DISSERTATION

submitted to the

Combined Faculty of Mathematics, Engineering and Natural Sciences

of the

Ruprecht–Karls University
Heidelberg

for the degree of

Doctor of Natural Sciences

put forward by

M.Sc. Dennis Sebastian Rieber

born in
Balingen, Baden-Württemberg

Heidelberg, 2022

Deployment of Deep Neural Networks on Dedicated Hardware Accelerators

Advisor: Professor Dr. Holger Fröning

Oral Examination: .....

I dedicate this dissertation to my loving wife

Tina Rieber

Without her, I could have never walked this long and winding path.

Abstract

Deep Neural Networks (DNNs) have established themselves as powerful tools for a wide range of complex tasks, for example computer vision or natural language processing. DNNs are notoriously demanding on compute resources and as a result, dedicated hardware accelerators for all use cases are developed. Different accelerators provide solutions from hyper scaling cloud environments for the training of DNNs to inference devices in embedded systems. They implement intrinsics for complex operations directly in hardware. A common example are intrinsics for matrix multiplication. However, there exists a gap between the ecosystems of applications for deep learning practitioners and hardware accelerators. How DNNs can efficiently utilize the specialized hardware intrinsics is still mainly defined by human hardware and software experts.

Methods to automatically utilize hardware intrinsics in DNN operators are a subject of active research. Existing literature often works with transformation-driven approaches, which aim to establish a sequence of program rewrites and data-layout transformations such that the hardware intrinsic can be used to compute the operator. However, the complexity this of task has not yet been explored, especially for less frequently used operators like *Capsule Routing*. And not only the implementation of DNN operators with intrinsics is challenging, also their optimization on the target device is difficult. Hardware-in-the-loop tools are often used for this problem. They use latency measurements of implementations candidates to find the fastest one. However, specialized accelerators can have memory and programming limitations, so that not every arithmetically correct implementation is a valid program for the accelerator. These invalid implementations can lead to unnecessary long the optimization time.

This work investigates the complexity of transformation-driven processes to automatically embed hardware intrinsics into DNN operators. It is explored with a custom, graph-based intermediate representation (IR). While operators like *Fully Connected Layers* can be handled with reasonable effort, increasing operator complexity or advanced data-layout transformation can lead to scaling

issues. Building on these insights, this work proposes a novel method to embed hardware intrinsics into DNN operators. It is based on a dataflow analysis. The *dataflow embedding* method allows the exploration of how intrinsics and operators match without explicit transformations. From the results it can derive the data layout and program structure necessary to compute the operator with the intrinsic. A prototype implementation for a dedicated hardware accelerator demonstrates state-of-the art performance for a wide range of convolutions, while being agnostic to the data layout. For some operators in the benchmark, the presented method can also generate alternative implementation strategies to improve hardware utilization, resulting in a geo-mean speed-up of $\times 2.813$ while reducing the memory footprint. Lastly, by curating the initial set of possible implementations for the hardware-in-the-loop optimization, the median time-to-solution is reduced by a factor of $\times 2.40$. At the same time, the possibility to have prolonged searches due a bad initial set of implementations is reduced, improving the optimization's robustness by $\times 2.35$.

Zusammenfassung

Tiefe Neuronale Netzwerke (Deep Neural Networks, DNNs) haben sich als mächtiges Werkzeug für eine Vielzahl an komplexen Aufgaben, wie Objekterkennung oder Sprachverarbeitung etabliert. Ihr Bedarf an Rechenressourcen ist enorm, weshalb dedizierte Beschleuniger für alle Einsatzzwecke entwickelt werden, von ultrahoch skalierenden Rechenzentren bis zu eingebetteten Systemen. Die Beschleuniger implementieren Instruktionen für komplexe Operationen direkt in der Hardware. Ein verbreitetes Beispiel sind Instruktionen für Matrixmultiplikationen. Aber es gibt eine Lücke zwischen dem Ökosystemen für die Anwender von Deep Learning und Hardwarebeschleunigern. Die effektive Nutzung von spezialisierter Hardware wird immer noch von Hardware- und Softwareexperten definiert.

Methoden, um solche Instruktionen in die Berechnung von DNNs einzubetten werden aktiv erforscht. Bisherige Veröffentlichungen setzen oft auf Transformationsgetriebene Ansätze, mit dem Ziel eine Transformationssequenz zu finden, die es ermöglicht einen Teil der DNN Berechnung mit der spezialisierten Instruktion zu ersetzen. Allerdings wurde die Komplexität dieser Aufgabe noch nicht erforscht, insbesondere für weniger übliche DNN Operatoren, wie zum Beispiel *Capsule Routing*. Nicht nur das Implementieren von DNN Operatoren mit spezialisierten Instruktionen ist eine Herausforderung, auch deren Optimierung auf der Zielhardware ist schwierig. Systeme, die mit Hardwarefeedback arbeiten werden hier oft genutzt. Die Latenz von verschiedenen Implementierungskandidaten wird gemessen, um den schnellsten Kandidaten zu finden. Aber da Hardwarebeschleuniger oft Programmierungs- und Speicherlimitierungen haben kann nicht jede arithmetisch korrekte Implementierung auch auf dem Beschleuniger genutzt werden. Diese ungültigen Implementierungen können den Optimierungsprozess unnötig Verlängern.

Diese Arbeit untersucht die Komplexität von Transformationsgetriebenen Prozessen, um spezialisierte Instruktionen in DNN Operationen einzubetten. Dabei wird eine eigens Implementierte Zwischendarstellung (Intermediate Rep-

resentation, IR) verwendet, welche auf Graphen basiert. Während sich Operationen wie *Fully Connected Layer* mit annehmabarem Aufwand verarbeiten lassen, führen komplexere Operatoren und Transformationen zu Skalierungsproblemen. Basierend auf diesen Erkenntnissen wird eine neue Methode zum Einbetten von spezialisierten Instruktionen vorstellt. Diese basiert auf Dataflussgraphen. Die *Dataflow Embedding* Methode erlaubt das Erforschen von Abbildungen zwischen DNN Operatoren und Hardware ohne explizite Transformationen. Aus den Ergebnissen können die Transformationen für die Datenanordnung und die Programmstruktur abgleitet werden. Eine Prototypenimplementierung für einen dedizierten Hardwarebeschleuniger kann alternative Implementierungsstrategien erzeugen, welche die Auslastung der Hardware verbessern. Diese sind um einen Faktor 2.813 schneller, während sie weniger Speicher belegen. Zu guter Letzt wird durch das sorgfältige Auswählen von Implementierungen für den Ausgangspunkt der Optimierung mit Hardwarefeedback die mittlere Zeit zur Lösungsfindung um den Faktor 2.40 reduziert. Zur selben Zeit wird die Wahrscheinlichkeit eines langen Optimierungsprozesses wegen eines schlechten Ausganspunkts reduziert, was die Robustheit der Optimierung um den Faktor 2.35 erhöht.

Table of contents

1 Introduction **1**

2 Background **7**
2.1 Deep Neural Networks 7
2.2 Hardware for Deep Learning 10
2.3 Software for DNNs 13
2.4 Related Work 19

3 Complexity Analysis of Iteration-Domain Embedding **25**
3.1 Tensor Compute Graphs 27
3.2 Intrinsic Embedding 33
3.3 Transformations 37
3.4 Evaluation 42
3.5 Discussion 49

4 Joint Program and Data-Layout Transformation through Dataflow Embedding **55**
4.1 Solving the Embedding Problem on the Dataflow Level 56
4.2 Polyhedral Representation of Dataflow Graphs 60
4.3 Embedding as a Constraint Satisfaction Problem 65
4.4 Experiment Setup 76
4.5 Evaluation 85
4.6 Search Robustness and Branching Strategy Optimization 94
4.7 Discussion 97

5 Hardware-Aware Initialization of Auto-Tuning on Hardware Accelerators **103**
5.1 Auto-Tuning Space Analysis 104
5.2 Validity Driven Model Initialization 114
5.3 Evaluation 118

5.4 Discussion 127

6 Discussion and Outlook **133**

6.1 Top-Down and Bottom-Up Embedding 133

6.2 Outlook 136

7 Conclusion **141**

References **145**

Notational Conventions

In this work we will use the following notational conventions:

- Multidimensional arrays, also called tensors, are denoted as boldface lower case alphabetical symbols and comma separated subscripts for each dimension, e.g. $\mathbf{x}_{\mathbf{i},\mathbf{j}}$. Note that vectors are considered tensors of dimensionality 1.
- Sets are denoted by calligraphic upper case alphabetical symbols, such $\mathcal{A}$, $\mathcal{B}$, $\mathcal{S}$. Subscripts and superscripts are sused to discern between different sets, such as $\mathcal{S}_1$ and $\mathcal{S}_2$ or $\mathcal{A}^{in}$ and $\mathcal{A}^{out}$.
- The $\%$ sign denotes a *remainder* division, also known as *modulo* operation
- $\lfloor\ \rfloor$ and $\lceil\ \rceil$ denote rounding down and up to the closest integer, respectively.

Chapter 1

Introduction

Deep Neural Networks (DNNs) established themselves as a key technology in the Machine Learning (ML) community for tasks like computer vision or language processing. From medical, over industrial to automotive applications, DNNs are used for tasks like object classification, image segmentation, physics simulation, path projection or hazard detection. The ability to extract complex, non-linear functions from the training data while maintaining the ability to generalize is a main driver of their success. A downside is their enormous computational demand for both training and inference. While the initial training can be scaled to thousands of machines, deployment is constrained by energy, time and financial factors for all application domains, ranging from embedded, automotive, industrial or mobile devices to hyper-scaling could environments.

This sparked the development of specialized hardware accelerators for DNN workloads. By offering complex hardware intrinsics, performing hundreds or thousands of arithmetic operations, both sequentially and in parallel, they are more energy and time efficient than general purpose hardware. Translating a DNN, built from operators like convolutions or pooling, down to executable instructions of an accelerator can be complex and multiple deployment routes are possible, as shown in Figure 1.1. The following paragraphs will explain the different concepts in detail.

The central problem for the deployment of DNN operators on hardware accelerators is to find *embeddings*. An embedding is the replacement of a full or partial DNN operator's computation with a hardware intrinsic. This is possible if the replaced part and hardware intrinsic are computationally equivalent. To find an embedding means to find such equivalent parts. To do this, it might be

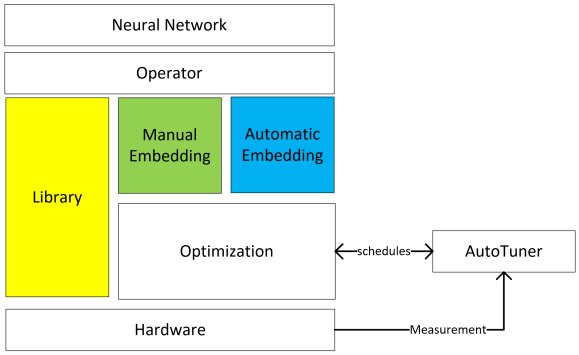


Figure 1.1 Possible deployment routes for DNNs.

necessary to transform the operator's data layout, data access function and order of computations to find the computation matching an intrinsic.

Target specific *libraries* directly implement the operators of a neural network, specifically, to improve throughput, latency or energy consumption through efficient usage of the hardware's memory hierarchy and processing elements. While this achieves best-in-class performance, the engineering effort for new operators or hardware architectures is high. Another downside is potentially poor hardware utilization for specific operator parametrizations. Prime examples for this problem are convolutions and matrix multiplications, which appear in many shapes and sizes, making library design challenging.

The second route is *manual embedding*. Recently, modular approaches are researched, where the operator is described by a set of a basic linear algebra functions like *dot-product* or functions like *sigmoid*. Individual functions are then optimized either manually, with *auto-tuning* tools or realized directly in hardware as intrinsics. This allows portable implementations of each operator and reduces the effort of supporting new hardware targets, since only the intrinsics need to be provided. However, describing how to realize a DNN operator with the available functions is still done manually. This manual implementation process limits the exploration of new operators. And like libraries, a fixed strategy of how the intrinsics are used in the operators can result in low hardware utilization. Also, in scenarios of hardware-software co-design, the mapping of high-level operators to intrinsics can change depending on the hardware configuration.

The last option is the *automatic embedding*. Automatically realizing the DNN operators as a sequence of accelerator intrinsics is challenging. Code generation for CPUs or GPUs gradually lowers application logic down to fine-granular machine instructions, executing scalar or vectorized arithmetic and memory instructions. In contrast, DNN accelerators perform more complex computations

like matrix multiplication in a single step, often fused with data parallel activation functions on the results. To utilize accelerators, the DNN operator needs to be expressed as a sequence of one or more of these complex hardware intrinsics. Methods used in compilers for general purpose hardware are not designed to handle the comparatively coarse intrinsics of DNN accelerators.

