial in-degree of a child node was greater than 1, the computation cost is updated in $cost\_store$, with the maximum cost set to 0 and the total cost set to the parallelized cost up to that point.

1. A node is dequeued from the queue.
2. The algorithm checks if the in-degree of the dequeued node is 0
   (a) If true, it updates the values of the current node in $cost\_store$.
3. The in-degree of all child nodes of the dequeued node is reduced by 1.
4. For each child node:
   (a) If the in-degree becomes 0, the child node is inserted into the queue for computation.
   (b) If the initial in-degree of the child node was greater than 1, the computation cost is updated in $cost\_store$:
      i. The maximum cost is set to 0.
      ii. The total cost is set to the parallelized cost calculated up to that point (i.e. the average of total cost and max cost).

This process is repeated until there are no nodes left to compute.

Chapter 4

Results

The complete ResNet example can be seen in Figure A.1. This sequence of transformations has 4 stages, all four of which are custom transformations not available in ONNX Runtime. We have implemented the first two using the ONNX API, and integrated it with the backtracking algorithm along with the cost model. Doing so allows us to programmatically follow this path, given a sound cost model. Due to the unavailability of transforms 4 (Fuse Conv and Add) and 5 (Fuse Conv and ReLU), we separately ran our cost model on them to get results as shown in Table 4.1.

Keeping in mind that we only need to estimate accurate *relative*, and not *absolute* costs, we can see that our cost model *correctly predicts* an increase in cost for the first two transforms, and following that a decrease in cost for transform 3. This is clearly visible through Figure 4.1, which are the column 2 and column 3 table values plotted as blue and red lines respectively.

The graph after transform 3 still has a higher cost than the original model, both through our cost model and benchmarking. Our hypothesis is that its the last transform that pushes the models performance and makes it better than the original graph. We could not estimate the cost for this last graph for the following reason: ONNX does not implement a node that fuses Conv and Relu, which means the only way to create such a model would be to change ONNX's source code. While this prevents us from seeing if our sequence is indeed optimizing, it does give us positive reinforcement with regard to our cost model, which seems to be predicting relative costs fairly accurately.

As a second example, consider the following equations which show how to optimize the computation involved in a Simple Recurrent Unit as detailed in [13]:

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

Graph	Actual Cost (s) (Benchmarked)	Estimated Cost (s)
Original Model	0.006753	0.0237696
After Transform 1	0.007033	0.0265976
After Transform 2	0.007072	0.0291380
After Transform 3	0.006931	0.02833681

Table 4.1: Running our cost model on the ResNet example. All benchmarking was done using an NVIDIA GeForce RTX 3050 GPU.

