

THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Neural Architecture Search as Program Transformation Exploration

Citation for published version:
Turner, J, Crowley, EJ & O'Boyle, MFP 2021, Neural Architecture Search as Program Transformation Exploration. in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021).* ACM Association for Computing Machinery, pp. 915–927, 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual, United States, 19/04/21.
https://doi.org/10.1145/3445814.3446753

Digital Object Identifier (DOI):
10.1145/3445814.3446753

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.



Download date: 21. Mar. 2026

Neural Architecture Search as Program Transformation Exploration

Jack Turner
jack.turner@ed.ac.uk
University of Edinburgh
United Kingdom

Elliot J. Crowley
elliot.j.crowley@ed.ac.uk
University of Edinburgh
United Kingdom

Michael F.P. O'Boyle
mob@inf.ed.ac.uk
University of Edinburgh
United Kingdom

arXiv:2102.06599v1 [cs.LG] 12 Feb 2021

ABSTRACT

Improving the performance of deep neural networks (DNNs) is important to both the compiler and neural architecture search (NAS) communities. Compilers apply program transformations in order to exploit hardware parallelism and memory hierarchy. However, legality concerns mean they fail to exploit the natural robustness of neural networks. In contrast, NAS techniques mutate networks by operations such as the grouping or bottlenecking of convolutions, exploiting the resilience of DNNs. In this work, we express such neural architecture operations as program transformations whose legality depends on a notion of representational capacity. This allows them to be combined with existing transformations into a unified optimization framework. This unification allows us to express existing NAS operations as combinations of simpler transformations. Crucially, it allows us to generate and explore new tensor convolutions. We prototyped the combined framework in TVM and were able to find optimizations across different DNNs, that significantly reduce inference time - over 3× in the majority of cases. Furthermore, our scheme dramatically reduces NAS search time. Code is available at this https url.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Compilers**.

KEYWORDS

program transformations, neural networks

ACM Reference Format:
Jack Turner, Elliot J. Crowley, and Michael F.P. O'Boyle. 2021. Neural Architecture Search as Program Transformation Exploration. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3445814.3446753

1 INTRODUCTION

Deep neural networks (DNNs) [39, 60] are everywhere, and there is a growing need to implement them efficiently [8, 12, 35]. This has led to an explosion in research from application [31] to hardware [9].

Currently, there are two distinct communities optimizing DNNs for commodity devices. In one, neural architecture search (NAS) researchers explore different network models trading off size and accuracy. In the other, compiler developers take the resulting networks and explore optimzations to deliver hardware performance. We argue that this two-stage deployment process is artificial and can be unified. This paper shows that program and neural architecture transformations can be *interleaved* delivering significant performance improvement and greater expressivity. Before we describe our approach, let us briefly look at the two existing communities.

1.1 Program Transformations

Within compiler research, there has been much focus on restructuring underlying tensor computations [15, 55, 80]. This involves significant loop nest restructuring [1, 10] exploiting characteristics of the target device e.g. vectorization in SIMD units or memory coalescing in GPUs [18].

There are currently many methods [10, 57, 64, 68] in use. The polyhedral model [72, 83], in particular, is a natural fit for many standard neural network operations. A key aspect of a good program transformation representation is that sequences of transformations can be easily composed and checked for legality. This allows automatic exploration of a large space of options which can then be evaluated, and the best selected [64].

1.2 Neural Architecture Search

The neural architecture search (NAS) community is also concerned with accuracy, space, and performance [19, 76]. While there have been advances in NAS for language models [63], large scale studies of NAS tend to focus primarily on convolutional architectures [17, 81, 82]. At the neural architecture level, many techniques have been proposed for the automatic generation of networks under strict budgets [30, 65, 77, 79]. These methods focus on overall network structure, selectively replacing components such as convolutional layers with computationally cheaper methods to balance the trade-off between accuracy and inference time.

One key strength of NAS is that it leverages the robustness of neural networks to deformation, reshaping and transforming them while incurring minimal damage to their ability to learn. This relies on networks maintaining their ability to extract feature representations for a given type of input, which we refer to as *representational capacity*. If an approximation or compression to a network does not inhibit its ability to learn from data, then it has not damaged the representational capacity of the network.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '21, April 19–23, 2021, Virtual, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446753

The ability of neural networks to weather compression without losing representational capacity is well-documented [16, 28, 40]. The legality of such neural architecture transformations, however, is not guaranteed and must be evaluated separately through either a training process or a small proxy task. Furthermore, while powerful, few NAS techniques explicitly take into account actual hardware behavior, and those that do face one of three problems. The first is that they only offer a black box solution with highly complex methods for predicating the search process on specific budgets (usually involving reinforcement learning [30, 65]). The second is that they do so while leaving the compilation pipeline fixed, often dismissing powerful candidate architectures because of an inappropriate, fixed choice of program transformations. The final problem is that they are limited to selecting from a pre-designed list of convolutional alternatives; they cannot synthesize their own.

1.3 Our Approach

We have two distinct communities who have the same goal but are siloed. NAS researchers assume the compiler is a black box bundled with the hardware, while compiler writers assume that the network architecture is set in stone. NAS designers can discover good networks but are limited to pre-defined options; compiler writers can efficiently exploit hardware structure but miss larger scale optimization opportunities.

What we want is the best of both worlds. We wish to combine neural architecture and compiler optimization in a unified framework. In this paper, we recast neural architecture search as program transformation exploration. By including transformations such as grouping and bottlenecking into the compiler optimization space, we leverage both the extensive history of program transformation research, and also discover new forms of neural architecture reduction that would not have been available to us otherwise.

Program transformations are necessarily restricted as they must be safe. Our solution is to unlock the space of *neural transformations* by introducing a new safety metric based on Fisher Potential [69]. It is a compile-time, cheap-to-compute metric that can reject damaging network changes, *eliminating the need to train while searching*. We, therefore, judge a transformation to be legal, not by data dependence preservation, but by the preservation of *representational capacity* (see section 5.2). This unification allows the exploration of a space that leads to more efficient implementations. It also shows that neural architecture options that previously required the engineering efforts of experts to develop, such as spatial bottlenecking, can be expressed as compositions of more fundamental transformations and discovered automatically (see section 5.3).

Our contributions are as follows:

(1) We reformulate popular Neural Architecture Search techniques as *program transformations*.
(2) We use Fisher Potential to provide transformation safety guarantees without the need to train.
(3) We unify the transformation and architecture search spaces, discovering new types of convolution.
(4) We evaluate 3 networks using these operations, ResNet, ResNext and DenseNet, on 4 platforms and demonstrate, in most cases, more than 3× inference speedup over a TVM baseline.

2 OVERVIEW

Here we develop a simple example to illustrate the direct connection between models and code, and hence, neural architecture search (NAS) and program transformation. This is followed by an outline of our approach.

2.1 Models and Code

The fundamental building block of a DNN is the tensor convolution, a generalization of a basic convolution.

Basic Convolution. Consider the first row in Figure 1. This shows a basic convolution where each element in the 2D output matrix is the weighted sum of the corresponding and neighboring elements of the 2D input matrix. The weights are stored in a small 2D filter or weight matrix W whose size corresponds to the size of the neighborhood. This is shown both diagrammatically on the left and as code on the right.

Tensor Convolution. A tensor convolution is a generalization of the basic convolution as shown in row 2 of Figure 1. The input is now a 3D tensor with a new dimension, C_i, referred to as the input channels. Similarly, the output is expanded by the number of output channels (C_o). The weight matrix is now a 4D tensor of $C_o \times C_i$ channels with two spatial dimensions (height, width). Two new loops scan the channels, the outermost, C_i shown by an arrow, while each of the tensors arrays have appropriate extra dimensions.

2.2 Models and Code Transformations

In the compiler setting, transformations change structure while maintaining meaning.