Most existing solutions to solve this *automatic embedding problem* automatically rely on abstract program representations and exploration of possible implementations through transformations. In such representations, loops, sum operations and multi-dimensional arrays are first-class members, on which transformations perform specific manipulations. For example, the ordering of loops or manipulation of an array's shape can be changed. This is combined with pattern matching algorithms to determine if an embedding is possible. In some tools, only some parts of this are automated, while other parts have to be determined by a human expert. The efficiency of this approach is determined by the set of available transformations and how they are applied. An embedding can fail if a necessary transformation is not available, or a specific sequence of rewrites not applied. As a result, the transformation and pattern matching process requires careful tailoring towards both, hardware intrinsics and operators.

Unlike libraries, after the embedding the *schedule* of each operator needs to be optimized. A schedule describes the specific order of load, store and compute instructions for a given workload and has a significant impact on how effective the hardware resources are utilized. This is challenging, especially for commutative workloads like matrix multiplications or convolutions, with millions of potential schedules for the same workload. As exhaustively trying out all solutions is not feasible, *auto-tuning* became a popular approach to solve this *scheduling problem*. Aided by different methods, like genetic algorithms or statistical models, promising schedule candidates for evaluation on hardware are proposed. After a fixed number of iterations, the auto-tuning stops and selects the best-performing implementations from the set of candidates executed in hardware. As hardware accelerators often have less flexible programming models, the schedules need not only provide good performance, but also need to be valid in terms of code generation. This validity issue adds a second dimension to the scheduling problem.

This work studies the deployment of DNNs on dedicated inference accelerators, specifically *automatic embedding* and *scheduling*. Existing solutions to bring DNNs to such accelerators require engineering effort, often on the level of individual operators. Paired with long-running optimizations, this can lead

application experts to resort to tried and true solutions like libraries on general purpose hardware, instead of exploring novel ideas. The motivation is to enable applications experts and researchers to use a wider range of hardware accelerators and novel operators without the overhead of expensive implementations. First, this work evaluates existing methodologies to automate accelerator code generation, which often focus on transformation-driven processes. Then, *dataflow embedding* is presented. It is a novel method to find embeddings, which builds on the insights from the analysis of transformation-driven processes. It works on dataflow graphs and produces detailed mappings between operator and intrinsics on a scalar operation level. From such mappings, the data layout, data access functions and computation order transformations that enable an embedding can be inferred. It is evaluated on the *Versatile Tensor Accelerator* (VTA), a custom hardware accelerator and the results match and even outperform the existing deployment solution. Finally, the scheduling problem is investigated, with special attention to the needs of accelerators, where issues with the robustness and time to solution are reported. Therefore, an improved auto-tuning initialization method is proposed which outperforms existing random and machine-learning based initialization.

Contributions

1. *Complexity analysis of iteration-domain embedding:* Transformation-driven instruction embedding on the iteration-domain level is so far the most widespread method attempting to solve the embedding problem. By designing and implementing a lightweight, graph-based intermediate representation (IR) that lends itself well to pattern matching and transformation, the advantages and limits of this *Top-Down* approach are explored. The results indicate limitations of this approach with respect to scalability for complex workloads. This contribution was published: [76].

2. *Dataflow Embedding — Methods and Constraints:* Motivated by the insights from the Top-Down approach, a method based on solving the problem on a lower level, without transformations is created. Based on the polyhedral program representation and constraint programming it enables embedding scalar dataflows. The results are then used to determine the necessary data layout and program transformations. For this purpose, several new constraints are developed and implemented.

A prototypical implementation for the VTA architecture has been implemented. Evaluation on the DeepBench Benchmark Suite shows equivalent performance to handcrafted strategies. Significant improvements are realized for convolution variations and edge cases, for which no efficient implementation strategy was specified. A geo-mean speed-up of $\times 2.813$ was achieved, while simultaneously having a smaller memory footprint. Portability is demonstrated on a modified configuration of the VTA accelerator, outperforming the manual embedding by $\times 4.99$ on the tested operators. Method and results were published: [75].

3. *Hardware Aware Auto-Tuning Initialization:* While the embedding process is one challenge, auto-tuning for specialized hardware also contains significant challenges. Especially the sparsity of the optimization space, with many invalid schedules, creates robustness issues in the tuning process. On the example of VTA an auto-tuning tool called AutoTVM, the problem is analyzed and a solution is presented. With a hardware aware initialization of the tuning, robustness and optimization time respectively improve by factors of $\times 2.35$ and $\times 2.40$ over existing solutions. Method and results were published: [77].

Organization of this Work

Chapter 2 provides context and basic information about the subjects of this work: deep learning workloads, their description and training as well as an overview of available hardware system and optimization methods. Then, a deeper dive into existing work for deploying DNNs on dedicated hardware is provided, giving insights to the research gap motivating this work. Chapter 3 looks at the general complexity of the embedding problem, with the help of a graph-based IR, designed for lightweight program transformation and pattern matching. The results, based on a set of deep-learning operators, demonstrate a difficulty cliff in the current state of the art and motivates new approaches. Chapter 4 proposes a new concept to solve the embedding problem. It introduces embedding on the dataflow level, enabling exploration of computation and data access patterns without transformations. This is evaluated on the VTA hardware, demonstrating significant performance improvements for specific convolution parametrizations and portability is evaluated on different VTA configurations. Chapter 5 dives deeper into the automatic optimization of DNN operators on accelerators, where

robustness issues are observed. With simple and hardware-aware initialization methods for the auto-tuning process, these issues can be mitigated, improving both optimization time and robustness. Chapter 6 discusses the benefits and issues of the methods presented in Chapters 3 and 4 before an outlook on possible future research directions is given. Lastly, Chapter 7 summarizes and concludes this work.

CHAPTER 2

Background

The purpose of this chapter is to set this work into context with Deep Neural Networks and their hardware and software ecosystem. This field of research is almost impossible to represent exhaustively. Therefore, the following chapter focusses on the works deemed most influential to the field or this work. While all presented concepts are important and represent their own research field, emphasis is on the inference optimization for dedicated accelerators, as it is the focus of this dissertation.

2.1 Deep Neural Networks

Deep Neural Networks (DNNs) are constructed from linear transformation and non-linear activation operators, arranged in sequential and parallel order. They operate on *tensors*, multi-dimensional arrays of data. An individual operator is denoted as

$$\mathbf{x}^l = \phi(\mathbf{w}^l \oplus \mathbf{x}^{l-1} + b^l) \tag{2.1}$$

where $\mathbf{x}^l$ is the computed activation of layer l, ϕ is the selected non-linearity and $\mathbf{w}^l$ the parameter tensor that is linearly combined with input $\mathbf{x}^{l-1}$ through linear operation $\oplus$. To this, the bias vector $\mathbf{b}^l$ is added. The most common activations functions in DNN are the *rectified linear unit* (ReLU) and variations of it, as well as the *sigmoid* and *tanh* functions.

Training of DNNs aims to adjust the values of weights in all layers so that the network solves a specific task. The most common method for this is

supervised learning on labelled training data, where a loss function is computing the difference between the correct solution and the network's result for a given input. Backpropagation [82] computes the gradient for each input-target pair, minimizing the loss function in the process. The most common method for is *Stochastic Gradient Descent* [78]. Several hyper-parameters, like *learning rate* or *regularization*, govern the training process, for example to prevent overfitting.

2.1.1 Convolutions

The dominant linear operators in modern neural networks are *convolutions* and variations of the *dense* operator. Convolutions are used on spatial data, like two-dimensional images with multiple channels, e.g. for red, green, blue and transparency. The most common form is the 2D-Convolution, or *Conv2D*, defined as:

$$\mathbf{x^l}_{n,c,h,w} = \sum_{k_h}^{K_H} \sum_{k_w}^{K_W} \sum_{ic}^{IC} \mathbf{w}^l_{c,ic,k_h,k_w} \cdot \mathbf{x}^{l-1}_{n,ic,f_h(h,k_h),f_w(w,k_w)} \tag{2.2}$$

Functions $f_h()$ and $f_w()$ describe how dimensions h, w of the input feature map $\mathbf{x}^{l-1}$ are accessed:

$$f_h(h, k_h) = (h \cdot s_h) + (k_h \cdot d_h) - \lfloor \frac{K_H \cdot d_h}{2} \rfloor \tag{2.3}$$

$$f_w(w, k_w) = (w \cdot s_w) + (k_w \cdot d_w) - \lfloor \frac{K_W \cdot d_w}{2} \rfloor \tag{2.4}$$

For the input tensor indexed by n, ic, h and w, parameters $s_h, s_w \in \mathbb{Z}^+$ are constant integer strides in the respective dimensions and $d_h, d_w \in \mathbb{Z}^+$ are constant *dilation* factors, enlarging the receptive field of the convolution. In essence, a three-dimensional stencil, indexed by k_h, k_w and ic is computed for every output element. The height, width and depth of the result tensor are indexed by h, w and c. When activations are processed in batches, each individual input activation is indexed by n. The operation has many opportunities for data reuse and parallelization.

Convolutions are compute intensive and can carry up-to millions of weight parameters. Different variations of the same workload are used to reduce the computational load or number of parameters, especially for embedded use cases. Depthwise convolutions reduce the number of *multiply-accumulate* (MAC) operations by applying only a single convolutional filter $K_H \times K_W$ to each input channel. Depthwise-separable [23] convolutions then follow up with a pointwise filtering over the channels. Strided convolutions process the image dimension

with fixed strides s_h, s_w to reduce the number of MACs while simultaneously compressing the output feature maps by the stride factor in each dimension. Dilated convolutions [117] reduce the number of weights and MACs by upscaling up the stencil with dilation factor d_h, d_w. They are commonly used in image segmentation applications. Grouped convolutions [55] were introduced to enable model-parallelism across devices during training by splitting the input and output channels into multiple groups. The results are stacked together, resulting in the same number of channels as without grouping. An effect of this split is a reduction of model parameters by the factor of the split and a significant reduction of MACs. Groups can be shuffled [122], which aims to benefit the overall application performance.

Variations for application classes beyond computer vision include *Conv1D* with only a single spatial dimension w, or *Conv3D* which adds another feature map dimension d. In practice, all modification for 2D can also be applied to the 1D and 3D variations.

2.1.2 Dense Operators

The second important DNN operator class are *dense*-like operations:

$$\mathbf{y}_{d_1,\ldots,d_i,j} = \sum_{k=0}^{K} \mathbf{x}_{d_1,..,d_i,k} \cdot \mathbf{w}_{j,k}^{T} \tag{2.5}$$

Algorithmically they are less complex than convolutions, due to the strictly linear data access pattern of both, the output and the reduction over a single dimension of size K. A common variations are the *fully-connected* layer with $i = 0$, with single input and output dimensions k and j, respectively. In convolutional neural networks they are often used as the final layers in image classification tasks, like in ResNet [42] or VGG [88]. With $i = 1$, it becomes a matrix multiply used in wide variety of operators like *Attention* [104] or *Capsule Routing* [83]. Operators like batch-reduce GEMM [36] with $i > 1$ have also been proposed, as they offer significant parallelization opportunities.

2.1.3 Tensor Expressions

A shared property of the presented DNN operators, as well as other operators, is that each output element is computed by the same operations, but with a different input interval. This kind of computation will be referred to as *tensor expression*, a terminology adapted from TVM [17].

Tensor expressions describe the computation of a tensor, using its dimensions as indices for access to the input tensors. Additional indices can be introduced by reduction operations, for example sums or products. They natively translate to loop nests with static control flow, meaning there are no if-statements and the iteration domains only depend on known parameters. An example is Equation 2.6:

$$\mathbf{c}_i = \sum_k \mathbf{a}_{i,k} \cdot \mathbf{b}_k \tag{2.6}$$

Only two indices are used to access input tensors $\mathbf{a}_{i,k}$ and $\mathbf{b}_k$, namely i and k. Index i is defined by $\mathbf{c}_i$ and k by the sum operation. Every element in $\mathbf{c}_i$ is then computed as the dot product between vector $\mathbf{b}_k$ and row i of matrix $\mathbf{a}_{i,k}$. Instead of a mathematical notation, a codelike shorthand notation is also used:

```
1 c[i] = ∑_k a[i,k] · b[k]
```

and both can be described in a loop nest like this:

```
1 for i in I:
2   for k in K:
3     c[i] += a[i,k] · b[k]
```

These properties allow a compact, hierarchical compute description of a tensor, based on the number of dimensions, arithmetic operations and memory access functions. Several modern compilation frameworks for DNNs rely on this representation to express their operators [17, 101, 93].

2.2 Hardware for Deep Learning

2.2.1 General Purpose Hardware

The most widespread general purpose accelerator for deep learning and scientific workloads are *(General Purpose) Graphics Processing Units* ((GP)GPUs). With massively parallel, lightweight compute units, large memory bandwidth and structured programming model they fit well to the computational demand of DNNs. The memory hierarchy with a programmable scratchpad and extensive register files enable heavy data reuse, if possible, in the workload. The Bulk-Synchronous-Parallel [98] (BSP) processing model can hide memory latencies through efficient suspension and resuming of thread workgroups. Features for DNN processing include *Tensor Cores*, half-precision 4×4 matrix multiplication units, accessible with hardware intrinsics on the level of thread workgroups.