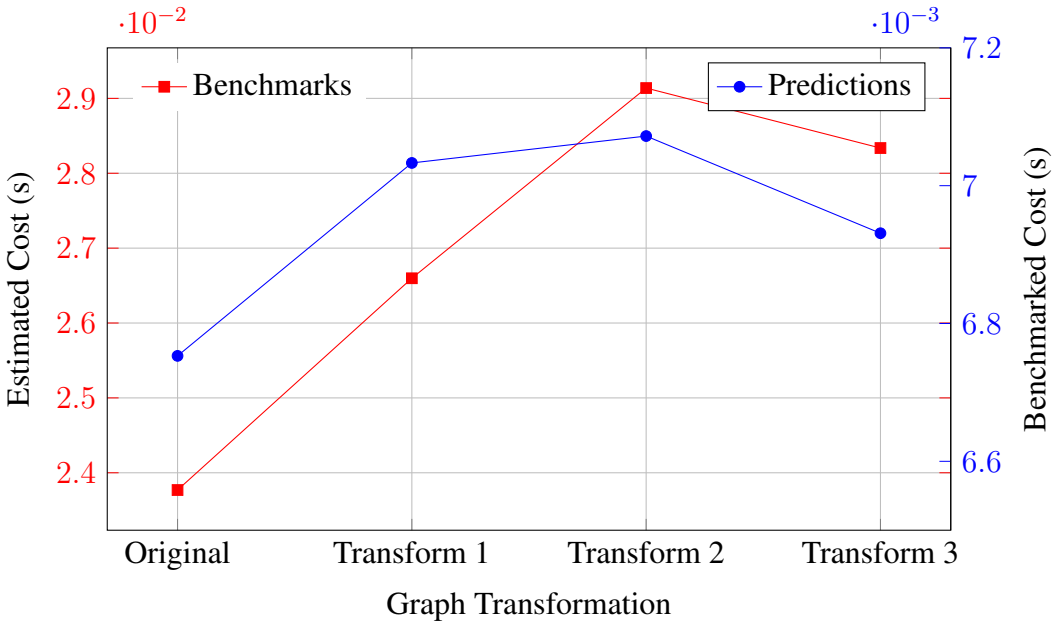


Figure 4.1: Comparison of estimated and actual costs with separate scales.

Graph	Benchmarked Cost (s)	Estimated Cost (s)
Original Model	0.000706	0.00036923
After Transform 1	0.000780	0.00054212
After Transform 2	0.000550	0.00050784
After Transform 3	0.000446	0.00031553

Table 4.2: Running our cost model on the Recurrent Neural Network example. All benchmarking was done using an NVIDIA GeForce RTX 3050 GPU

We begin with a computation that involves 4 operations. Then, the first graph transform increases the number of operations from 4 to 5. This is a degrading transform as the number of operations have increased and we observe a decrease in performance as a result, as shown in Table 4.2. However, doing so allows us to discover the form that involves only 3 operations. This would never have been discovered without the initial degrading transform. Thus, this is an application of relaxed graph substitutions at play. We can observe that the final graph has a lower inference time than the initial, showing that the transforms have indeed performed a valid optimization. This has been visualized in Figure 4.2.
As the main objective was to show that given our backtracking search algorithm (see Algorithm 1), our cost model is capable of guiding the search to a valid optimization path, we did not focus on implementing the transforms themselves for this example. Each of the 4 forms of the equation have been modeled manually in ONNX and then benchmarked using our cost model. We can see in Figure 4.1 that our cost model accurately portrays the increases and decreases in real inference time, lending accurate ordinality to our measures and helping guide our search to the most optimal model.

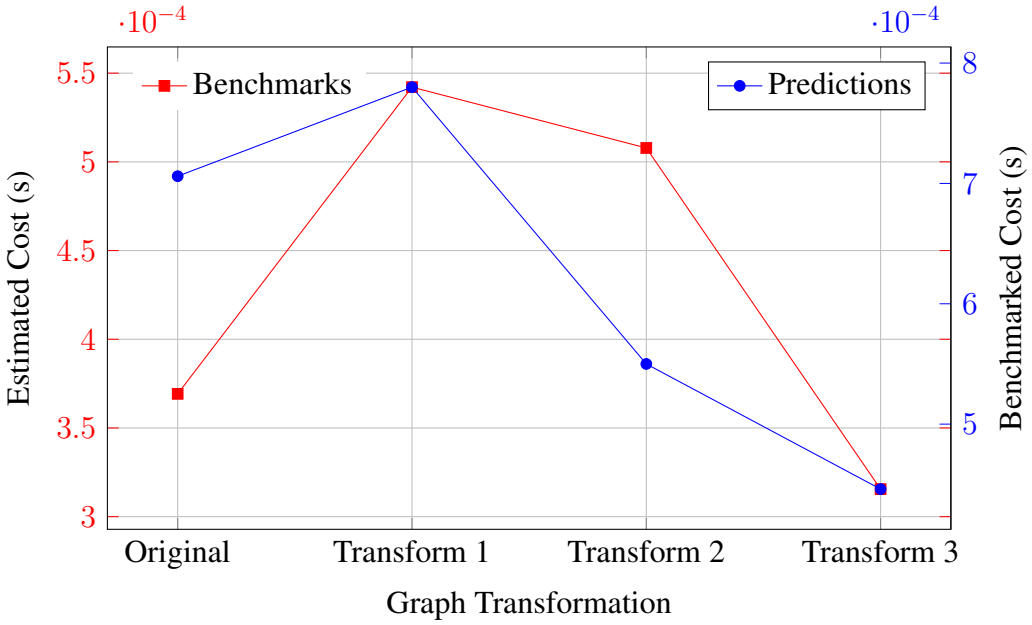


Figure 4.2: Comparison of estimated and actual costs with separate scales.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this report, we explained the notion of non-greedy optimizations, motivated their need, and gave an exposition on our attempt at implementing the same, with our work being based on [13]. We saw, using an example, how by allowing intermediate results that are less optimal we can find paths that ultimately lead to graphs better than what we would have obtained using a greedy approach.