Code Transformation. Loop interchange changes the order of computation and memory access without affecting the values computed. In row 2 of Figure 1, interchanging the outer loop iterators gives the code in row 3 with the C_o and C_i loops interchanged. We can represent this using polyhedral notation $[C_i, C_o] \mapsto [C_o, C_i]$ or TVM-like syntax: `.interchange(CI,CO)`. This is shown diagrammatically with the arrow denoting the new outermost, C_o loop order. From a neural architecture perspective nothing has changed; the number of computations and memory accesses remains the same, as do the values of the output matrix. From a code perspective, the memory access behavior is significantly altered, affecting execution time.

Model Transformation. A popular choice in NAS for reducing convolutional complexity is bottlenecking [65]. For a convolution with C_o filters, we choose a bottlenecking factor B. This reduces the size of the weight and output tensors and reduces the range of the *outermost loop iterator* by a factor B. Again this can be represented as $[C_o] \mapsto [C'_o < C_o/B]$ or `.bottleneck(B)`. This is shown in row 4 of Figure 1. From a NAS point of view, this is a standard operation reducing complexity with a minimal effect on representational capacity. From a program transformation point of view, this is illegal as the computed values are changed.

2.3 Combined Space

If we consider swapping a full convolution for a reduced convolution as a program transformation, we can combine them. For

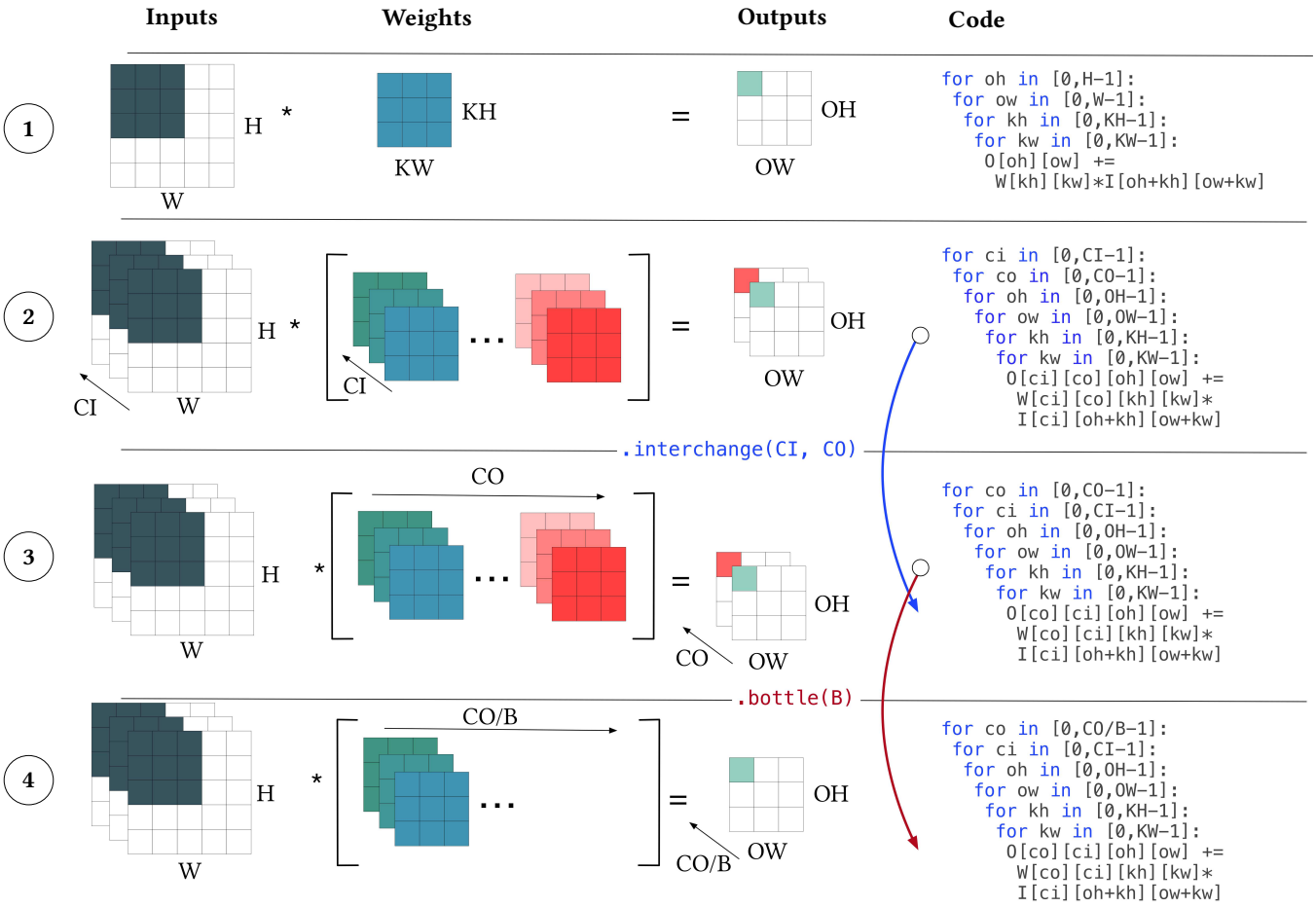


Figure 1: An illustration of the relation between neural architecture components, code and transformations. The first row shows a basic convolution. The second shows a tensor convolution. In the third row, we see a program transformation: loop interchange. The fourth row shows a neural architecture transformation: bottlenecking. Here, the number of weights is divided by B. The accuracy of the network after training does not change, but the number of operations being performed is drastically reduced.

example, we could apply loop interchange to the code in Figure 1 row 4, ($[C_o' < C_o/B, C_i] \mapsto [C_i, C_o' < C_o/B]$). With C_i now the outermost iterator, we can reapply bottlenecking, giving input channel bottlenecking. Such an optimization is both semantically invalid, and also unavailable in existing neural architecture search spaces. However, given the robustness of neural networks to noise, in some specific cases it may be just as representationally preserving as output channel bottlenecking. It is only through the lens of program transformations over loop nests that we are able to automatically access such operators.

3 NEURAL ARCHITECTURE SEARCH (NAS)

In this section we briefly describe NAS, which seeks to automate the design of neural networks for given tasks and budgets. For more details, please see [76].

Starting from an overall network skeleton, NAS attempts to design one or more cells that slot into different locations within the skeleton. Cells are described as a DAG with nodes as intermediate feature maps (or tensors) and edges representing possible operations (for example, convolution or tensor product). The task is then to find the best DAG (or cell) that can be slotted into the skeleton and trained on a given dataset. Recent work takes cells as predefined options and attempts to select where best to place each of the cells in the skeleton [69, 77]. to match specific constraints.

3.1 Neural Architectures Components

The vast majority of neural architectures consist of sequences of inter-connected components. The convolution operation dominates computational complexity and is the primary component of interest in NAS [17, 81]. We now introduce the convolution operation, and the variants that are considered as possible substitutions in our NAS baseline.

Standard Convolution. Figure 1 row 2 shows the standard convolution operation, in which an input volume of size $H_i \times W_i \times C_i$ is convolved with a set of C_o filters of size $K_h \times K_w \times C_i$, each producing an individual feature map in the output. The convolution operation is

$$\forall c_o, w_o, h_o \quad O(c_o, w_o, h_o) = \sum_{c_i}^{C_i} \sum_{k_h}^{K_h} \sum_{k_w}^{K_w} I(c_i, w_o + k_w, h_o + k_h) * K(c_o, c_i, k_w, k_h). \tag{1}$$

Bottlenecked Convolution. A popular choice for reducing convolutional complexity is bottlenecking [29] as shown in Figure 1 row 4. A bottlenecking factor B is selected, reducing the number of weights/filters to C_o/B. ResNets [29] frequently feature bottleneck blocks that consist of trios of convolutions, one to bring the number of channels down, another for processing, and a final one to bring the number of channels back up. A similar pattern is relied on in several NAS techniques [65, 66]. The bottlenecked convolution operation is

$$\forall c'_o < \frac{C_o}{B}, w_o, h_o \quad O(c'_o, w_o, h_o) = \sum_{c_i}^{C_i} \sum_{k_h}^{K_h} \sum_{k_w}^{K_w} I(c_i, w_o + k_w, h_o + k_h) * K(c'_o, c_i, k_w, k_h). \tag{2}$$

Grouped Convolution. Here, the C_i channel input is split along the channel dimension into G groups, each of which has C_i/G channels. Each group is independently convolved with its input split, producing C_o/G channels which are concatenated along the channel dimension. Let $O = [O'_1; \ldots, O'_G]$ i.e. each slice $O'_g, g \in 1, \ldots G$ is concatanated to form O. I'_g is similarly defined. Group convolution is then as follows:

$$\forall c'_o, w_o, h_o \quad O(c'_o, w_o, h_o) = \sum_{c'_i}^{C'_i} \sum_{k_h}^{K_h} \sum_{k_w}^{K_w} I(c'_i, w_o + k_w, h_o + k_h) * K(c'_o, c'_i, k_w, k_h). \tag{3}$$

This reduces the number of basic convolutions from $C_o \times C_i$ to $G \times C_o/G \times C_i/G = (C_o \times C_i)/G$ reducing the number of operations used by a factor of G. Note that we can think of standard convolution as grouped convolution with $G = 1$.