Support for quantized processing, such as 4bit integers[1] or 16bit floating point [2] formats were added in newer generations to further support deep learning use cases. This results in high efficiency in terms of operations per watt. However, their high structural demand in the application reduce the efficiency of sparse workloads. Utilizing the massive performance potential requires deep understanding of both the workload and the hardware architecture, which is why libraries like cuDNN [21] are a popular solution among application specialists.

General purpose CPUs are designed for low-latency, single-threaded operations. Their process- and data-parallelism is limited, compared to the tens of thousands of threads and thousands compute units in GPUs. With the comparatively short SIMD units, high hardware utilization in sparse applications is achievable. Support for 8bit integer and extreme low-bit quantization is possible due to extensive instruction sets for bit-wise operations. Especially in embedded devices, for example from ARM, these features are utilized [84]. Paired with the high processing frequency, CPUs are still viable many DNN use cases.

2.2.2 Domain Specific Accelerators

The computational demand of DNNs sparked research into dedicated hardware accelerators, especially for convolutional operators. *Conv2D's* data reuse and parallelism gives ample opportunity to design different hardware architectures. Improved throughput and energy efficiency are the main drivers for dedicated accelerators. *Eyeriss* [20] is a massively parallel accelerator with a grid of processing units and a freely programmable dataflow. A main benefit is the reduced data movement cost, as data flows from one processing unit to the next, avoiding memory accesses. A similar architecture is *DianNao* [15]. Different dataflow schemes have been proposed for these kinds of accelerators, extensively reviewing the differences between weight and activation static flows. However, among experts it is still discussed which influence hardware architecture and technology have on energy efficiency, compared to the dataflow [115].

Instead of manipulating the dataflow, other accelerators offer a fixed set of hardware intrinsics. Data is moved between the specialized processing unit and the memory subsystem, which often has one or more layers of local buffers to mitigate the memory access cost. This hardware class is sometimes called loop-back architectures. Efficiently using the intrinsics, optimizing the order of

[1] https://developer.nvidia.com/blog/int4-for-ai-inference, accessed 04.2022

[2] https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8, accessed 04.2022

computations and avoiding low utilization is crucial to achieve good performance in these systems. A popular example is Google's *Tensor Processing Unit* (TPU) [50], which has a 256×256 matrix-multiplication unit. It is implemented in a systolic array, a form of massively parallel processing grid through which the data flows in a predetermined pattern and well suited for linear algebra problems. Similarly, *VTA* [67] is a deep-learning accelerator offering load, store and processing operations for 2D matrices. Its design features a reconfigurable processing element and buffers, enabling HW/SW-Codesign for each use case. Another example for embedded devices is PULP [25], a RISC-V based accelerator. More specialized processing units implement full convolutions and even fuse them with further pooling operations [109]. By replacing the single, high-performance processing unit with smaller, lightweight compute units with SIMD intrinsics a hybrid of dataflow and loop-back is created. In this work, we regard them closer to loop-back than dataflow. For example, *VIP* [46] has an extensive scratchpad memory in each processing unit, allowing for more complex sequences of intrinsics. *Cambricon* [61] offers a similar ISA to *VIP*. What they all have in common are intrinsics to load, process and store matrices or even full tensors in a single command. This enables better efficiency compared to vector or scalar instructions, if they can be mapped to the workload. Here we want to emphasize the last sentence, as mappings between complex instructions and DNNs workloads is a topic of ongoing research, including this work.

Intrinsics for VTA, PULP, VIP, Cambricon and TPU offer matrix-matrix or matrix-vector operations. Intrinsics for full convolutions, like in [109] are used rarely. However, direct implementations for convolutions are often used when generating hardware architectures directly from DNN for Field Programmable Gate Arrays (FPGAs). The reconfigurable nature of these devices allows custom hardware for every DNN. Examples are AutoSA [110], which is a continuation of PolySA [24], using the polyhedral model to generate custom systolic arrays and the spatial and temporal schedules at the same time. Another example is Tabla [87], which generates FPGA accelerators for 2D and 3D DNNs based on the hardware templates for the Winograd convolution method. Performance optimization, especially memory accesses, use analytical models in an exhaustive search space sweep.

2.3 Software for DNNs

2.3.1 Network Description and Training

Frameworks such as TensorFlow [2], Pytorch [72] or Caffe [47] focus on application-level interfaces. They provide descriptive interface to build computational flows from parametrizable operator building blocks like convolutions, pooling, activations or dense-like operations. Tensors are first class concepts and data-layout transformation like *transpose* or *reshape* are available. A key feature is the gradient based optimization (training) of the described DNNs. From the network description and loss function specification the tools automatically compute the gradient. Network training is automated and can be scaled from single devices to large clusters.

The abstract descriptions are executable on a wide range of back-ends, from GPU clusters to personal computer CPU by mapping the operators to highly optimized libraries. Two widespread choices are cuDNN [22] for NVIDIA GPUs or NNPack [3] for x86 architectures. The specificity of libraries provides near optimal performance on specific hardware, but increases the engineering effort when exploring new operators or hardware architectures.

2.3.2 Inference Optimizations

Describing and training a DNN is the first half of its life cycle, the second part is using the DNN. This is typically called *inference*. There, DNNs are used to classify or make predictions on real-world inputs, based on their training. Efficiency in time, throughput and energy usage are important for all production environments, but especially for edge and embedded devices. Tools and methods to optimize inference performance are an active topic of research, focussing on optimizing the different stages of deployment: graph, strategy and schedule level. The main goal is lowering the general DNN description into target specific code, fully utilizing the target's memory hierarchy and computing capabilities. Choices on one level influence the available optimization paths on the levels below.

Several compilation stacks have been presented for deep learning inference and vary by a wide range of parameters. TensorComprehesions [101] is only targeting GPUs. It uses genetic algorithms to iteratively improve operator implementations,

[3] https://github.com/Maratyszcza/NNPACK, accessed 05.2022

focussing on tiling and other loop optimizations. *Graph Lowering* (GLOW) [81] has a multi-level IR, where the high-level representation performs graph-level optimizations like operator fusion. High-level constructs are defined as a series of low-level intrinsics like matrix-multiply. This way, different hardware targets only have to implement the low-level building-blocks without looking at high-level operators. However, the high-level to low-level mapping is a manual task. Similarly, *Tensor Processing Primitives* (TPP) [37] defines a set of hardware-agnostic intrinsics to compose high-level operators. The implementation of each intrinsic is then target-specific. It is embedded into the PlaidML [121] compiler. TVM [17] is also a multi-level system, with an emphasis on portability for different back-ends. Contrary to GLOW or TPP, high-level operators are implemented directly in a tensor expression programming language, while providing an optimization tool to fine-tune operators individually for every hardware target. This allows target-specific, high-level implementations of an operator.

2.3.2.1 Graph-Level Optimization

The first optimization level is the graph level, where operator fusion is one of available concepts at this level. For example, the convolution, bias, activation operators in Equation 2.1 are often fused into single computational kernel. While a convolution is expensive to compute, the bias and activation are computationally cheap, *injective* operations. A driving performance factor in modern computer architectures are memory accesses. Thus it is more efficient to directly apply the bias and activation after the linear operation computes an individual result instead of storing the results in memory and then loading them again.

Modern DNN architectures like Inception Networks [95], ResNet variations [42, 113], NASNet [126] or ultimately randomly connected networks [114] have more complex wiring between operators, offering fusion opportunities beyond linear layers. The general problem of finding the best operator fusion for a specific hardware target and network is NP-hard [29] but practical solutions for this problem have been proposed [48], exploring an optimization space bounded by rules exhaustively. An alternative approach, using equality saturation [112, 69] is also explored [116], yielding promising results. Need or support of specific fusions can be determined or limited by the hardware architecture.

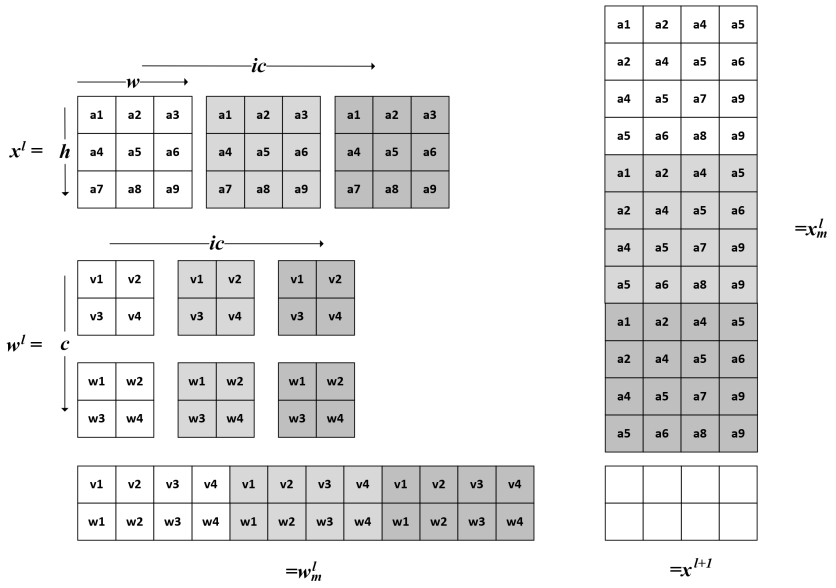


Figure 2.1 Example for the im2col process for a *Conv2D*. Matrices $\mathbf{w}_m^l$ and $\mathbf{x}_m^l$ are formed from the tensors $\mathbf{w}^l$ and $\mathbf{x}^l$. A matrix multiplication between $\mathbf{w}_m^l$ and $\mathbf{x}_m^l$ computes the output tensor x^{l+1}, which can directly be reshaped into its original layout.

2.3.2.2 Strategies

The strategy is concerned with data layout and hardware intrinsics used to implement a computational kernel. It is also the point where the implementations of individual operators become hardware aware. On the application level the most common data layouts for activation tensor in *Conv2D* are $NCHW$ and $NHWC$, image major and channel major respectively. The right-most, or inner-most, dimensions are consecutive in memory. The former is used by PyTorch, while the latter is native to Keras. However, for inference, more specialized data layouts are common. In Intel's x86 devices the $NCHWxc$ data format is preferred [35]. The channel dimension c is *tiled* by a factor of x to enabled efficient loading of x values into the SIMD unit. For 512bit SIMD width and 32bit floating point values x is 16. Further x86 layout and performance questions are explored in [44, 62] and the oneDNN library[4].

On GPUs, several memory-layouts are used, but *im2col* is the most interesting. Based on a method to perform *Conv2D* as a general matrix-multiply [103] (GEMM), several frameworks and libraries adopted the approach. The process is exemplified in Figure 2.1 By repeating segments of the image's rows or columns according to the stencil size to form a matrix $\mathbf{x}_m^l$ of size $ic \cdot k_h \cdot k_w \times n \cdot h \cdot w$ and reshaping the weights to $\mathbf{w}_m^l$ of shape $oc \times ic \cdot k_h \cdot k_w$, *Conv2D* is expressed as

[4]https://github.com/oneapi-src/oneDNN, accessed 04.2022

GEMM. The total number of operations remains the same, however the memory footprint increases. The larger a matrix is, the more efficient its computation becomes, in terms of moved bytes per operation. The matrices are constructed from the product of multiple convolution dimensions, consistently creating large matrices. This ensures, performance robustness over a wide range of convolution parametrizations. Further, GPUs can hide the construction of $\mathbf{x}_m^l$ by only creating smaller sub-matrices on the fly when loading them into the scratch-pad. This reduces the global memory footprint and avoids the latency of constructing the full matrix before processing. The approach is the base of cuDNN [21] and PyTorch's *Conv2D* implementation [72].

While *im2col* is known to a wider audience, other GEMM-based algorithms are known. Their design space is explored in [6] and experimental evaluation revealed several promising alternatives to *im2col*. Using additional processing after the GEMM operation, or breaking the operation into several smaller GEMM calls reduces the memory footprint compared to *im2col*.

Moving away from matrix multiplications, other strategies include convolutions using Winograd filtering algorithms [58] or FFT [99]. Winograd reduces the number of necessary multiplications and is relatively memory efficient, while FFT reduces the total number of operations necessary to compute the convolution but can potentially increase the memory footprint. Both hardware and workload influence which method has the lowest latency [125].

General purpose CPUs can leverage their high degree of freedom in programming, allowing highly specialized implementations, like indirect convolutions [27]. By using an indirection buffer that contains pointers to pixel-rows, a GEMM-based convolution can be performed without the memory overhead of im2col. Since dedicated accelerators often have limited or no pointer-arithmetic capabilities, such implementations are harder to realize. The memory indirection also makes compiler-based optimizations more challenging, as the data accesses is not strictly determined by loop iteration variables.

For CPUs and GPUs, the data layout is only of interest with respect to performance. Correct code could be generated for any layout. For dedicated accelerators, data layout and program structure must match precisely to allow the use of intrinsics. Often, this is specified by a human expert and the deployment toolchain ensures the required data layout is generated. Similarly, hardware intrinsics are mostly still placed manually for each operator. This limits the exploration of new operators or kernels created from graph-level operator fusion on a specific accelerator.