Because individual heuristics for such special cases would lead to a convoluted software, a backtracking algorithm is used to exhaustively search the entire space of graphs enabled by a predetermined set of transformations. At a given iteration, the backtracking algorithm estimates the performance of an intermediate result using a *cost model*, and uses a parameter α to set how relaxed of a substitution we can tolerate, based on factors such as external time-bounds or performance bounds.

The most crucial part of this work is the cost model. We divided the cost model into three facets:

1. Estimate the cost of an individual operator: we do this by (i) finding its lowest possible runtime, t_{opt}, by dividing its FLOPS with peak CPU flops, and (ii) dividing this value with the platform percentage of performance(PPP).

2. We estimate the cost of intermediate writes to and reads from memory, by simply dividing number of bytes with bandwidth, and where bandwidth is estimated using the mbw linux utility.

3. We then consolidate the layer and memory costs through a heuristic that models model parallelism.

Finally, we compare the estimations made by our cost model to the actual benchmarked values, and see that it is fairly accurate at determining relative costs, which is sufficient for our purposes.

Finally, we present the results comparing the benchmarks obtained via our cost model with actual inference time on real-world hardware.

5.2 Future Work

This work holds a lot of potential yet to be manifested, and we briefly explain the future directions as follows. As discussed in the results section, we could not test our cost model on graph 5 in the chain because ONNX doesn't support a fused Conv+Relu node. We wish to hence either (i) facilitate this node within ONNX Runntime, or (ii) Port our cost model to Tensorflow RT which *does* implement such a node. We also wish to discover and add more potential downgrading transformations to improve the variety of sequences searched. Our memory model can also be improved, potentially by modeling the memory hierarchy imposed by caches and understanding the ONNX Runtime implementation better for a better FLOPS estimate. Finally, our cost model, while promising, has scope for improvement: We do not have a way of modeling optimizations that ONNX may do, which is to say, when using it one must disable optimizations manually, lest they get unreliable results. Furthermore, the algorithm that models parallelism currently assumes that there's always a new thread available — i.e. that there are infinite threads. This works alright when optimizing small graphs, but will be problematic when optimizing large graphs with many branches.

Our current efforts at implementation can be seen on our public GitHub repository [1].

[1] https://github.com/thehthakur/Take_It_Easy/tree/main

Appendix A

Diving Into The ResNet Example

A.1 Diving Into the Transformations

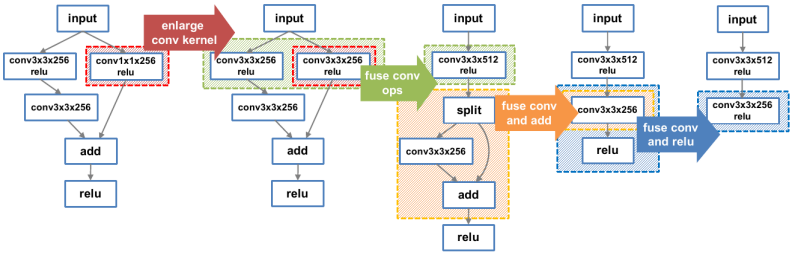
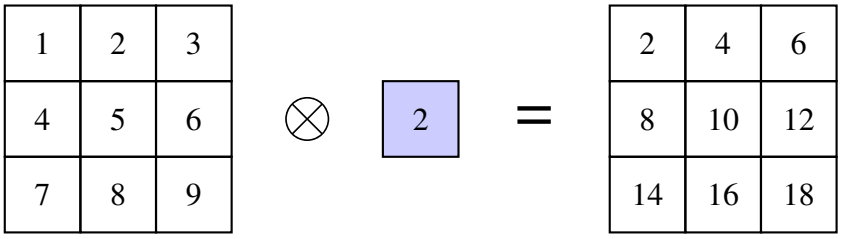