Depthwise Convolution. Notice that if the number of groups is equal to the number of input channels and the number of output channels, $G = C_i = C_o$, then there is a single, 2D convolutional filter for each input channel C_i. This is known as depthwise convolution and is a special case of group convolution.

3.2 NAS Search Space Example

Typically, the NAS heuristic finds the best cell to fill in the skeleton by choosing appropriate operations. Figure 2 gives an example cell from [17]. The skeleton is ResNet-like with 5 cells in series. Each cell contains 4 inter-connected nodes (A,B,C,D). Between nodes, downsampling takes place where the spatial dimensions are halved in size while doubling the channel depth. This restricts the types of operations available on each edge. This gives a total of 15625 possible cells, which captures most of the available options within cell-based NAS techniques, each of which the authors train exhaustively on various datasets.

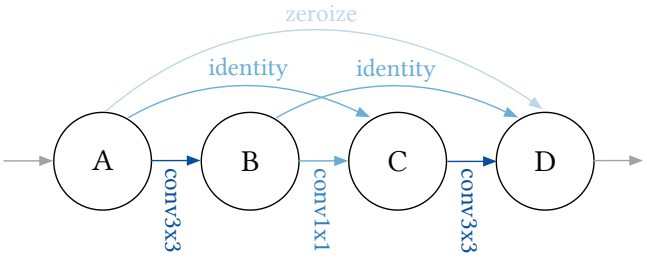


Figure 2: An illustration of a Neural Architecture Search design space (as used in [17]). Every cell has exactly four nodes (A,B,C,D) representing intermediate feature map states. Edges represent operations that transform intermediate states from source to target node, taken from a list of options specified by the designer.

Rather than designing cells by choosing which of a predetermined list of options is best, we instead wish to choose a sequence of transformations from an original model which will allow us to step through the cell design space. We describe the program transformations we use in the next section.

4 PROGRAM TRANSFORMATIONS

Due to the restricted, static, convex and affine nature of tensor convolutions, it is natural to describe program transformations of them in the well-studied polyhedral model. In this section we give a brief introduction to the model; for more a detailed description, we refer the reader to [72, 75]. The polyhedral model of a program consists of three main components:

The **domain** is a collection of the possible statement instances that occur within the iteration space of a set of loop bounds. We can represent each statement instance with a multidimensional co-ordinate corresponding to the iterator values of the loops that surround it. As the constraints on the loops are affine, the bounded domain forms a convex polyhedron.

A **set of accesses** are affine mappings of the iteration space to memory. Two statement instances have a dependence ordering between them if they have accesses to the same memory location and at least one of them is a write.

A **schedule** assigns a timestamp to each statement instance, dictating the order in which they are executed. Different schedules for the same program represent possible program transformations. The transformed schedules are determined to be semantically preserving if dependence ordering is preserved.

To illustrate this, consider the implementation of the 1×1 convolution at the start of a residual block listed in Algorithm 1. We can describe the domain as follows:

$$\begin{aligned} \texttt{S1}(c_o, h, w) \quad & | \, 0 \leq c_o < C_o \wedge 0 \leq h < H \wedge \\ & \quad 0 \leq w < W \\ \texttt{S2}(c_o, h, w, c_i) \quad & | \, 0 \leq c_o < C_o \wedge 0 \leq h < H \wedge \\ & \quad 0 \leq w < W \wedge 0 \leq c_i < C_i \end{aligned}$$

Algorithm 1 Naive implementation of 1×1 tensor convolution.

```
for (co=0; co<Co; co++)
  for (oh=0; oh<OH; oh++)
    for (ow=0; w<OW; ow++)
S1    O[c_o][h][w] = 0.;
      for (ci=0; ci<Ci; ci++)
S2      O[co][oh][ow] +=
          W[co][1][1] *
          I[ci][oh][ow];
```

We can also describe the schedule as follows:

$$T_{S1}(c_o, h, w) = (c_o, h, w)$$
$$T_{S2}(c_o, h, w, c_i) = (c_o, h, w, c_i)$$

We now briefly discuss the transformations we consider.

Loop Interchange. Interchanging two nested loops involves applying a permutation to the schedule of the enclosed statements. For example, in the algorithm shown in 1 and a statement $S1$ we can express this as simply as:

$$T_{S1}(c_o, h, w) = (c_o, w, h)$$

Strip-Mining. This is performed by mapping an iterator into two new iterators whose combined range is that of the original. A constant strip-mining factor is selected which forms the range of the new inner loop. The outer loop range is that of the original divided by the strip-mine factor. For example, to strip mine the inner c_i loop of Algorithm 1 we have

$$T_{S2}(c_o, h, w, c_i) = (c_o, w, h, c_i/32, c_i \pmod{32})$$

Tiling. Tiling is a combined transformation consisting of strip-mining followed by interchange. For example, to tile c_i loop of 1 we have

$$T_{S2}(c_o, h, w, c_i) = (c_i/32, c_o, w, h, c_i \pmod{32})$$

There are many more transformations that can be expressed in the polyhedral model. For further detail, we point the reader to the excellent polyhedral literature [27, 71].

4.1 Legality

Legality in the polyhedral framework is determined by preservation of data dependences. If there is a data dependence between two dynamic statement instances under the original program schedule, the relative ordering between these statements must be preserved under the new transformed schedule.

In the case of Algorithm 1, we can see that $S1$ is the definition, or source, of a dependence that must occur before its usage (in $S2$). For an instance i of statement $S1$ and j of statement $S2$, assuming constant data dependences, we can represent dependences in matrix form D. Elements of D are non-negative integers such that an element $d_{S1,S2}$ indicates that i must execute before j (formally, $i \prec j$, where $\prec$ denotes lexicographical ordering).

Given a linear transformed schedule of the iteration space I in matrix form T and a dependence matrix D, then a transformation is legal iff:

$$\forall i, j, S1, S2, D \quad i \rightarrow j \in d_{S1,S2} \rightarrow T(i) \preceq T(j)$$

Algorithm 2 Grouping transformation of tensor convolution.

```
for (g=0; g<G; g++)
  for (co=Co/G*g; co<Co/G*(g+1); co++)
    for (ci=Ci/G*g; ci<Ci/G*(g+1); ci++)
      for (oh=0; oh<OH; oh++)
        for (ow=0; w<OW; ow++)
          for (kh=0; kh<KH; kh++)
            for (kw=0; kw<KW; kw++)
              O[co][oh][ow] +=
                        W[co][ci][kh][kw] *
                        I[co][oh+kh][ow+kw];
```

which is to say that the lexicographical ordering is preserved.

5 UNIFIED SPACE

Our approach to joining NAS and compiler optimizations is to describe convolutional alternatives as polyhedral program transformations. Due to the affine nature of convolutions, this is relatively straightforward, except for checking transformation legality.

We adapt a newly developed legality check based on Fisher information [69], called Fisher Potential, that allows us to check the legality of transformations without having to retrain the network each time. This leads to a dramatic reduction of transformation search time as shown in Section 7.