Table 2.1 AutoTVM parametrized schedule template. Each *knob* defines the values a template parameter can take. Picking a value for each knob specifies one configuration.

Name	Type	Possible values
c_nthread	Threading	1, 2, 4
h_nthread	Threading	1, 2, 4
reorder_h_c	Reordering	1, 2
tile_h	Split	1, 2, 4, 7, 8, 14, 16, 28, 56, 112
tile_w	Split	1, 2, 4, 7, 8, 14, 16, 28, 56, 112
tile_ic	Split	1, 3
tile_c	Split	1, 2, 4, 8, 16, 32, 64

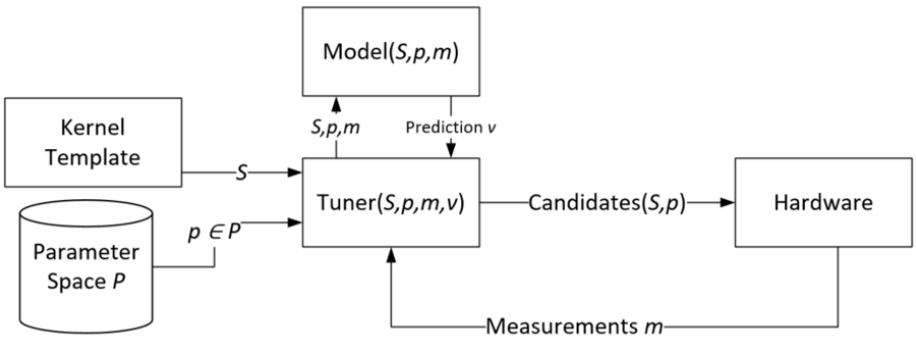


Figure 2.2 Auto-tuning system as used in AutoTVM

2.3.2.3 Schedule Optimization

When not relying on library implementations, the last optimization step is finding an ideal execution schedule for every individual workload on the used hardware. In the design of a single DNN, a wide range convolution parametrization can appear. For example, the first convolution in ResNet18 [42] defines c, h, w as $[64, 224, 224]$ with $ic = 3$, however the last residual block c, h, w as $[512, 7, 7]$ with $ic = 256$. Optimizing data-reuse, data movement and parallelization creates very different implementations for each layer. Of course, the hardware architecture also influence the implementation.

Since manual optimization is infeasible at scale, hardware-in-the-loop systems, searching for the best implementation by evaluating different candidates in hardware gained attention in recent years. Possible optimization targets are throughput (GOP/s), latency or energy per inference. The concept of these so called *auto-tuning* systems existed long before the recent surge of machine learning applications, but attention towards them was renewed due to the sheer size and complexity of the optimization problem. From loop transformations like tiling, reordering, unrolling, vectorization, fission or fusion a search-space is generated.

A widely used tool in this space is AutoTVM [18]. As shown in Figure 2.2, the tuning process is constructed from the following components: **i**) the search-space template and its parameters, **ii**) a tuner; the method to select the candidate for hardware evaluation, **iii**) a model aiding the tuner and **iv**) a measurement cycle, executing candidates on the hardware and providing measurements to the tuner. AutoTVM constructs the search space from parametrized workload implementations, where transformations like loop ordering, tiling scheme or the used hardware intrinsics are declared. For any given workload, the Cartesian product of options values for each declared transformations forms the search space. The indices over all knobs form a structured grid, such that every configuration is has a unique identifier. An example space for layer 1 in ResNet18 is in Table 2.1. In the example, a *split* operation on dimension h with value 16 creates an inner loop h_i of length 16 and outer loop h_o of length $\frac{224}{16} = 14$. The *reorder* specifies in which order loops h and c processed, and the *threading* determines the parallelization of these loops.

AutoTVM uses *Simulated Annealing* [52] (SA) in combination with a machine learning model to select candidates for evaluation in the measurement cycle. Gradient Boosted Trees [33, 16] are used to learn how different candidates perform on hardware, specifically a *ranking-loss* function is used to learn the relative performance of candidates, instead of regression to determine the absolute performance. SA traverses the search-space for a fixed number of steps, selecting candidates to evaluate based on the ranking provided by the model and which have not been evaluated in hardware, yet. Exploitation and exploration are balanced by adding random samples to the selected candidates. The resulting measurement batch is then evaluated in hardware and the gathered data used to further train the performance model. The process is repeated until a pre-defined number of steps is reached or no more candidates are available in the search space.

Different features can be used as input to the model. The simplest are values or indices of the tuning parameters themselves, which shows good performance for individual operators. However, the model is then specific for a given operator/hardware combination and cannot be used to tune a different operator. More complex feature sets are provided to enable *transfer learning*. Memory accesses, loop trip counts and other metrics are aggregated into a more extensive feature vector. Training the model on tuning runs from different operators on the same hardware improves the initial ranking quality and ultimately produces

$$\mathbf{r}_{i,j} = \sum_k \mathbf{q}_{i,k}\mathbf{p}_{k,j}$$

(a) Workload as a tensor expression

```
1    for j in J:
2      for k in K:
3        r[i,j] += q[i,k] · p[k,j]
```

(b) Workload in (one possible) loop representation

$$\mathbf{c}_i = \sum_k \mathbf{a}_{i,k}\mathbf{b}_k$$

(c) GEMV instruction as a tensor expression

```
4    for k in K:
5      b[i] += a[i,k] · b[k]
```

(d) Instruction in (one possible) loop representation

```
6    r[:,j] += GEMV(q·p^T[j,:])
```

(e) A possible embedding strategy

Figure 2.3 Examples for tensor expressions, the equivalent loop programs and a possible instruction embedding.

better implementations faster. The trade-off between specificity and portability is often search space and hardware specific.

Recently, tools like *Ansor* [124] are developed. Instead of specifying a template per workload, templates are specified on a per-hardware basis, giving more scheduling freedom. As of this writing, only templates for general-purpose architectures (CPU and GPGPUs) have been presented.

2.4 Related Work

Code generation for accelerators presents a set of unique challenges, like detecting where and how hardware intrinsics can be utilized. First a motivating example of this *embedding problem* is presented, before diving deeper into the material. The workload is a matrix multiply and it should be mapped to a hardware intrinsic performing a matrix vector product. Figures 2.3a, 2.3b, 2.3c and 2.3d show workload and instruction as tensor expressions, as well as possible loop implementations. The resulting implementation requires j calls to the scaled matrix vector product and a transpose on matrix $\mathbf{p}$ to match the access pattern of the instruction, as shown by Figure 2.3e.

For a human expert, such a transformation sequence can be very intuitive. To automate this, existing work mostly focussed on joint process of transformation and pattern matching [93, 17]. Different implementations of the workload are generated and checked if the loop nest and access functions match one of the instructions.

Unless the intrinsic can compute the full workload in a single call, schedule optimizations are necessary to generate well performing implementations. The embedding dictates which loops are mapped to the intrinsic, their ordering and whether they need to be split. That leaves the remaining loops as degrees of freedom for scheduling decisions. Thus, scheduling choices can only be made after an embedding is selected and therefore this work views them as sequential problems in the deployment.

A large body of publications concerning the deployment for DNN operators on hardware accelerators has been published in past years. To bring more structure into this discussion, it is split into the two already presented accelerator groups. The first group targets DNN accelerator with an ISA containing complex intrinsics, like matrix multiply. This group is closest to this work and the motivating problem. Second are tools to explore data and operation mapping in dataflow and CGRA architectures. This group is more specialized towards convolutional workloads and has a less general programming interface than the loop-back type.

This discussion will not include MLIR [57], which is aimed to serve as platform of interaction and optimization for different front ends, intermediate representations and back ends. It is not restricted to deep learning applications, or any other class program as integrations of different tools into the ecosystem demonstrate, for example like RISE [41] and XLA.HLO [1] in [64]. Many of the methods discussed in the following paragraphs could be implemented as a MLIR dialects or optimization passes for a specific target.

Tools targeting ISA accelerators Table 2.2 presents a selection of tools used to bring CNNs to loop-back style hardware accelerators, for example TensorCores in NVIDIA GPUs [5], Intel's VNNI instructions, VTA [67] or more specialized hardware like [109]. **Data-Layout Transform** describes if and what changes the tool can make to the data layout and **Embedding** marks if the tool can automatically rewrite the program as necessary to place the hardware intrinsic in the workload. Lastly, **Optimization** describes how the implementation is optimized before deployment.

TVM [17] is an open-source compiler stack for DNNs, describing networks in a functional language called *relay* [79]. Individual operators for execution are lowered to a scheduling language inspired by Halide [74]. Feedback-based

[5] https://www.nvidia.com/en-us/data-center/tensor-cores, accessed 12.2021

Table 2.2 Comparison of DNN deployment tools for ISA based hardware accelerators

Name	Workloads	IR	Data-Layout Transform	Embedding	Optimization
TVM [17]	CNN, MLP LSTM	Loops	×	×	AutoTVM [18] Ansor [124]
DORY [12]	CNN, MLP MLP	Loops Caches	×	×	CP
ISAMIR [93]	CNN, MLP LSTM	Loops, Registers	transpose	✓	computed at compile time
UNIT [111]	CNN MLP	Loops, Registers	×	✓	AutoTVM
Lift [94, 65]	CNN, MLP	Functional Lang.	×	✓	auto-tuning
AKG [123]	CNN	Polyhedral	×	✓	auto-tuning
Glenside [89]	CNN	Access Patterns, Equality Graphs	✓	✓	×

performance optimization, or auto-tuning, is then used to find good schedules for individual operators [18, 124]. Code for accelerators can be generated by embedding hardware intrinsics into the AST. This *embedding* is only semi-automatic. For every operator an expert has to specify an embedding strategy which statically defines the data layout and binds the instruction's loops to workload loops. Based on this, code for individual operators can be generated and optimized. Pre-determined strategies are limited in their ability to adjust to different operator layouts or parametrizations and are not part of AutoTVM's optimization space. Especially if a dimension in the instruction is larger than the dimension in this specific operator instance, workarounds like zero-padding are necessary, but reduce hardware utilization.

DORY [12] is on the same level of embedding automation as TVM and is a tool to automate the deployment of workloads to PULP-NN intrinsics [34]. Both data layout and the calls to accelerating functions are specified manually. Performance optimization is done by a *Constraint Programming* (CP) based process, searching for the best memory hierarchy utilization.

ISAMIR [93] automates the embedding problem at loop level by matching tensor dimensions and arithmetic operations in the loop nest to the ISA, striving to derive loop orderings and data layout from the embedding attempts. If this is not possible, a non-deterministic program transformation is performed. The resulting program is then matched against the ISA, again. How the tool deals with too-small dimensions on for the embedding, specific exploration strategies and a full set of rewrites are not reported.

Similar to ISAMIR, UNIT [111], which builds on TVM, automates the embedding process based on arithmetic operations and register addressing. It verifies that the addressed data and loop iterators mapped between workloads and instruction strictly overlaps. Program rewrites are limited to embedding. If no possible embedding is present in the workload, the deployment will fail. For example, in order to embed NVIDIA's Tensor Core intrinsic the workload needs to be specified in the im2col format by a human expert. UNIT is a first step towards unifying the embedding and scheduling problem by adding the space of different embeddings strategies as a dimension to the scheduling problem. It then relies on the tuner for the final choice. While this integrates the two steps, fundamentally different implementation strategies, like im2col vs. direct convolution, cannot be explored.

Rewrite systems to create different implementations for same computation are also used by LIFT [94, 41], for scheduling and exploration of possible embeddings of specialized hardware intrinsics into DNN operators [65].

Similarly, Glenside [89] is based on transformations of functional workload descriptions. It focusses on the representation and transformations of access patterns in *pure* tensor functions. Transformations include *transpose*, *slicing*, *flatten* or *reshape*. Glenside is using a equality saturation [112] solver on this representation to find solutions different kinds of problems, among other how to embed a GEMM intrinsic into a convolution via im2col. In their description, the authors of Glenside focus on the representation aspect. Performance optimization results are not discussed or presented.

AKG builds on TVM, but the back end is swapped for polyhedral scheduling and code generation facilities. Using im2col, convolutions are processed as matrices, of which subsequent fractal decompositions are mapped to a GEMM intrinsic. Similar to VTA or CPU targets, the input channel of the activation is packed and lowered to the innermost dimension. The powerful tiling and operator fusion of the polyhedral IR are used for aggressive memory access optimizations in the auto-tuning process.