Figure A.1: An example introduced in section 3 and taken from [13]

Initially, we wanted to look for graph transformations that were already available in any of the deep learning frameworks, such as PyTorch, TensorFlow, etc. But after going deep into the implementations of the transformations provided by the TensorFlow XLA, we noticed that every transformation provided by the deep learning frameworks always improves performance and never degrades it, which dismisses the prospect of relaxed substitutions. Hence, we decided to implement the transformations provided in the [13], as shown in figure A.1.

A.1.1 Kernel Expansion



Input Matrix (3 x 3) Kernel Matrix (1 x 1) Output Matrix (3 x 3)

Figure A.2: Convolution Operation with a 1x1 kernel matrix, a padding of zero, and stride 1

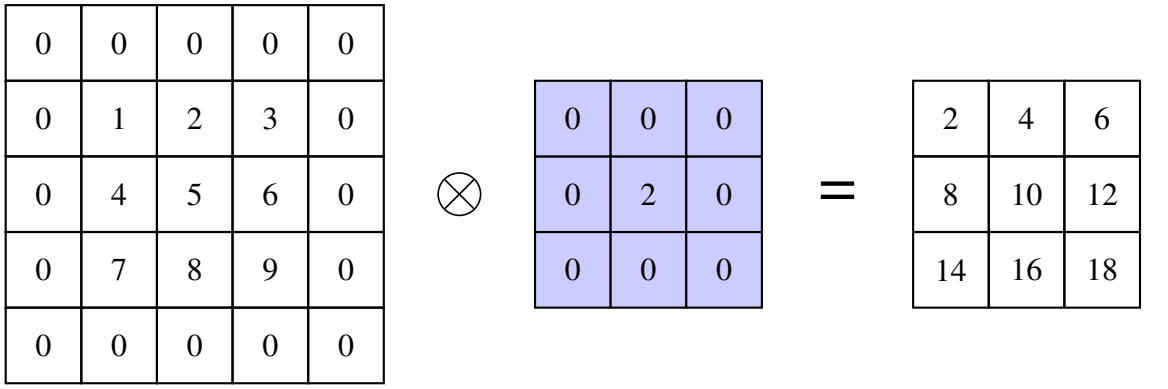


Figure A.3: A convolution operation after padding both, the kernel and the image.

The substitution `enlarge_conv_kernel` increases the kernel size from 1x1 to a larger 3x3 size. This change enables further transformations that can lead to improved performance. However, as previously mentioned, simply increasing the kernel size does not guarantee correctness.

Let us assume the initial kernel size is f, and the final kernel size is $f + k$, where we are increasing the kernel size by k. The initial padding is p, and the final padding is p'. The input size is n.

$$\frac{n + 2p - f}{s} + 1 = \frac{n + 2p' - (f + k)}{s} + 1 \tag{A.1}$$

$$2p - f = 2p' - (f + k) \tag{A.2}$$

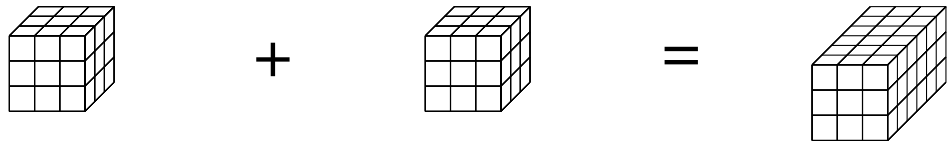
$$\boxed{p' = p + \frac{k}{2}} \tag{A.3}$$

Hence, enlarging the kernel will also require adding padding to the input in the convolution operation to maintain the same output dimensions.

A.1.2 Convolution Fusions

Convolution Fusions combine multiple convolution oeprations into a single, more efficient oepration. This approach aims to reduce the number of operations and memory access overhead by merging sequential convolutional layers.