5.1 Extending the Polyhedral Model

Bottlenecking. The bottleneck transformation is a reduction in the outer iterator in the domain node by parameterized constant factor B. Let the vector $\underline{J}$ denote the iterators spanning the domain of the tensor loop nest, $\underline{J} = [c_o, c_i, h, w, k_h, k_w]$ where c_o is the outermost iterator and $\underline{J'} = [c_i, h, w, k_h, k_w]$ be the enclosed inner iterators, then bottlenecking can be expressed as

$$T_S(c_o, J') = (c'_o, J') \mid c'_o < C_o/B$$

The value B is constrained such that $C_o \pmod{B} \equiv 0$. This gives a factor B reduction in computation. Figure 1 row 4 shows an example of bottlenecking.

Grouping. Grouping can be thought of as tiling the two outer iterators by a common factor and then discarding one of the iterators. The original iterator domains must both be divisible by the grouping factor G. Let $J' = [c_i, J'']$, such that $J = [c_o, c_i, J'']$ then we can define grouping as

$$T_S(c_o, c_i, J'') = (g, c_o/G, c_i/G, J')$$

to give the algorithm in Figure 2. Note as C_o, C_i and G are compile-time known constants each of the loop bounds is affine. Examining the code we see that each slice g of the output array refers only to the corresponding slice of the weight and input array.

Depthwise. Depthwise convolutions can be considered as a special case of group convolutions where the group size equals the number of output channels, $C_o, G = C_o$. For this transformation to be possible, the number of input and output channels must be equal, $C_o = C_i$. This means the two inner loops will have strip

Algorithm 3 Depthwise transformation of tensor convolution.

```
for (g=0; g<Co; g++)
  for (oh=0; oh<OH; oh++)
    for (ow=0; w<OW; ow++)
      for (kh=0; kh<KH; kh++)
        for (kw=0; kw<KW; kw++)
          O[g][oh][ow] +=
                    W[g][kh][kw] *
                    I[g][oh+kh][ow+kw];
```

counts of 1 as $C_o/G = C_i/G = 1$ and

$$T_S(c_o, c_i, J'') = (g, 1, 1, J')$$

which can be trivially simplified to

$$T_S(c_o, c_i, J'') = (g, J')$$

5.2 Fisher Potential as a Legality Check

The transformations described above fail to preserve traditional program semantics. We instead state that a schedule transformation for a neural network with respect to a task is legal if the final classification accuracy on the held out data is the same, or similar to within a small δ. Training all possible transformed networks to determine accuracy, however, would be prohibitively expensive.

To address this, we employ *Fisher Potential* [69] as a pre-training legality check. It is a cheap-to-compute measure that is effective at rejecting any architectures whose final test accuracy is significantly below the original starting point, *without needing to perform training*. Informally, Fisher Potential is, locally, the total information that each loop nest (layer) contains about class labels under a simplifying assumption of conditional independence. We note that the specific measure could easily be swapped out for another, such as [46], or any of the measures developed in [2], and that this is an area of active development.

Fisher Potential refers to the use of aggregated Fisher Information at initialization to estimate the effectiveness of network architectures before training. Where [40] used the diagonal of the Hessian in order to compress networks, [50, 67] showed that this computation could be approximated via the Fisher Information Matrix. This approximation was empirically shown to be effective at estimating the importance of individual neurons in a neural network [49], and to correlate to final accuracy [26]. In SNIP [41], the authors used a measure heavily related to Fisher Information *at initialization* in order to prune connections in the network prior to the training process. This was adapted in [69] to perform architecture search, by summing the Fisher Information over neurons to get layer-wise importance estimates. It is this form of the measure that we use in this paper.

Formal Definition. For a single channel, c, of a convolutional filter in a network, consider that for some input minibatch of N examples, its outputs are an $N \times W \times H$ tensor where W and H are the channel's spatial width and height. We refer to this tensor as the activation A of this channel and denote the entry corresponding to example n in the mini-batch at location (i, j) as A_{nij}. As the network has a loss function $\mathcal{L}$, then we can get the gradient of the loss with respect to this activation channel $\frac{\partial \mathcal{L}}{\partial A}$. Let us denote this gradient as g and index it as g_{nij}. The channel c error, Δ_c can then be computed by

$$\Delta_c = \frac{1}{2N} \sum_{n}^{N} \left(- \sum_{i}^{W} \sum_{j}^{H} A_{nij} g_{nij} \right)^2 . \tag{4}$$

This gives us a filter-wise score for a particular channel. In order to gauge the sensitivity of the full convolution, we sum Δ_c over each channel (as in [69]):

$$\Delta l = \sum_{c_o}^{C_o} \Delta_{c_o} \tag{5}$$

This score is summed for each of the convolutional blocks in the network. For an original network and a proposed alternative architecture, we reject the proposal if its score is below that of the original.

To summarise, the Fisher Potential of a proposed network architecture is the sum of the Fisher Information for each layer when given a single random minibatch of training data, as performed in [69].

Example. Let us consider, candidate networks designed from NAS-Bench-201 [17] as shown in Figure 3. Each point represents a different neural architecture of which there are 15625. The y-axis shows final CIFAR-10 top-1 error (lower is better), and the x-axis shows the Fisher Potential assigned to the model *at initialization* (higher is better). We can see that without requiring any training, Fisher Potential is able to filter out poorly-performing architectures, visible in the cluster of low scoring networks on the left with poor final errors. Many good networks are also discarded — this is unfortunate but acceptable for our scenario, since the space is large and densely populated with networks. We note again that this measure in particular could be swapped out for an improved one in future.

5.3 Expressive Power

Neural Architecture Search techniques rely on hand-designed exploration spaces. themselves are engineered by hand. For example, a recent paper found that bottlenecking could be applied in the spatial domain [53]. This can now be added to a list of candidate operations. However, spatial bottlenecking is automatically captured within our framework as the combination of existing transformations. As an example, consider the convolution in row 2 of Figure 1. We use the shorthand notation $[i] \xrightarrow{B(b)} [i(b)]$ to denote bottlenecking domain i by factor b and $[i, j] \xrightarrow{\text{int}} [j, i]$ to denote interchange. Then the spatial bottleneck operation can be constructed as the following sequence of transformations:

$$\begin{aligned} T_S : [C_o, C_i, H, W, K_h, K_w] &\xrightarrow{\text{int.}} \\ [H, W, C_o, C_i, K_h, K_w] &\xrightarrow{\text{B(b).}} \\ [H(b), W, C_o, C_i, K_h, K_w] &\xrightarrow{\text{int.}} \\ [W, H(b), C_o, C_i, K_h, K_w] &\xrightarrow{\text{B(b).}} \\ [W(b), H(b), C_o, C_i, K_h, K_w] &\xrightarrow{\text{int.}} \\ [C_o, C_i, H(b), W(b), K_h, K_w] & \end{aligned}$$

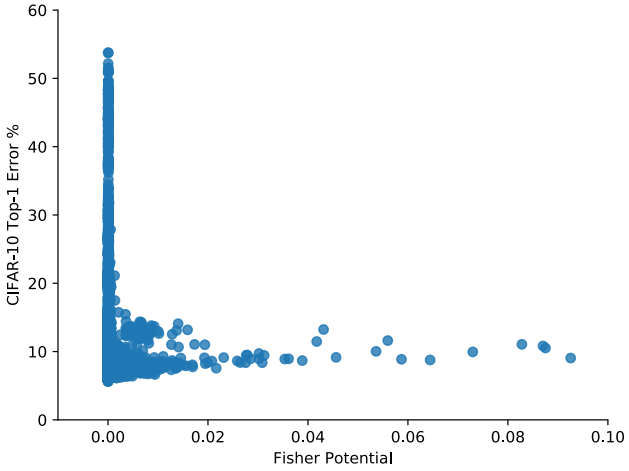


Figure 3: Fisher Potential as a rejection filter for invalid architectures. Each point is a different neural architecture from NAS-Bench-201 [17]. The y-axis shows final CIFAR-10 top-1 error (lower is better), and the x-axis shows the Fisher Potential assigned to the model *at initialization* (higher is better).