Tools targeting dataflow accelerators The second group is about tools targeting dataflow architectures. This class of accelerators enables dynamic data routing in the hardware. Determining the temporal and spatial data movement through the accelerator is the main objective of this group. Global data-layout transformations are not explored, as they are mainly concerned with data locality in the processing elements, not outside it. Timeloop [71] has a focus on on loop-level program

rewrites for dataflow based accelerators such as Eyeriss [20] and DianNao [15]. Workload and hardware are abstracted over the 7 loops of a 2D convolution and user-defined annotations specify which hardware and instruction loops are possible mapping candidates. Timeloop then attempts to map the operator to the available buffer memories and processing units through successive tiling and reordering. Similarly, dMazeRunner [26] targets the same class of hardware and workload. Through the use of search space restrictions, they can reduce the search time to a few minutes. Both, Timeloop and dMazeRunner build on analytical models of energy, execution and latency to sweep the optimization space exhaustively. The embedding problem is not discussed, as they receive it as part of the problem definition.

While Timeloop and dMazeRunner focus on DNN workloads, Novatzki et. al. [70] propose a more general approach, solving an *Integer Linear Programming* (ILP) problem to find the ideal spatial placement. Similar is the work by Chaudhuri et. al. [13], with a SAT-based compiler for the dataflow in CGRA devices. It uses a flow-graph abstraction and SAT solving. The goal is to fully compile an application with a static schedule for a CGRA hardware target. Their main contribution is the static scheduling in time and space for a CGRA with a flexible dataflow architecture.

Only ISAMIR and Glenside concern themselves with automatic transformation of the data layout to make a deployment possible. UNIT, TVM for VTA and AKG rely on static, predetermined layouts. However, this is an important factor for DNN workloads, where different application frameworks offer various, layouts, like $NCHW$ in PyTorch and $NHWC$ in Keras. Likewise, specific hardware accelerator libraries and interface expect specific layouts, like $NHWC$ in the MLI library[6] or $NCHWnc$ for VTA. Manually translating these layouts for every operator and front-end creates overheads for describing the exact same computations. Similarly, manual effort is necessary for the compiler stacks presented in Section 2.3.2. The translation from high-level constructs to low-level intrinsics is done by a human expert and is not automated. Exploring and determining how high-level operators can be expressed as a sequence of low-level intrinsics is an open research question and motivates this thesis.

[6]https://github.com/foss-for-synopsys-dwc-arc-processors/embarc_mli, accessed 04.2022

Chapter 3

Complexity Analysis of Iteration-Domain Embedding

The following chapter will focus on the complexity analysis of transformation-driven embedding processes, operating on the loop level of DNN operators. We will call this the *Top-Down* approach. The fundamental problem to solve is finding a sequence of program rewrites that allows a pattern matching process to find a possible embedding location. The goal is to express the operator as a sequence of intrinsics, without manually specifying a sequence of transformations for a specific operator and intrinsic pairing. The implementation should be the result of an embedding process, which in itself should be easily adjustable and extendable for different hardware targets. This, of course is not trivial to achieve and this chapter will explore how current, transformation-driven processes are suited for this problem. Workload representation and embedding on the level of tensors, loops and tensor indices level reflects the abstraction level of existing publications in this field. As discussed in Chapter 2, the *scheduling* problem is not bound directly to the embedding problem and thus will be discussed in later chapters.

In general, embedding complex instructions into tensor computations has two main challenges. First, not every opportunity to embed an instruction is immediately visible in the original formulation of the kernel. There can be many reasons why an intrinsic cannot be embedded immediately. A first example is a possible mismatch between the dimension ordering of a tensor. This was already demonstrated in Figure 2.3. A transpose on one of the input matrices

The content of this chapter was published at the ITEM 2020 Workshop [76]

is necessary to enable an embedding. While for matrix-vector or matrix-matrix multiplication operations the number of possible orderings is limited, the seven different dimensions *Conv2D* make it more challenging to align the dimensions correctly between workload and intrinsic.

Another, trivial example is the order in which the input data tensors are specified in the computation. In the following two snippets only the ordering of inputs **a** and **b** are different.

```
1 for i in I:
2   for k in K:
3     c[i] += a[i,k] · b[k]
```

```
1 for i in I:
2   for k in K:
3     c[i] += b[k] · a[i,k]
```

At some point in the embedding process, either through pattern matching or transformations, this needs to be accounted for.

A last example is the distance, or pairing, of computations in a workload. For example, a scaled matrix-vector product multiplies each output element with a scalar value. It can be implemented as in the following snippet:

```
1 for i in I:
2   for k in K:
3     c[i] += a[i,k] * b[k]
4   c[i] *= s
```

Assuming the same matrix-vector intrinsic as in Figure 2.3, a direct mapping between workload and intrinsic is not possible. If the matrix-vector operation were replaced by a similar intrinsic, the scaling operation in line 4 would be dangling. A solution would be to move the scaling into a separate loop. Such a transformation is called *loop fission*.

```
1 for i in I:
2   for k in K:
3     c[i] += a[i,k] · b[k]
4 for i in I:
5   c[i] ·= s
```

As the examples showed, transformations are a possible path to enable the embedding of intrinsics. Different combinations of transformations lead to a space of computationally equivalent candidates.

The second challenge is searching this space for candidates that can be realized with a given set of intrinsic. In each created candidate, an instruction selection algorithm needs to find possible compositions of matching intrinsics. A pattern matching process needs to determine when loops and access functions overlap

in a way that allow an embedding, while also preventing matches that would break the correctness, as in the previous example.

To handle these challenges, this chapter presents a graph-based *intermediate representation* (IR) to capture the computational sequence and hierarchy as well as memory access functions of both, workloads and intrinsics. Specifically, this chapter presents:

- A graph-based IR to describe tensor computations and specialized tensor intrinsics. The IR represents the performed computation and data dependencies, allowing transformations and pattern matching.
- Algorithms for pattern matching and instruction selection in this IR.
- A method to generate a search space of legal kernels variants through graph transformations.

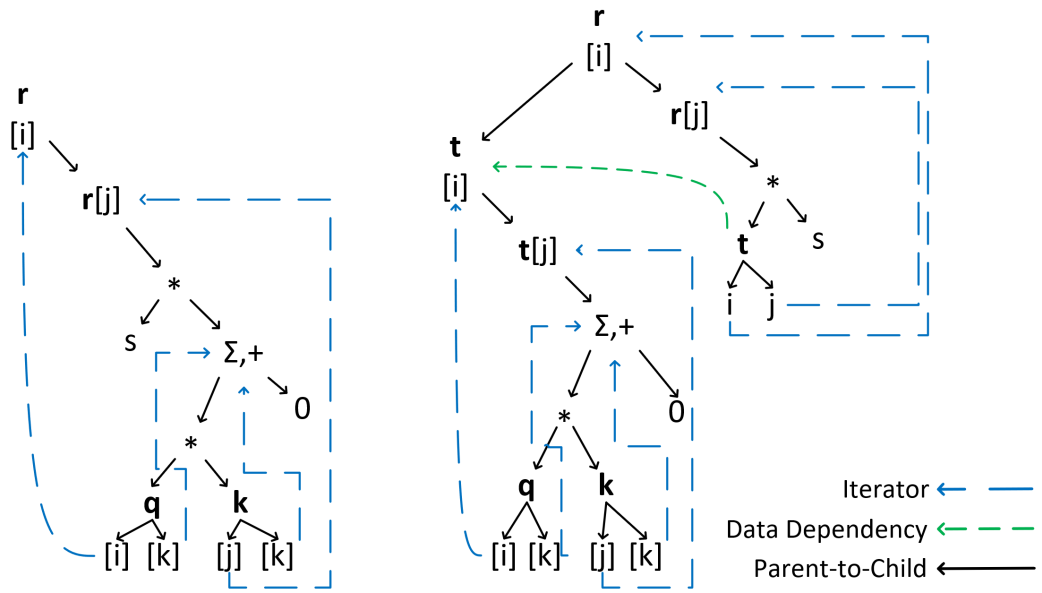
Section 3.1 describes the IR and Section 3.2 discusses the pattern matching algorithm. Section 3.3 presents a set of computational transformation of a deep-learning kernel to expose patterns for the mapping. The resulting complexity of transformations, pattern matching and search spaces is evaluated in Section 3.4.

3.1 Tensor Compute Graphs

First, the graph-based IR for tensor expressions is presented. It will be used to discuss the embedding problem in the following sections. The IR captures the computational properties of tensor expressions, together with data dependency and loop structure information in *Tensor Compute Graphs* (TCG). A TCG has nodes of several types: *tensor*, *reduction*, *compute*, *input* and *access* nodes. The nodes form a directed graph and each node has multiple children and one parent node. Data dependencies and index variable definitions are represented by additional edges. Formally, the graph is defined as:

Definition 3.1. *A TCG is a labeled, directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{D}, \mathcal{I})$, where $\mathcal{N}$ is a set of nodes and $\mathcal{E}$ the set of directed edges connecting $\mathcal{N}$ to form a rooted tree describing the computation. $\mathcal{D}$ is a set of directed Read-After-Write (RAW) data dependency edges. $\mathcal{I}$ is a set of memory access edges $\mathcal{I} : n_{in} \mapsto n_{it}, (n_{in}, n_{it} \in \mathcal{N})$ mapping each input dimension n_{in} to a node defining an iterator n_{it}.*

Nodes representing different parts of a tensor expression have different semantic meanings:



(a) Two Graphs for the same computation, but with a different formulation. The right-hand graph corresponds to equations, 3.2 and 3.3, while the left-hand graph expresses equation 3.1. For brevity, we left out the child-to-parent edges. Nodes with a capital letter followed by an interval ($\mathbf{t}[i]$) are *tensor* nodes. Nodes with bold letter ("**t**") represent tensor *Inputs*, lower case letters are scalar *Inputs*. Algebraic symbols (+,*) are *compute* nodes, nodes with a $\sum$ symbol are *reductions*. Lower case letters and symbols in brackets ([i]) are *access* nodes.

```
1  for i in r_i:
2    for j in r_j:
3      for k in ∑_l K:
4        r[i,j] = q[i,1] · k[j,1]
5      r[i,j] ·= s
```

(b) Loop-nest for left-hand side TCG

```
1  for i in t_i:
2    for j in t_j:
3      for k in ∑_l:
4        t[i,j] = q[i,1] · k[j,1]
5  for i in r_i:
6    for j in r_j:
7      r[i,j] = t[i,j] · s
```

(c) Loop-nest for right-hand side TCG

Figure 3.1 Examples of how TCGs represent loop-based computations.

- **Tensor** nodes describe the formation of an output tensor dimension and the compute flow. The child to the right of each *tensor* node describes how each of the dimension's elements is computed. The child to the left of a *tensor* node is the preceding computation in the program flow, the parent is the succeeding one. This does not reflect data dependencies, only the order of computations. Tensor nodes also define iterators over inputs. Similar to Static Single Assignment (SSA) IRs [80], the values in tensor nodes cannot be manipulated.

- **Reduction** nodes describe reductions of an input dimension over an operation. The child to the right is the initial value, the left-hand child

the input. Reduction nodes also define iterators over inputs. Reduction nodes compute a scalar value that can be used by the parent for further computation or storing the result.

- **Compute** nodes represent element-wise unary or binary functions. Children are inputs or preceding computations. They are ordered according to the function. For example, a subtraction has a strict operand ordering (right-to-left), while an addition has a relaxed ordering due to its commutative nature.

- **Input** nodes represent an input tensor with n children of type *access*, one for each tensor dimension. The children are ordered right-to-left, with the rightmost representing the innermost dimension. If a dependency exists, an edge points to the *tensor* node computing the dimension.

- **Access** nodes form an expression tree to describe memory access functions for a tensor dimension. They either carry a reference to a node defining an iterator, a constant scalar value or an arithmetic operation. Iterators are defined by either *tensor* or *reduction* nodes and describe the traversal of the dimension.

The graph construction rules are defined in Figure 3.2. The left-hand side graph in Figure 3.1a is used to explain the fundamental concepts. It shows the TCG representation of the following equation:

$$\mathbf{r}_{i,j} = s \cdot \sum_{l} \mathbf{q}_{i,l} \cdot \mathbf{k}^{T}_{l,j} \tag{3.1}$$

Further, Figures 3.1b and 3.1c are loop nests corresponding to the respective graphs in Figure 3.1a. This demonstrates how close the TCG is to the loop representation.

A graph is always connected and rooted in a *tensor* node with no *parent*, in this case $\mathbf{r}[i]$. The nodes $\mathbf{r}[i]$ and $\mathbf{r}[j]$ are *tensor* nodes defining the shape of the computed tensor, which is two-dimensional here. Generally, the shape can be inferred by recursively following the path of right-hand side children, until a non-*tensor* node is encountered. Each additional *tensor* node adds an inner dimension to the tensor. This nesting creates a rectangular iteration space. Assuming a linear memory address space, the innermost dimension is continuous in memory. Following the parent-to-child connection, the next node is a *compute* node, multiplying the result of the *reduction* and the static *input* s. The *reduction*

```
node = tensor
  | reduction
  | compute
  | input
  | access

iterator = reference to: tensor | reduction
operator = Arith/Logic op
interval = {
  upper : <integer>,
  lower : <integer>
}

tensor = { parent : tensor | NULL
  left child : tensor | NULL
  right child : tensor | reduction | compute | input
  data-user : input | NULL
  size: interval
}

reduction = {
  parent : tensor | reduction | compute
  left child : compute | reduction | input
  right child : input
  size : interval
  stride: <integer>
}

compute = {
  parent = tensor | reduction | compute
  left child = compute | reduction | input
  right child = compute | reduction | input
  op = operator
}

input = {
  parent = tensor | reduction | compute
  dependency = tensor  | NULL
  children = [access]
}

access ={
  parent = input | access
  val = iterator | operation | scalar constant
  children = [access] | NULL
}
```

Figure 3.2 Tensor Compute Graph construction rules

sums up the result of the scalar multiplication of different values in $\mathbf{q}$ and $\mathbf{k}^T$. Which input values are used is determined by the *access* nodes and the iterators represented by edges connecting to the defining node.