The fusing of convolution layers can only be done if the following conditions are satisfied:



Kernel Matrix (3 x 3 x 3) Kernel Matrix (3 x 3 x 3) Output Matrix (3 x 3 x 6)

Figure A.4: Fusing two kernel matrices

Constraints on the Source Graph

- `conv1.kernel == conv2.kernel`
- `conv1.stride == conv2.stride`
- `conv1.padding == conv2.padding`

Construct the Target Graph

- `conv3.outChannels = conv1.outChannels + conv2.outChannels`
- `conv3.weights = concat(conv1.weights, conv2.weights)`
- `split.sizes = [conv1.outChannels, conv2.outChannels]`

A.1.3 Fusing Conv and Add Nodes

One of the more non-obvious and harder to unpack transformations proved to be the fusion of a Convolutional node and an Add node. The process is best understood by first revisiting concepts of Convolution networks.

- An image with $J \times K$ pixels and C channels will be described by a tensor of dimensionality $J \times K \times C$. We introduce a filter described by a tensor of dimensionality $M \times M \times C$ comprising a separate $M \times M$ filter for each of the C channels.
- Secondly, we introduce multiple such filters, each detecting a seperate abstract feature in the input image. The number of these specific filters are referred to as the channels of the

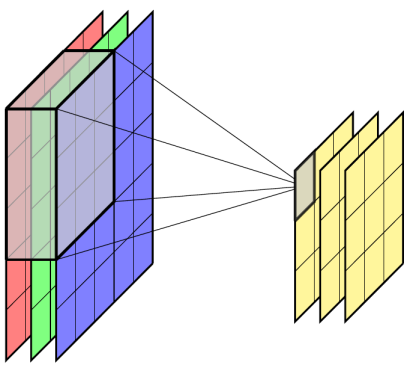


Figure A.5: An illustration from [5] that shows a multi-dimensional convolutional filter layer extended to include multiple independent filter channels.

kernel and are denoted by the third number in the notation $3 \times 3 \times 512$ for a convolutional layer.

- For each filter, after performing the convolution operation across all channels of the input image, the resulting C channels are summed to produce the final feature map representing the abstract feature the filter is designed to detect.

Now with the basis of the convolutional approach laid down, we can explore the optimization. The input is a $3 \times 3 \times 512$ image which was formed after fusing two $3 \times 3 \times 256$ convolutional operations together. The first 256 channels out of the 512 denote the input that came from Conv1 and the rest 256 denote the channels that came from Conv2. As Figure 3.1 shows, the expected behaviour is for the Conv1 channels to undergo one more convolution, say Conv4, and then simply get added to the Conv2 output.

The optimization here is to cleverly modify the kernel matrix of a new Conv operation, call it Conv3, that includes within it the Convolution operation on Conv1 channels and subsequent addition. The approach is as follows:

- In Conv3, there are 256 different filters. Each of these filters is responsible for extracting some abstract feature.
- Each of these filters has 512 channels, 256 devoted to processing the channels consolidated from Conv1 and the remaining 256 for processing the add.
- Note that each filter within the Conv4 layer has 256 channels within it to handle the 256 input channels from Conv1.
- The fusion is accomplished in two steps. The first is to have the first 256 channels of Conv3 be initialized with the weights of the corresponding channel in the $3 \times 3 \times 256$ Conv4.
- Now, we know that, per abstract feature, after carrying convolution over all channels, the final feature map is a result of summing up over all the channels. In our current arrangement the first 256 channels, when summed up, will give the output feature map corresponding to performing Conv4 for that abstract feature. The remaining part is to add to this sum precisely the channel that corresponds to Conv2, that, when added, will simulate doing a sum post convolution.
- The trick here is to use the **identity kernel**, which simply outputs the input as is without modification. It is also called a do-nothing kernel.

$$I = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

- Let us make a few constructs:

- Let 0 represent a 3x3 kernel filled with all zeros.
- Let each row of the matrix represent the kernel corresponding to one of the 256 abstract features in Conv3.
- Let $f_{i,j}$ denote a $k \times k$ filter for abstract feature i and channel j.