The expressiveness of the polyhedral model, equipped with these new transformations, means that there are novel transformations that can be derived from this unified framework.

6 IMPLEMENTATION AND SETUP

We expressed our new transformations as polyhedral transformations and implemented the resulting operators in TVM [10]. We selected TVM as its API allows composition of transformation sequences, and is a well-accepted state-of-the-art optimizing compiler for convolutional neural networks.

Transformation Space. An overview of the transformations used in this paper is given in Table 1. The first four are standard program transformations available within TVM. To these, we add our two neural architecture transformations. Finally, for GPU platforms, we enable GPU mapping transformations.

Baseline TVM. TVM allows users to implement schedules for each new operator by hand. Due to the large number of operators involved, for each device we use TVM's default schedules; automatic schedule design [84] has yet to be incorporated into TVM. We then enable auto-tuning of parameter values within the schedule to find best performance.

Search. Our current search process is relatively naive. We enumerate random sequences of transformations through TVM, and generate 1000 configurations of the resulting operations for each network. We then check which candidates pass our implementation of the Fisher Potential legality test and select the best performing one.

Table 1: A description of the autotuning primitives available to TVM for optimizing operations, including GPU mapping, standard primitives, and our two new optimizations.

Optimization	Description
Program Transformations	
`reorder`	Interchange nested loops
`tile`	Cache and register blocking
`unroll`	Loop unrolling
`prefetch`	Memory coalescing between threads
`split`	Divide iteration into multiple axes
`fuse`	Combine two axes into one
Neural Architecture Transformations	
`bottleneck`	Reduce domain by factor B
`group`	Slice and offset two loops by factor G
Mapping to GPU	
`blockIdx`	Block-wise parallelism
`threadIdx`	Threads within blocks
`vthread`	Striding thread access

6.1 Experimental Setup

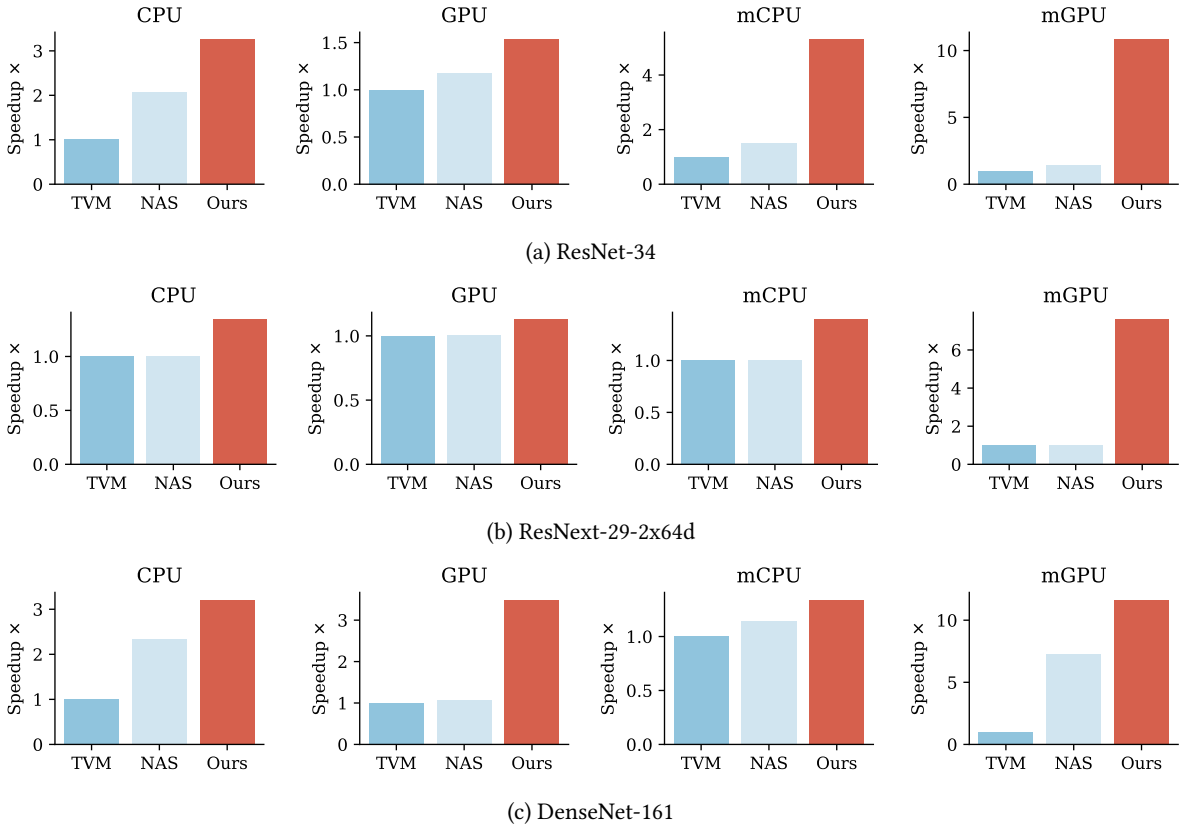
Platforms. We evaluate on an ARM A57 mobile CPU (mCPU), and an Nvidia 128-Core Maxwell mobile GPU (mGPU) available on the Jetson Nano board. We also evaluate an Intel Core i7 (CPU) and an Nvidia 1080Ti (GPU). This is representative of a wide range of deployment targets, from mobile to server class. We use TVM `v1.7` compiled with CUDA 10.1 and LLVM 8.0. For training and accuracy evaluation we use PyTorch `v1.4`.

Neural Models. We evaluate our methodology on three popular networks: ResNet-34 [29], ResNeXt-29 [78], and DenseNet-161 [34]. These networks were chosen to represent a range of convolutional architectures, from standard 3×3 convolutions in ResNet-34 to grouped convolutions in ResNeXt and a heavy reliance on 1×1 convolutions in DenseNet. CIFAR-10 models are trained for 200 epochs with Stochastic Gradient Descent (SGD) [6] and a learning rate of 0.1, decayed by a factor of 10 at epochs 60, 120, and 160. ImageNet models were trained using the default PyTorch script[1], which trains for 90 epochs with SGD, starting from a learning rate of 0.1 and decaying by a factor of 10 every 30 epochs.

Comparison. The models are implemented with each operation written as a TVM Tensor Expression [10], which is an einsum-style syntax for expressing tensor computations. This is lowered to TVM IR, where our transformations can be employed. This allows for a fair comparison of each approach.

For each network, we consider three approaches. First we compile the model using TVM using its default schedule (labeled TVM). We report the best performance achieved after auto-tuning. Next, we use BlockSwap [69] as NAS to compress the modifiable convolutions in the network, followed by compilation with TVM (labeled NAS). Finally, we apply our unified approach as described in the previous section (labeled Ours).

[1] https://github.com/pytorch/examples/tree/master/imagenet



(a) ResNet-34

(b) ResNext-29-2x64d

(c) DenseNet-161

Figure 4: End-to-end performance for several networks on different hardware devices on CIFAR-10. TVM represents the original model compiled with TVM default schedules. The NAS columns represent the BlockSwap-compressed copies of the models which are then compiled with TVM default schedules. Ours represents our unified NAS-compilation strategy, with new operators stacked into the network blocks.

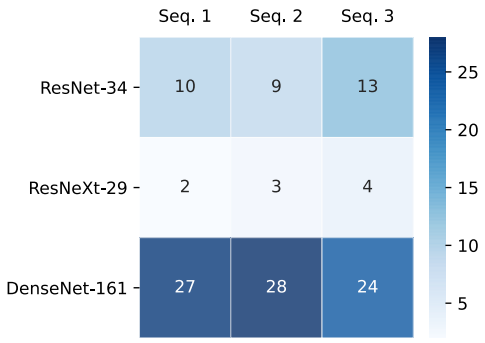


Figure 5: Frequency of operation application.