Note that a different notation of the problem leads to a different TCG representation. The right-hand side graph in Figure 3.1a corresponds to Equations 3.2 and 3.3:

$$\mathbf{t}_{i,j} = \sum_{l} \mathbf{q}_{i,l} \cdot \mathbf{k}^T_{l,j} \tag{3.2}$$

$$\mathbf{r}_{i,j} = s \cdot \mathbf{t}_{i,j} \tag{3.3}$$

In this case, Equation 3.2 computes a temporary matrix $\mathbf{t}$, which is then used in Equation 3.3 to compute the final output. Multiple computations in the same kernel form a sequence of compute subgraphs along the left-hand children of *tensor* nodes. Because $\mathbf{t}[i]$ is the left-hand child of $\mathbf{r}[i]$, it is the predecessor $\mathbf{r}[i]$ and computed first. In this case, there is also data dependency, which is defined by the edge between the computation of *tensor* $\mathbf{t}[i]$ and *input* $\mathbf{t}$. This edge enforces the ordering between $\mathbf{t}[i]$ and $\mathbf{r}[i]$.

In general terms, every graph and subgraph represents the shape of a computed tensor, followed by the *compute* and *reduction* nodes describing how each element is calculated. Therefore, only *tensor* nodes can have other *tensor* nodes their children. This is also specified in Figure 3.2. Because every element of a *tensor* node can be computed independently by the same subgraph, each element implicitly exposes computational parallelism. Further, a node is *dominating* all nodes in the subgraphs formed by their children. This means a node is *dominated* by all nodes following the parent connection until the root is encountered.

The presented IR is designed to represent strict loop programs with no branches, neither static nor data dependent. Programs can only consist of sequences of loop nests. Scalar operators can be wrapped in 1D tensors and loops of length one. This restricts the class of problems that can be described, but fits well to DNN operators and allows for easy program manipulation.

3.1.1 Data Dependencies and Memory Access

So far, the explanation covered the structural description of a computation. This sections details how memory access and data dependencies are represented and which kind of analysis this enables. Tensor access functions are described with three types of nodes. The first is the *iterator* defined by the data consumers,

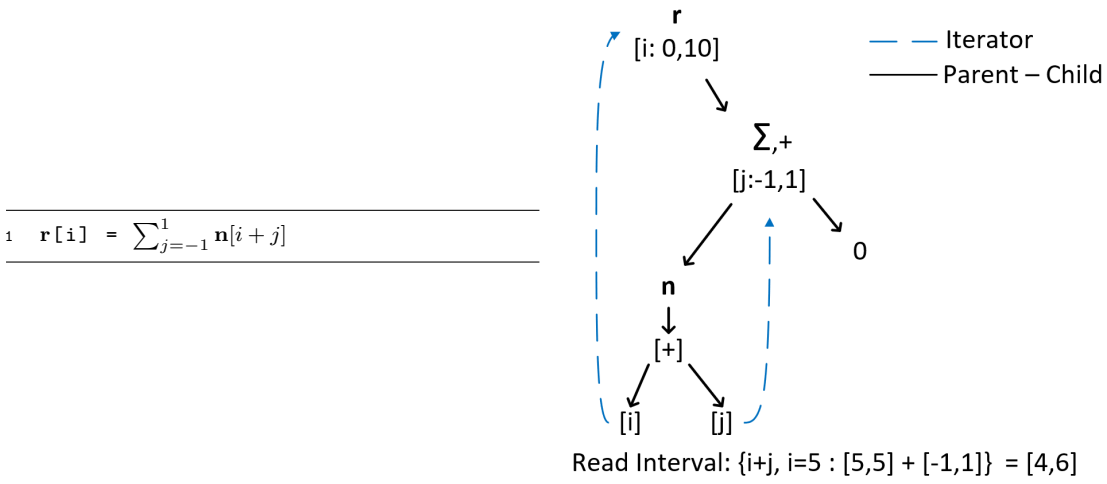


Figure 3.3 Tensor expression and simple graph of 1D sliding window

meaning that *tensor* and *reduction* nodes define indices used to access input dimensions. Second are *constant* values, like strides or offsets. The third are arithmetic operations, combining iterators and constant values. To express the access function computation, TCGs use an *expression tree* for each dimension. This expression tree forms naturally from the original tensor expression, as shown in Figure 3.3. The term $\mathbf{n}[i+j]$ becomes the subgraph under $\mathbf{n}$, which consist of node `[+]`, combining the values of the iterators `[i]` and `[j]`. The values of `[i]` and `[j]` are defined by the tensor node $\mathbf{r}$ and the reduction node $\Sigma+$.

Every *tensor* and *reduction* node defines an iteration domain as an integer interval $I = (l, n)$ with l und n being the lower and upper bounds. The iteration domain can have a regular stride $s > 1$. Interval analysis [66] can compute the read interval of every input dimension at each point of a computation. The *tensor* or *reduction* node at the "point of interest", and all dominators, contribute point intervals with a range of 1 from their currently computed elements, whereas each child node contributes its full interval to the access function calculation. Similar to the *tensor* nodes forming a tensor, the combination of read intervals forms a rectangular slice of the input tensor. The example in Figure 3.3 shows a simple 1D sliding window, where each output value in $\mathbf{r}$ is a sum of the pixel itself and its neighbors. To calculate which values are required for one value of $\mathbf{r}$, one has to compute the intervals according to the access function $i + j$ of $\mathbf{n}$. Iterator value $i = 5$ creates the input interval $[5, 5] + [-1, 1] = [4, 6]$.

Interval-based computation leads to an over-approximation of the accessed data for strided memory access functions. However, this only affects the computation of the access interval, not the index calculation. This conservative

approximation is often used in the DNN and image processing domain, for example by TVM or Halide. UNIT [111] uses a different method to evaluate the accessed memory but achieves a comparable analysis granularity. It decouples the computation matching from memory access analysis, verifying the former before the latter.

The last component in the TCG are data dependencies. Transformations of operations with sequential computations need to maintain correct ordering. Because tensor values cannot be modified in place, tracking read-after-write dependencies is sufficient to preserve the original program formulation. They are represented as edges between the producer and consumer of a tensor.

3.1.2 Intrinsic Representation

The hardware's intrinsics are also modelled as TCGs. It can be differentiated between intrinsics with variable size and intrinsics with a fixed size. A fixed intrinsic size has static input and output array dimensions. Since some architectures have instructions that perform the same computation, but with different input and output dimensions, we allow multiple iteration intervals for *tensor* and *reduction* nodes in the TCGs of intrinsics. For intrinsics with variable size, the dimensions of input and output tensors are part of the intrinsic's parameters. For this type of instructions, only the upper and lower bound of each argument are needed. Intrinsics are always rooted in a *tensor* node, and the computed results are always continuous in memory.

3.2 Intrinsic Embedding

Intrinsics and operators are described by the same structure, which makes instruction selection a pattern-matching problem. Intrinsic embedding is split into three steps: preprocessing, pattern matching and selection, each of these detailed in the following sections. After a preprocessing step, a modified version of the top-down tree matching algorithm [45] finds matching intrinsics in a kernel. Due to the TCG's semantics, customizations for pattern matching and instruction selection are necessary.

3.2.1 Preprocessing

The preprocessing takes a graph in the presented TCG IR and transforms it into one with strict tree property, meaning there exists only one path connecting

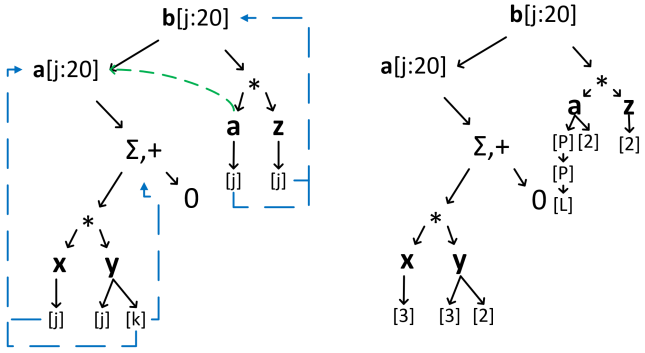


Figure 3.4 Example how a TCG is made a tree before pattern matching. In the right-hand side graph input **a**'s dependency becomes the path to the tensor node. P is for a move to the parent and L for a move to the left child node. Iterators only have moves in P direction. Thus, they are reduced to their distance.

any node pair in the graph. The parent and child connections already have this property. But edges formed by data dependencies and iterators introduce additional paths in the TCG. The edges created by iterator and data dependencies are resolved to paths from the input to referenced node by only traversing the parent and child edges. This is shown in Figure 3.4. The result is a graph with tree properties and a set of unique paths. Because the node defining an iterator is always dominating the input using it, the path only heads in the direction of the parent node, and thus, is reduced to an integer of the respective path's length. The path length then replaces the actual references in the access node for pattern matching. The data dependency paths are stored at the respective leafs of the tree. This process is the same for intrinsics and operators.

The last step during preprocessing is the creation of a data structure for the pattern matching from all available intrinsic graphs. In essence, the pattern matching algorithm reduces the problem of tree matching to a series of string matchings. To do this, the root-to-leaf path of every intrinsic's TCG is described as a string of labels. A label contains type information of a node, or a right-to-left enumeration of child nodes to encode the direction. The label information is gathered from the algebraic data types in Figure 3.2. Using the Aho-Corasick Algorithm [5] a finite-state machine (FSM) for root-leaf-path matching is generated. This FSM is the pattern-matching machine used to find embeddings in a workload.

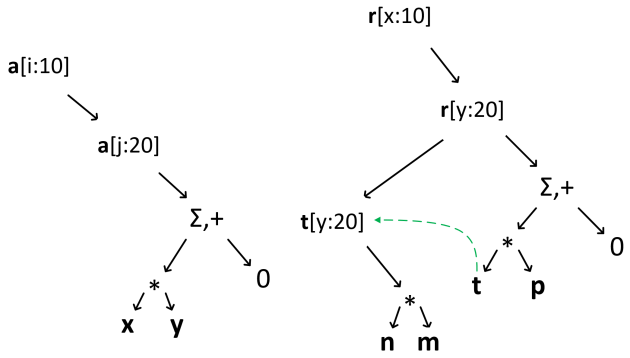


Figure 3.5 Example for a false match. The graph on the left is the pattern, the graph on the right is the kernel. Without the second matching condition, we would match the pattern root **a**[i:10] with kernel root **x**$[x : 10]$. This would cut off **t**$[y : 20]$ from the computation.

3.2.2 Pattern Matching

During pattern matching a workload TCG is traversed in right-to-left pre-order and each node is passed to the patter- matching FSM, which matches the operator against all previously inserted intrinsic in parallel. The result of this step is a set of nodes and the intrinsic matching at this point.

During matching the algorithm searches for identical root-leaf paths (label string sequences) between instructions and kernel. After a root-leaf path matched, a counter at the pattern's root node in the workload, associated with the matched instruction, is incremented. If the FSM state indicating a match has a data dependency path stored, then it is compared to the paths in the kernel. There are two conditions for a full match of an instruction: first, the counter associated with an instruction needs to match the number of leaf nodes in the instruction. This means that the same root-to-leaf paths exist in the kernel and the instruction and therefore the pattern matches. Second, the counter needs to match the leaf count of the right-hand side subgraph. This is a necessity of the execution model and IR semantic. The right-hand side subgraph of a *tensor* node describes how each *tensor* node element is computed. If the subgraph matches a pattern, the subgraph can be computed by this intrinsic.

As discussed in the beginning of this chapter, the pattern matching also needs to make sure, that a computation is not left dangling after a match. If a subgraph is not fully covered by a pattern, it means that not all required computations can be performed by the intrinsic. And since intrinsics cannot issue intrinsics themselves, there is no possibility to fill the gap with another intrinsic. This does not limit the ability to compose a kernel from several instructions, it only

prevents the generation of artefacts that cannot be executed in hardware. See Figure 3.5 for an example. The TCG on the left describes a matrix-multiply intrinsic. The graph on the right contains a computation, where the pattern of matrix multiplication is present but is only a part of full computation. Without the second condition, the matrix-multiply intrinsic would be matched without representing the computation correctly.

Since an intrinsic might support less input dimensions than an operator has, which affects the total leaf count, output dimensions can be masked out selectively to find matches.