.

$$\begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} & \cdots & f_{1,256} & I & 0 & 0 & \cdots & 0 \\ f_{2,1} & f_{2,2} & f_{2,3} & \cdots & f_{2,256} & 0 & I & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ f_{256,1} & f_{256,2} & f_{256,3} & \cdots & f_{256,256} & 0 & 0 & 0 & \cdots & I \end{bmatrix}$$

- The above kernel matrix can be seen as the original Conv4 weight matrix concatenated with an "Identity" matrix. This identity matrix is actually a matrix of matrices where the diagonal elements are identity kernels.

By configuring the last 256 channels to either the identity matrix I or zero, as shown in the matrix above, we can selectively sum the layers that would have been added in the original computational graph.

Bibliography

[1] GitHub - onnx/tensorflow-onnx: Convert TensorFlow, Keras, Tensorflow.js and Tflite models to ONNX. `https://github.com/onnx/tensorflow-onnx`.

[2] ONNX Operators - ONNX 1.18.0 documentation — onnx.ai. `https://onnx.ai/onnx/operators/`. [Accessed 27-11-2024].

[3] onnx2tf. `https://pypi.org/project/onnx2tf/`. Command-line tool to convert ONNX files to TensorFlow.

[4] Martín Abadi. Tensorflow: learning functions at scale. *SIGPLAN Not.*, 51(9):1, sep 2016.

[5] Christopher M Bishop and Hugh Bishop. *Deep learning: Foundations and concepts*. Springer Nature, 2023.

[6] Intel Corporation. *OpenVINO, https://docs.openvino.ai/2024/index.html*, 2024.

[7] The Linux Foundation. *Memory Bandwidth Benchmarking, https://manpages.ubuntu.com/manpages/focal/man1/mbw.1.html*, 2019.

[8] The Linux Foundation. *PyTorch Docs, https://pytorch.org/docs/stable/index.html*, 2023.

[9] The Linux Foundation. *ONNX Optimizer, https://github.com/onnx/optimizer*, 2024.

[10] The Linux Foundation. *ONNX Profiling Tools, https://onnxruntime.ai/docs/performance/tune-performance/profiling-tools.html*, 2024.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Pengchao Hu, Man Lu, Lei Wang, and Guoyue Jiang. Tpu-mlir: A compiler for tpu using mlir, 2023.

[13] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. *Proceedings of Machine Learning and Systems*, 1:27–39, 2019.

[14] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[15] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[16] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[17] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[18] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[19] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International conference on machine learning*, pages 2430–2439. PMLR, 2017.

[20] Adrian Osterwind, Julian Droste-Rehling, Manoj-Rohit Vemparala, and Domenik Helms. Hardware execution time prediction for neural network layers. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 582–593. Springer, 2022.

[21] TVM project. *Halide IR, https://github.com/dmlc/HalideIR*, 2023.

[22] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *International Conference on Learning Representations*, 2017.

[23] Zhe Tao, Stephanie Nawas, Jacqueline Mitchell, and Aditya V. Thakur. Architecture-preserving provable repair of deep neural networks. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[24] TensorFlow. *HLO IR, https://github.com/tensorflow/mlir-hlo*, 2023.

[25] Chuan-Chi Wang, Ying-Chiao Liao, Ming-Chang Kao, Wen-Yew Liang, and Shih-Hao Hung. Perfnet: Platform-aware performance modeling for deep neural networks. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, pages 90–95, 2020.

[26] Matthias Wess, Matvey Ivanov, Christoph Unger, Anvesh Nookala, Alexander Wendt, and Axel Jantsch. Annette: Accurate neural network execution time estimation with stacked models. *IEEE Access*, 9:3545–3556, 2020.

[27] Kun Wu, Mert Hidayetoğlu, Xiang Song, Sitao Huang, Da Zheng, Israt Nisa, and Wen-mei Hwu. Hector: An efficient programming and compilation framework for implementing relational graph neural networks in gpu architectures. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 528–544, 2024.

Acknowledgements

Any palpable progress with this work wouldn't have been possible without Prof. R. P. Gohil's guidance on a very fundamental level: we needed to base our goals on a concrete feasibility analysis rather than quixotic assumptions of glory. We really did face quite a few contingencies even after narrowing our focus down to something realistic, but, thanks to his prudent reminders, we now have some measurable progress.