7 RESULTS

In this section we first present the performance of the three approaches on different networks and platforms for the CIFAR-10 dataset. This is followed by an analysis of accuracy, size, search time and the highest performing new convolutions found. We then drill down into one network and examine the impact of transformations layer-by-layer. In order to compare against alternative NAS techniques, we evaluate the performance of FBnet across the three networks. Next, we evaluate the performance of our approach on the ImageNet dataset additionally examining the tradeoff in accuracy and performance across variants of the ResNet and DenseNet family of networks. Finally we show that our approach allows for fine-grained exploration of an accuracy/size tradeoff, producing a new Pareto optimal network design.

7.1 CIFAR-10 Network Performance

The results for each network are shown in Figure 4. The combination of TVM and NAS is a strong baseline, however, for each of the models our unified method is able to generate models with improved hardware performance. In general there is more performance to gain on the mGPU than other platforms, as relaxed memory pressure from smaller designs is of increasing importance.

ResNet-34. NAS is able to find a modest improvement of 1.12× speedup over TVM on the GPU platform, increasing to 2× on i7 CPU, showing the power of compressed convolutions. Our unified approach is able to find further improvement giving 2× and 3×

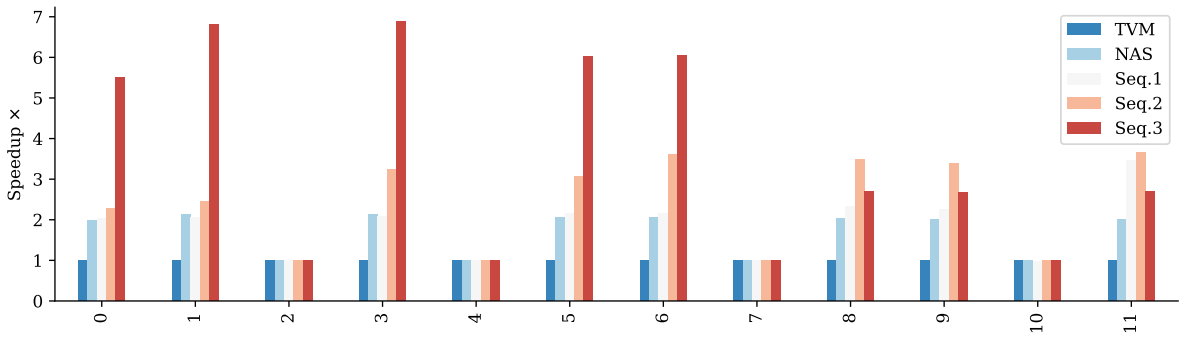


Figure 6: Exploring different sequences of transformations for an individual layer of ResNet-34 on the Intel Core i7 CPU. `NAS` is the result of applying grouping with factor 2 first, then compiling with TVM. The other three sequences are interleaved transformations produced by our method.

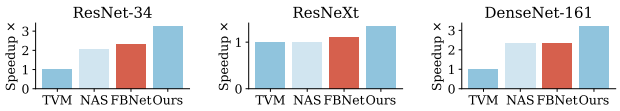


Figure 7: Intel Core i7 performance of FBNet on the three networks. FBNet is able to improve over NAS with large training cost. Our approach outperforms FBNet with no training required

speedup respectively. The improvement is more dramatic on the smaller platforms with 5× and 10× speedups on the mCPU and mGPU.

ResNext-29-2x64d. NAS is unable to find any improvement here due to the already highly compact structure of the network. There are simply no NAS options that improve performance over the TVM baseline across all platforms. Despite this our approach is able to find small improvements of 1.3× and 1.1× on the CPU and GPU platforms by combining NAS and program transformations. This increases to 1.4× on the embedded CPU and 7× on the mGPU.

DenseNet-161. The impact of NAS is much more varied here. While it can find 2.2× improvement on the CPU, it has a negligible improvement over TVM on the GPU. It also struggles on the mCPU but finds 6× on the mGPU. Our approach is able to improve by over 3× for both server platforms, but only finds 1.2× on the mCPU while acheiving 10× on the mGPU.

7.2 Analysis

Accuracy For all CIFAR-10 networks pictured in Figure 4, changes in accuracy were less than 1% in absolute difference. The ResNet-34 in Figure 6 had an original ImageNet Top-1 and Top-5 accuracy of 73.2% and 91.4% respectively. The final network compiled with our transformations had a Top-1 and Top-5 accuracy of 73.4% and 91.4%; it was slightly more accurate than the original but considerably faster. To give a broader overview, we present the accuracy of several ImageNet models in Figure 8, which shows that for each model accuracy degradation is small or non-existent.

Size: One benefit of these neural transformations is their effect on model size, both in terms of weights and runtime memory usage. We found that CIFAR-10 networks could be compressed in size by $2-3\times$. Likewise, the ImageNet ResNet-34 in Figure 6 was compressed from 22M parameters to 9M without a loss in accuracy.

Search time: Our search process involves suggesting configurations and rejecting or accepting them based on Fisher Potential. Since Fisher Potential is extremely cheap to compute, the search time overhead introduced by our method is small, less than 5 minutes on a CPU. During this time we were able to explore 1000 different configurations, discarding approximately 90% of the candidate transformation sequences through the Fisher Potential legality check.

7.3 Transformation Sequence Case Studies

There were 3 particular transformation sequences that dominated the list of best performing transformations. In neural architecture terms, the resulting operations of each of these sequences of transformations is a new convolution-like operator, previously unavailable to NAS.

`Sequence 1` applies grouping to the kernels over the spatial domain of the input. These are then concatenated to form one output. The exact sequence is: `[split → interchange → group → interchange → fuse]`.

`Sequence 2` is an operator in which the output channels have been unrolled by factor 16 and then the remaining iteration domain has been grouped by factor $G = 2$. The exact sequence is: `[unroll → group → interchange]` `Sequence 3` is an operator that was devised by splitting up the iteration domain of the output channels, and applying different levels of channel grouping to each new domain (e.g. $G = 2$ on first half, $G = 4$ on the second half). The exact sequence is: `[split → group → interchange → group]`.

Figure 5 shows how often each of these sequences appear in our best performing networks. ResNext-29 has the fewest instances as it contains the fewest layers, while DenseNet-161 has the most layers and hence most instances. Each of them is applicable across all networks and may be more widely applicable as standard tensor convolutions for other networks.

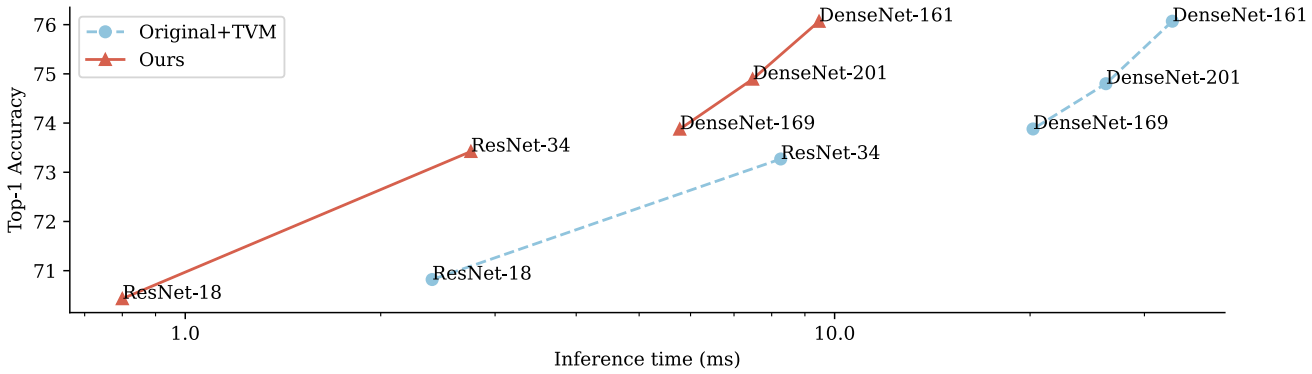


Figure 8: Accuracy vs inference time (log scale) of our approach (Ours) compared to TVM (Original + TVM) when applied to different variants of ResNet and DenseNet on the ImageNet dataset. Our approach gives a significant reduction in inference time.