3.2.3 Selection Algorithm

Pattern matching can produce multiple, conflicting ways of implementing an operator with intrinsics. A conflict exists if two matched instructions share one or more nodes in the tree. The selection is driven by a global cost optimization. Each intrinsic has an associated cost function that contributes to the global cost. For this work, coverage is used as a heuristic metric. Coverage is defined as:

$$C_{nodes} = \frac{sum\ of\ nodes\ covered}{total\ nodes\ in\ tree} \tag{3.4}$$

Maximizing coverage globally maps the highest number of nodes in the kernel to an instruction. The selection algorithm is agnostic to the exact cost function and can be used with other metrics.

Algorithm 1 is used to find the best combination of instructions recursively. Branches are bound if they offer no improvement over the best local selection. The bounding decision is made by analyzing the next possible match in the tree. For every intrinsic matched at the local node, the next intrinsic not overlapping with the local selection is selected. This next node is called *continuation*. Only the local selection with the best cost properties for each continuation is stored. The function returns the best candidate, which is a combination of the local selection and continuation. The algorithm locally selects the best instruction for the current node using the $MaxInstr$ function, $Next$ calculates the continuation. $Best$ selects the candidate with the best global cost function property. If the next node in the match list is not a possible continuation, a null pattern is used to allow exploration of search paths not selecting any match at this location.

Algorithm 1 Selection Algorithm

```
1:  function SELECTION(matches[], index)
2:      if index == matches.size then
3:          return MAXINSTR(matches[index])
4:      end if
5:      for instr : matches[index] do
6:          node,nextIndex = NEXT(instr, matches, index)
7:          maxCont = MAXINSTR(continuation[nextIndex], instr)
8:          continuation[nextIndex] = maxCont
9:      end for
10:     if continuation[index+1].empty then
11:         continuation[index+1] = Nullpattern
12:     end if
13:     for (index,node) : continuation do
14:         candidate = SELECTION(matches, index)
15:         candidate = candidate ∪ node
16:         candidateList.insert(candidate)
17:     end for
18:     return BEST(candidateList);
19: end function
```

3.3 Transformations

This section introduces how the transformations driving the search for possible operator implementations are handled. Transformations are necessary because the original implementations might not expose all matchable patterns to embed available intrinsics. The reason for this is the composable nature of workloads and the freedom provided by commutative and data independent operations. Computational correctness is maintained by only performing transformations that do not violate data dependencies.

Transformations are implemented as graph rewrites: $\mathcal{G} \xrightarrow{f_t(m)} \mathcal{H}$ is a *transformation* where $\mathcal{G}$ is a source graph and $\mathcal{H}$ the result graph. Function $f_t(m)$ is a transformation, applied on domain m. This domain describes a specific subgraph of $\mathcal{G}$ and is the working set of nodes and edges of transformation f_t.

For this chapter, two transformation properties are important: *independence* and *monotony*. The former describes if and when two different rewrite operations can be reordered but still create the same result graph. Two rewrites f_1 and f_2 are independent when the resulting graph of the transformation sequences $\mathcal{G} \xrightarrow{f_1(m_1)} \mathcal{H}_1 \xrightarrow{f_2(m_2)} \mathcal{H}_2$ and $\mathcal{G} \xrightarrow{f_2(m_2)} \mathcal{H}_2 \xrightarrow{f_1(m_1)} \mathcal{H}_1$ are equal. For this to be possible, the domains need to be *conflict-free*:

Definition 3.2. *Two transformations* $\mathcal{G} \xrightarrow{f_1(m_1)} \mathcal{H}_1$ *and* $\mathcal{G} \xrightarrow{f_2(m_2)} \mathcal{H}_2$*, are conflict-free if* $m_1 \cap m_2 = \emptyset$*.*

For this property, f_1 and f_2 can either be the same or two different transformations. The second condition, monotony, is defined as follows:

Definition 3.3. *A rewrite is monotonic, if for any number of* $n \in \mathbb{Z}^+$ *applications of* $\mathcal{G} \xrightarrow{f_1(m_1)} \mathcal{H}_1 ... \xrightarrow{f_1(m_n)} \mathcal{H}_n$ *of transformation* f_1 *over domains* $m_0, ..., m_n$*, the next application* $\mathcal{H}_n \xrightarrow{f_1(m_{n+1})} \mathcal{H}_{n+1} \notin \{\mathcal{G}, \mathcal{H}_1, ..., \mathcal{H}_n\}$*.*

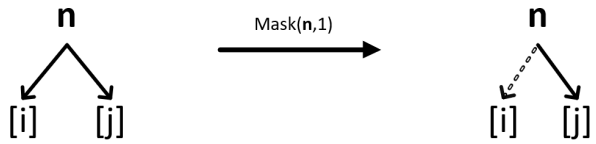
It means that the repeated application of transformation f_1 never generates a graph that was already produced by a previous application or is the original graph $\mathcal{G}$. Monotonic behaviour is important for the ordering of transformations. It prevents cyclic behaviour if a single transformation is repeatedly applied on a graph, allowing an exhaustive exploration of this transformation's space without additional measures to prevent cycles. The condition only applies to single transformations. Combinations of different rewrites can still create cycles in specific application sequences. Therefore, global transformation ordering is an important consideration during the design of a search space.

3.3.1 Set of Transformations

Graph rewrites, or transformations, are the central component of the embedding process presented in this chapter. The selected set of transformations is mainly concerned with structural rewrites, i.e. the arrangement of loops and individual computations or input ordering. An exception is the *masking* transformation, as it is necessary to generate matching patterns for any compute kernel with the used pattern matcher.

Input Masking Hiding outer input dimensions for pattern matching is necessary, because a pattern needs to cover all leafs of a subtree. This is displayed in Figure 3.6. If the number of dimensions in the operator is higher than the number of input dimension of an instruction, the patterns would not match. The missing dimensions are bridged by loops surrounding the instruction. This rewrite has non-monotonic behaviour.

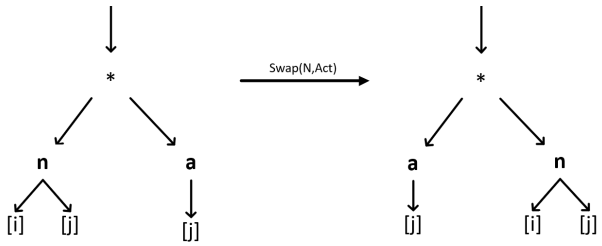
Operand Swap This transformation swaps the two child nodes of a *compute* node, see Figure 3.7. In pattern matching, the ordering of child nodes is relevant for matches. To perform a swap, the operation performed by the *compute* node




```
1   r[i,j] = n[i,j] · a[j]
2        Mask
3   r[i,j] =* n[j] · a[j]
```

Figure 3.6 Input Masking Transformation

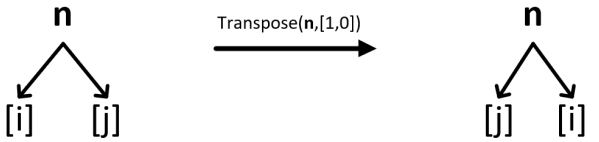
needs to be commutative. This transformation is non-monotonic. Therefore, if possible, the inputs of *compute*-nodes are ordered into a canonical form, reducing the number of required transformations.




```
1   r[i,j] = n[i,j] · a[j]
2        Swap
3   r[i,j] = a[j] · n[i,j]
```

Figure 3.7 Operand Swap Transformation

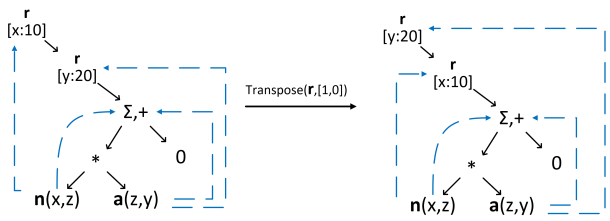
Input Transpose Reordering the dimensions in input tensors. In the TCG, this changes the children edge order of the input node. An example for this shown in Figure 3.8. This rewrite has non-monotonic behaviour.




```
1   r[i,j] = n[i,j] · a[j]
2        Transpose(n,[1,0])
3   r[i,j] = n[j,i] · a[j]
```

Figure 3.8 Input Transpose

Output Transpose Reordering the dimensions of the output tensor. In the TCG, this changes the order of tensor nodes. An example for this shown in Figure 3.9. This rewrite has non-monotonic behaviour.




```
1   r[i,j] = n[i,j] · a[z,y]
2        Transpose(r,[1,0])
3   r[j,i] = n[i,j] · a[z,y]
```

Figure 3.9 Output Transpose

Compute Fission Moves a *compute* node into a new subgraph, which only performs the computation described by the moved node. How this changes the graph is shown in Figure 3.10. All data dependencies of the original computation are now linked to the new subtree and a new dependency to the old subgraph is introduced. Only possible if there is more than one compute or reduce statement in the source graph. This transformation is monotonic.

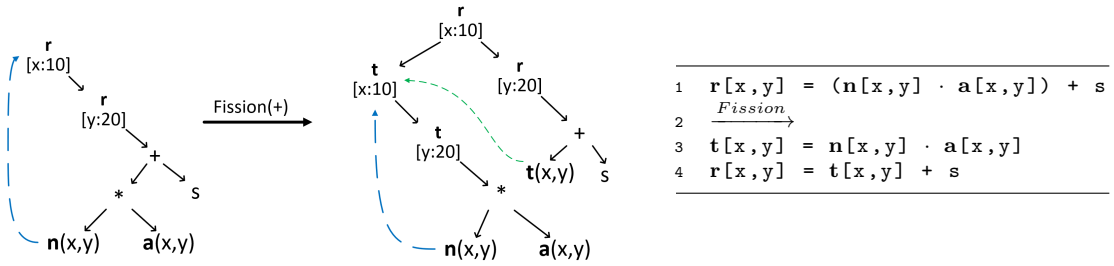


Figure 3.10 Compute Fission Transformation

Reduction Fission Moves a *reduction* into a new subgraph that only performs the reduction described by the moved node, as shown in Figure 3.11. The *reduction* in the source tree is replaced with an additional *tensor* dimension that is removed by the subsequent reduction. All data dependencies of the original computation are now linked to the new subtree and a new dependency to the old subgraph is introduced. Only possible if there is more than one compute or reduce statement in the source graph. This transformation is monotonic.

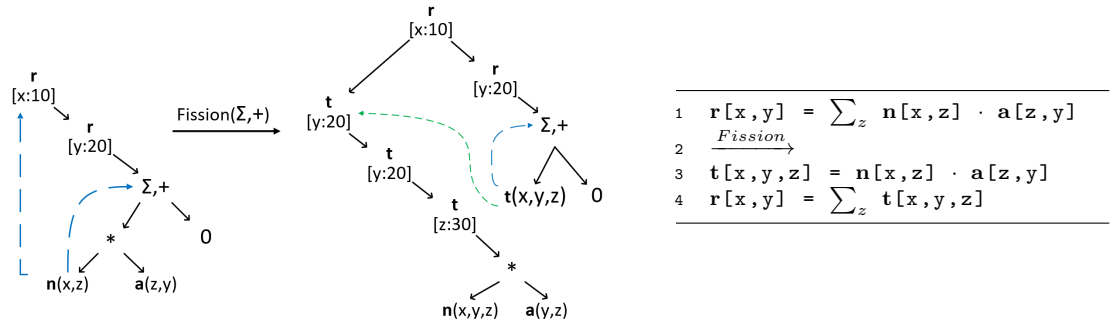
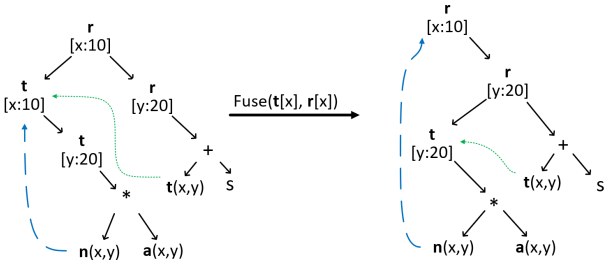


Figure 3.11 Reduction Fission Transformation

Fusion Reduces the distance between two computations, by fusing the left child of a *tensor* node into the parent. This is illustrated in Figure 3.12. This is similar to the fusion of loop nests. It allows the use of results before all dimensions of a tensor are computed. The fusion is only possible if the iteration interval is

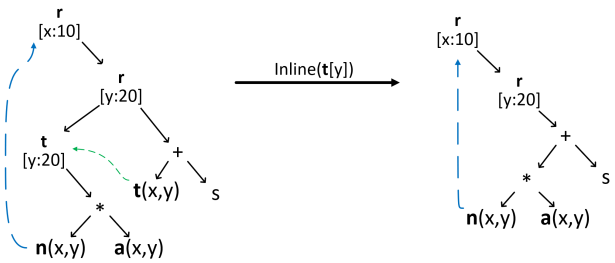
identical and no data is consumed before it is computed. This transformation is monotonic.




```
for x:
  for y:
    t[x,y] = n[x,y] * a[x,y]
for x:
  for y:
    r[x,y] = t[x,y] + s
  Fusion
for x:
  for y:
    t[x,y] = n[x,y] * a[x,y]
  for y:
    r[x,y] = t[x,y] + s
```