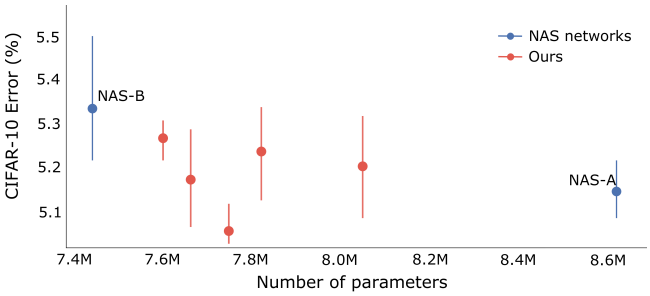


Figure 9: Two NAS models (the blue points labeled NAS-A and NAS-B) composed of grouped blocks (with g=2 and g=4 respectively) can be chained together by a series of parametrized transformations in our framework, yielding the points in red. Each point is the mean of three training runs, with error bars.

7.4 Exploring Layer-Wise Optimizations

In Figure 6, we examine the impact of our sequences layer-by-layer on one network on one platform: ResNet-34 on the i7. The exact configurations mirror the experiment in the original TVM paper [10]. While we achieve a 3x improvement overall as shown in Figure 4, the speedup achieved varies considerably across layers. In 4 of the 11 layers no performance improvement is found, as Fisher Potential marks these individual layers to be extremely sensitive to compression. Simple grouping with a factor $G = 2$ is able to give around 2× speedup across 7 of the 11 layers. Sequence 1 is able to give a small improvement in most cases — particularly layer 11 — where spatial reduction yields new parallelization opportunities. Sequence 2 provides a slight improvement in many cases, and through the later layers it is able to offer larger speedups through improved data reuse. Sequence 3 is the best option for most early layers but suffers compared to other sequences in the later layers.

7.5 Alternative NAS comparison

To provide an alternative evaluation against an existing NAS technique, we re-implement FBNet [77] using the convolutional blocks available in our NAS space, and our three baseline networks as the skeletons into which the selected cells are inserted. Figure 7, shows the performance of resulting networks on CIFAR-10 relative to TVM, NAS and our approach on the Intel Core i7 for each network. In each case, FBNet does provide a modest improvement over NAS. However, it is worth noting that this involves an expensive training step at each stage of evaluation (requiring ~3 GPU days per network). Our approach is able to consistently improve over FBNet, with no training required.

7.6 ImageNet Network Performance

We also applied our technique to the ImageNet classification dataset, to show that the method transfers beyond CIFAR-10. In Figure 8, we first show the resultant networks for running our method on ResNet-18, ResNet-34, DenseNet-161, DenseNet-201, and DenseNet-169, with their inference times recorded on the Intel i7 CPU. For each network, accuracy is within 2%, and there are significant gains in inference time.

7.7 Interpolating Between Models

The additional expressive power of our method allows us to explore the search space of networks in a more fine-grained manner than traditional NAS techniques. This is because NAS techniques simply choose operations from a list, whereas we generate new ones. To illustrate this consider Figure 9 where we plot the accuracy and inference time of a ResNet-34 with two BlockSwap-based models (the blue points labeled NAS-A and NAS-B). We can explore alternatives by a series of parametrized transformations in our framework, yielding the points in red. Each of these points is a new possible blocktype that would not be accessible to a traditional NAS technique unless explicitly written by the human designer. During this interpolation we are also able to find a Pareto optimal point.

8 RELATED WORK

Compiler optimization. There has been much interest in autotuning DNN code generators [10, 23, 37, 48, 64, 72]. Polyhedral compilers are particularly well-suited [72, 83] as they have in-built abstractions for exploiting parallelism and memory layout in a principled form. Polyhedral compilation itself benefits from a wealth of literature [5, 13, 24, 38, 71, 75] that describes a unified compilation scheme and associated set of legality checks delivering state-of-the-art performance across a wide range of standard compiler benchmarks [27, 73, 74], as well as image and neural network specific ones [51, 72, 83].

Other popular approaches include loop synthesis [10] and rewrite rules [64] which optimize through parametrized schedules. In particular, TVM [10] is an adaptation of Halide [57] with specific extensions for deep neural network abstractions. Triton [68] is a tile-based IR for parallelizing tensor-based computation. Most of these frameworks focus on operator-level optimization, though higher level graph transformation has also shown promising results [37].

Though the toolchains are exceptionally feature-rich, there is evidence that some implementations have been highly-engineered for specific workloads, at the cost of general support for newer optimizations [4]. All current approaches, however, are limited by their inability to exploit NAS transformations.

Reducing the amount of computation at the cost of accuracy has been examined in a compiler context using loop perforation [20, 59, 62] in approximate computing. Within approximate computing, there is extensive work on probabilistic program transformations [47, 58, 85] and language support for approximation [25, 52]. Where some approximate computing methods accept small absolute differences in the outputs of programs, our transformations may render the numerical outputs of our programs completely different while maintaining the legality of the program. The benefit of Fisher Potential, while it does not give any guarantees on behavior, is that its domain specificity allows us to capture these transformations.

Model Optimization. From a machine-learning perspective, deep neural networks should be highly compressible as there is significant evidence that they are vastly overparametrized [16, 21]. There are several popular strategies that can be employed. One can prune a network by removing unimportant connections between weights [22, 28, 50]. However, the sparsity introduced can lead to poor memory behavior [14]. An alternative is to retain dense weight tensors by pruning channels [42] although the resulting irregular structures cause a significant slowdown [56].

Network distillation [3, 31, 70], takes a trained large network and uses its outputs to assist in the training of a smaller network. This, however, leads to a large design space as there are many ways to make a network smaller [11, 32, 33, 36, 78].

Neural architecture search instead automates the process of finding efficient networks. In [86] the authors use an RNN to generate network descriptions and filter the options using reinforcement learning. Several networks are generated and trained in parallel, which requires a large quantity of possible networks to be stored at once. To reduce this overhead, an alternative method is to design one *supernet* that contains all of the possible subnetworks [54]. This allows all models to have access to a shared set of weights, reducing the computational requirement. Subsequent works have made extensive use of this technique [7, 44, 45, 77].

However, there is some evidence to suggest that weight sharing schemes aggressively hamper the ability of the NAS agent to identify good architectures, and under such constrained spaces, random architecture search provides a competitive baseline [43, 61]. Although NAS is a rapidly developing area, no-one to the best of our knowledge has considered formalizing NAS as program transformation.

9 CONCLUSIONS AND FUTURE WORK

This paper presents a new unified program transformation approach to optimizing convolution neural networks by expressing neural model operations as formal program transformations. We develop a new legality check based on Fisher potential, and show that different types of convolution can be described as transformations of more fundamental building blocks. We implemented this approach in TVM and show that our combined approach significantly outperforms both TVM and state-of-the-art NAS. We eliminate the need to train while searching, dramatically reducing search time. Future work could consider further transformations in the neural architecture space in order for a more extensive analysis, and to expand the possibilities beyond that of just convolutional networks.

ACKNOWLEDGMENTS

The material is partially based on research by Michael O'Boyle which was sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The views and conclusions contained herein are those of the authors and do not represent the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This work was partially supported by the Engineering and Physical Sciences Research Council (grant EP/L0150), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*.

[2] Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D Lane. 2021. Zero-Cost Proxies for Lightweight NAS. *arXiv preprint arXiv:2101.08134* (2021).

[3] Lei Jimmy Ba and Rich Caruana. 2014. Do Deep Nets Really Need to be Deep?. In *Advances in Neural Information Processing Systems*.

[4] Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 177–183.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[6] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 177–186.

[7] Liang-Chieh Chen, Maxwell Collins, Yukun Zhu, George Papandreou, Barret Zoph, Florian Schroff, Hartwig Adam, and Jon Shlens. 2018. Searching for efficient multi-scale architectures for dense image prediction. In *Advances in Neural Information Processing Systems*.

[8] Tianshi Chen, Yunji Chen, Marc Duranton, Qi Guo, Atif Hashmi, Mikko Lipasti, Andrew Nere, Shi Qiu, Michele Sebag, and Olivier Temam. 2012. BenchNN: On

the broad potential application scope of hardware neural network accelerators. In *International Symposium on Workload Characterization*.

[9] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation*.

[11] François Chollet. 2017. Xception: Deep Learning with Depthwise Separable Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[12] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. 2019. Visual Wake Words Dataset. *arXiv preprint arXiv:1906.05721* (2019).

[13] Albert Henri Cohen. 1999. *Analyse et transformation de programmes: du modèle polyédrique aux langages formels*. Ph.D. Dissertation. Versailles-St Quentin en Yvelines.

[14] Elliot J Crowley, Jack Turner, Amos Storkey, and Michael O'Boyle. 2018. Pruning neural networks: is it time to nip it in the bud? *arXiv preprint arXiv:1810.04622* (2018).

[15] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *arXiv preprint arXiv:1801.08058* (2018).

[16] Misha Denil, Babak Shakibi, Laurent Dinh, Ranzato Marc'Aurelio, and Nando de Freitas. 2013. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*.

[17] Xuanyi Dong and Yi Yang. 2020. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HJxyZkBKDr

[18] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs. In *In International Workshop on Machine Learning and Programming Languages*.

[19] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. 2019. Neural Architecture Search.

[20] Michael Figurnov, Aijan Ibraimova, Dmitry Vetrov, and Pushmeet Kohli. 2015. Perforatedcnns: Acceleration through elimination of redundant convolutions. *arXiv preprint arXiv:1504.08362* (2015).

[21] Jonathan Frankle and Michael Carbin. 2019. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*.

[22] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The State of Sparsity in Deep Neural Networks. *arXiv preprint arXiv:1902.09574* (2019).

[23] Perry Gibson, José Cano, Jack Turner, Elliot J Crowley, Michael O'Boyle, and Amos Storkey. 2020. Optimizing Grouped Convolutions on Edge Devices. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 189–196.

[24] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming* 34, 3 (01 June 2006), 261–317. https://doi.org/10.1007/s10766-006-0012-3

[25] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. 2015. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 383–397.

[26] Aditya Sharad Golatkar, Alessandro Achille, and Stefano Soatto. 2019. Time matters in regularizing deep networks: Weight decay and data augmentation affect early learning dynamics, matter little near convergence. In *Advances in Neural Information Processing Systems*. 10678–10688.

[27] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *International Workshop on Polyhedral Compilation Techniques*.

[28] Song Han, Huizi Mao, and William J. Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations*.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[30] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[31] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[32] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[33] Gao Huang, Schichen Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2018. CondenseNet: An Efficient DenseNet using Learned Group Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[34] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[35] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*.

[36] Yani Ioannou, Duncan Robertson, Roberto Cipolla, and Antonio Criminisi. 2017. Deep Roots: Improving CNN Efficiency with Hierarchical Filter Groups. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[37] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Symposium on Operating Systems Principles*.

[38] Wayne Kelly and William Pugh. 1998. *A framework for unifying reordering transformations*. Technical Report.

[39] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436.

[40] Yann LeCun, John S. Denker, and Sara A. Solla. 1989. Optimal Brain Damage. In *Advances in Neural Information Processing Systems*.

[41] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip H. S. Torr. 2019. SNIP: Single-shot network pruning based on connection sensitivity. In *International Conference on Learning Representations*.

[42] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning filters for efficient convnets. In *International Conference on Learning Representations*.

[43] Liam Li and Ameet Talwalkar. 2019. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638* (2019).

[44] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations*.

[45] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. 2018. Neural architecture optimization. In *Advances in Neural Information Processing Systems*.

[46] Joseph Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley. 2020. Neural Architecture Search without Training. *arXiv preprint arXiv:2006.04647* (2020).

[47] Sasa Misailovic, Daniel M Roy, and Martin C Rinard. 2011. Probabilistically accurate program transformations. In *International Static Analysis Symposium*. Springer, 316–333.

[48] Naums Mogers, Valentin Radu, Lu Li, Jack Turner, Michael O'Boyle, and Christophe Dubach. 2020. Automatic generation of specialized direct convolutions for mobile GPUs. In *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 41–50.

[49] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. 2019. Importance estimation for neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 11264–11272.

[50] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2017. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *International Conference on Learning Representations*.

[51] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. Polymage: Automatic optimization for image processing pipelines. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 429–443.

[52] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. 2015. Flexjava: Language support for safe and modular approximate programming. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 745–757.

[53] Junran Peng, Lingxi Xie, Zhaoxiang Zhang, Tieniu Tan, and Jingdong Wang. 2018. Accelerating Deep Neural Networks with Spatial Bottleneck Modules. *arXiv preprint arXiv:1809.02601* (2018).

[54] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. 2018. Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning*.

[55] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. Archit. Code Optim.* 14, 3, Article 26 (Aug. 2017), 25 pages. https://doi.org/10.1145/3107953

[56] Valentin Radu, Kuba Kaszyk, Yuan Wen, Jack Turner, José Cano, Elliot J Crowley, Björn Franke, Amos Storkey, and Michael O'Boyle. 2019. Performance aware convolutional neural network channel pruning for embedded GPUs. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 24–34.

[57] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Acm Sigplan Notices*, Vol. 48. ACM, 519–530.

[58] Martin Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*. 324–334.

[59] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01* 1, 2 (2015).

[60] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.

[61] Christian Sciuto, Kaicheng Yu, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. 2019. Evaluating the Search Phase of Neural Architecture Search. *arXiv preprint arXiv:1902.08142* (2019).

[62] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 124–134.

[63] David R So, Chen Liang, and Quoc V Le. 2019. The evolved transformer. *arXiv preprint arXiv:1901.11117* (2019).

[64] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *International Symposium on Code Generation and Optimization*.

[65] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[66] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *International Conference on Machine Learning*.

[67] Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. 2018. Faster gaze prediction with dense networks and Fisher pruning.

[68] Philippe Tillet, HT Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *International Workshop on Machine Learning and Programming Languages*.

[69] Jack Turner, Elliot J. Crowley, Michael O'Boyle, Amos Storkey, and Gavin Gray. 2020. BlockSwap: Fisher-guided Block Substitution for Network Compression on a Budget. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SklkDkSFPB

[70] Jack Turner, Elliot J Crowley, Valentin Radu, José Cano, Amos Storkey, and Michael O'Boyle. 2018. Distilling with performance enhanced students. *arXiv preprint arXiv:1810.10460* (2018).

[71] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. 2006. Polyhedral code generation in the real world. In *International Conference on Compiler Construction*.

[72] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[73] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*.

[74] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 54.

[75] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. In *International Workshop on Polyhedral Compilation Techniques*.

[76] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. 2019. A survey on neural architecture search. *arXiv preprint arXiv:1905.01392* (2019).

[77] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. FBNet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[78] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[79] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. 2018. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[80] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. 2016. A Systematic Approach to Blocking Convolutional Neural Networks. *arXiv preprint arXiv:1606.04209* (2016).

[81] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. 2019. Nas-bench-101: Towards reproducible neural architecture search. *arXiv preprint arXiv:1902.09635* (2019).

[82] Arber Zela, Julien Siems, and Frank Hutter. 2020. NAS-Bench-1Shot1: Benchmarking and Dissecting One-shot Neural Architecture Search. *arXiv preprint arXiv:2001.10422* (2020).

[83] Tim Zerrell and Jeremy Bruestle. 2019. Stripe: Tensor Compilation via the Nested Polyhedral Model. *arXiv preprint arXiv:1903.06498* (2019).

[84] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. *arXiv preprint arXiv:2006.06762* (2020).

[85] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A Kelner, and Martin Rinard. 2012. Randomized accuracy-aware program transformations for efficient approximate computations. *ACM SIGPLAN Notices* 47, 1 (2012), 441–454.

[86] Barret Zoph and Quoc V. Le. 2017. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*.