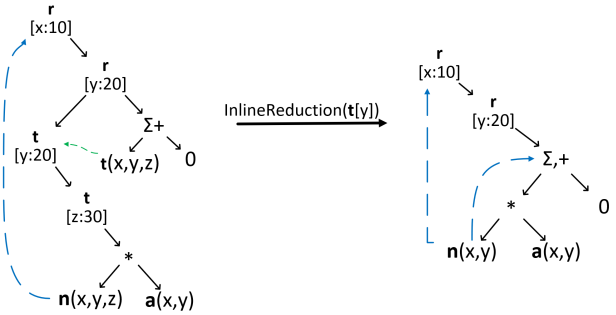
Figure 3.12 Fusion Transformation

Inlining Replace an input with the preceding computation. The source graph for *reduction* (Figure 3.14) and *compute* (Figure 3.13) nodes looks different, but the target graph has the same structure. Because multiple inputs could use this computation, the inlining is performed at every input node consuming the computation selected for inlining. This transformation is monotonic.




```
for x:
  for y:
    t[x,y] = n[x,y] · a[x,y]
  for y:
    r[x,y] = t[x,y] + s
  Inline
for x:
  for y:
    t[x,y] = n[x,y] · a[x,y] +
          s
```

Figure 3.13 Compute Inlining Transformation




```
for x:
  for y:
    for z:
      t[x,y,z] = n[x,z] · a[z,
          y]
    for z:
      r[x,y] += t[x,y,z]
  Inline
for x:
  for y:
    for z:
      r[x,y] = n[x,z] · a[z,y]
```

Figure 3.14 Reduction Inlining Transformation

3.3.2 Loop Tiling and Padding

Hardware intrinsics sometimes operate on a fixed width [67, 50, 46]. Deployment tools therefore need measures to handle loops that don't exactly fit the intrinsic. Primary tools for this are *tiling* and *padding*. Loop tiling splits a loop it into a sequence of individual iteration intervals. The inner loop iterates over the content of an interval, while the outer loop selects the interval. The goal is either to better utilize spatial and temporal locality in memory accesses or to modify the overall program structure to fit a specific hardware requirement. For the embedding, the focus is on tiling parameters necessary to match the intrinsics in an operator, as the general search for ideal tiling parameters is an orthogonal optimization problem in and of itself. Tiling is done by annotating the *tensor* node with the outer tiling factor, that is used during the code generation to create appropriate loops and instruction calls. This allows the search for good parameters globally, after the instructions have been matched.

$$\text{tiling} = \begin{cases} dim_{loop}/dim_{instr} \textbf{ if } (dim_{loop} + \text{padding}) > dim_{instr} \\ 0 \textbf{ else} \end{cases} \tag{3.5}$$

The second measure is the extension of tensor dimensions through *padding*. This extension either happens with zeros or ones, depending on the affected algebraic operations.

$$\text{padding} = \begin{cases} dim_{instr} * (\lceil dim_{loop}/dim_{instr} \rceil) - dim_{loop} \textbf{ if } dim_{loop} \% dim_{instr} \neq 0 \\ 0 \textbf{ else} \end{cases} \tag{3.6}$$

Both measures can be applied after the instruction selection, reducing complexity. First, the padding is determined for each *tensor* and *reduction* node in the graph mapped to an instruction by Equation 3.6. A second annotation contains the outer tiling interval by Equation 3.5.

3.4 Evaluation

Using the presented IR and instruction selection, the mapping of typical neural network layers to intrinsics similar to existing architectures like VIP [46] or Cambricon [61] is investigated. The evaluation architecture provides different classes of scalar, vector and matrix operands. When V is a vector, M is matrix and S a scalar value, Table 3.1 defines the instructions. Dot and matrix-vector

Table 3.1 List of available instrinsics.

Type	Operands	Result	Operations
Vector-Scalar	V, S	V	[add,mul,sub,div,max,exp]
Vector-Vector	V, V	V	[add,mul,sub,div,max,exp]
Reduction	V	S	[add, mul, max]
Dot	V, V	S	[add,mul,sub,div,max,exp][add,mul,max]
Matrix-Vector	M, V	V	[add,mul,sub,div,max,exp][add,mul,max]

Table 3.2 Canonicalization Table. Rules how different inputs are ordered to reduce the number of necessary instruction and transformation patterns.

OpType	Order	Canonical Form	Description
non-commutative	$[a, b]$	$[a, b]$	No change
commutative	$[a, b]$	$[a, b]$	No change
commutative	$[a_{const}, b]$	$[b, a_{const}]$	Constant values to the right
commutative	$[a, b_{dep}]$	$[b_{dep}, a]$	Intermediate values to the left

each perform two operations. The first is for the element-wise operation, the second for the reduction. The choice for this set of intrinsics was motivated by its flexibility, in order to support a wide range of different operators.

On the application side, six operators from neural networks are under investigation. They range from operators like a fully-connected layer, a variation of the *dense* operator from Chapter 2, to kernels with more complex operation sequences, like *Capsule Routing* [83]. The *Attention* workload is implemented according to Equations 3.2 and 3.3 in Section 3.1. Table 3.3 shows a complete list.

Since padding and tiling can be applied after the embedding, they don't contribute to the complexity of the embedding problem. Therefore, the dimensioning of workloads and instructions aren't considered in this evaluation. This decision was made in the full awareness of the reduced utilization and performance introduced by padding. However, it is in line with existing work such as TVM [17].

3.4.1 Transformation and Selection Process

Starting from the original kernel $\mathcal{G}$, sequences of transformations presented in Section 3.3 lead to many possible implementations. The creation process is structured to successively generate every possible implementation. Individually, the *fusion* and *fission* operations are monotone. However, their complementary nature can introduce loops into the search. Therefore, this work uses a fixed order of transformations:

1. The search starts with the *canonicalization* of inputs using the *swap* transformation, if possible. The rules of canonicalization are specified in Table 3.2. The goal is to remove variations created by commutative operations that still have other possible ordering criteria. *Constant* values are pushed to the right, while inputs with a dependency are pushed to the left. This helps reduce the number of necessary matching patterns for instructions and transformations for the workload. The case of two constant values is not considered under the assumption that operations on two constant values are computed ahead of the embedding.

2. The results of the canonicalization process is then split through iterative *fission*, until no further decomposition is possible. This is shown in Algorithm 2, where the two parts of the graph rewriting system are demonstrated, as well. First, in line 2 all possible fissions locations in a given graph are determined and put into a queue. Then, each operation is performed subsequently, modifying the original graph (line 6). On this new graph, the new fission options are evaluated and the previous step repeated (line 7). This process monotonously constructs a graph with more individual loop nests, until no further fission is possible. The result is a graph $\mathcal{F}$, representing a workload where each algebraic operation exists in an individual loop nest.

3. The graph resulting from *fission* is now subsequently *fused* together in every possible permutation. The *inline* transformation is then applied, as well. First, every individual option for a fusion is determined (line 5). From the available options, a *fusion set* is created. This set contains all conflict-free combinations of fusion options. Conflicts are determined as in Definition 3.2. All graphs from the possible options are generated (line 7) and then put into operation queue and solution set (lines 9, 10). Multiple sequences of fusions can lead to the same graph. To prevent duplicate entries in the result set $\mathcal{S}$, each candidate is filtered against already existing solutions (line 8). This is done with the same pattern-matching algorithm as used intrinsic selection. This process is repeated until no more options for fusions are available. The monotonous behaviour of the fusion operation guarantees a stopping point.

4. The resulting set of graphs is then matched against the instructions, with updating *masking* transformations in each iteration, until all possible input mappings are applied. Since the highest number of input dimensions in

Algorithm 2 Algorithm for finest fission

```
1: procedure FULL FISSION(𝒢)
2:     𝒪 ← FindFissionOptions(𝒢)
3:     while 𝒪 ≠ ∅ do
4:         while option ← 𝒪.pop() ≠ ∅ do
5:             𝒢′ ← ApplyFission(𝒢′, option)
6:         end while
7:         𝒪 ← FindFissionOptions(𝒢′)
8:     end while
9:     return 𝒢′
10: end procedure
```

Algorithm 3 Algorithm for full fusion

```
1: procedure FUSIONS(𝒢)
2:     𝒬.push(𝒢)
3:     while 𝒬 ≠ ∅ do
4:         𝒢′ ← 𝒬.pop()
5:         𝒪 ← FindFusionOptions(𝒢′)
6:         for option ∈ fusion_set(𝒪) do
7:             ℱ ← ApplyFusion(g, option)
8:             if ℱ ∉ 𝒮 then
9:                 𝒬.push(ℱ)
10:                𝒮 ← 𝒮 ∪ ℱ
11:            end if
12:        end for
13:    end while
14:    return 𝒮
15: end procedure
```

our instruction set is two, every higher dimension is masked out by default, reducing the number of masking combinations. The mapping results are then stored for further processing.

3.4.2 Solution Space

Table 3.3 shows the evaluated operators, together with characteristics of the search and solution space. The number of generated candidates varies immensely, especially between $Attention$ and $Conv2D$. Although they have the same number of nodes, the number of resulting candidates differs by two orders of magnitude. This is rooted in the structural differences of the graphs. $Conv2D$ generates a 4-dimensional tensor, using reductions and multiplications over a stencil memory access pattern. The second step in the search space creates a subgraph for each algebraic operation in the kernel. This leads to multiple subgraphs, each with at

Table 3.3 List of evaluated operators. Hit rate is the ratio of solutions to candidates. Number of transformations (#trans) is reported as median. Search configuration 4 in Table 3.4 is used.

Operators	#nodes	#candidates	#solutions	Hit rate	#trans
FC + Bias	16	24	4	16.6%	7
FC + Bias + Sigmoid	29	144	4	2.7%	12
GEMM	11	35	4	11.4%	8
Attention	26	289	9	3.1%	11
Conv2D (NHWC)	26	59,503	682	1.1%	25
Conv2D + ReLU (NHWC)	56	440,428	1,428	0.3%	33
Capsule Routing	127	1,023,516	12	0.001%	32

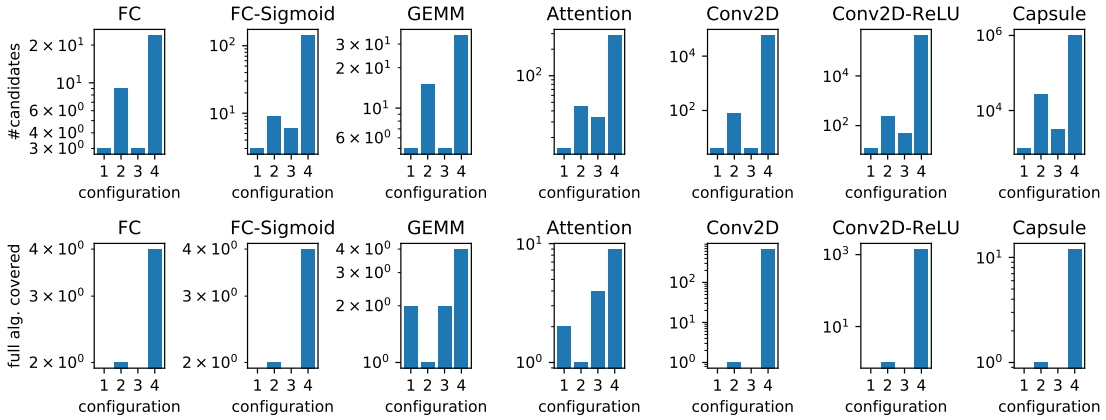


Figure 3.15 Impact of four different transformation configurations on finding full algebraic coverage. The configurations are detailed in Table 3.4. The top row shows the number of generated candidates, the bottom row the number of candidates with full coverage.

least four *tensor* nodes, which increases the number of possible combinations of fusions. In the $Attention$ kernel, the fission only creates three subgraphs with a maximum of three *tensor* nodes. The same explanation holds for $FC + Bias$ and $FC + Bias + Sigmoid$, as they are a sequence on vector operations, instead of complex, high-dimensional computations. By far the most candidates are created by the $CapsuleRouting$ algorithm, as its sequential nature creates many possible opportunities for transformations.

The fifth column in Table 3.3 shows the number of solutions, which are candidates that have 100% algebraic coverage. Algebraic coverage is the mapping of *compute* and *reduction* nodes to an instruction. In other words, the evaluation ignores the parts of the graph that represents loops and focus on the computational part. If an operator can be computed by a sequence of available instructions and possibly outer loops to cover tiling and outer dimensions of the tensor, it has 100% algebraic coverage. Only a fraction of the generated candidates can actually

Table 3.4 Configurations in the transformation study

Config. No.	Swap	Mask	Fission	Fusion + Inline
1	✓	✓	x	x
2	✓	✓	✓	x
3	✓	✓	x	✓
4	✓	✓	✓	✓

be fully mapped onto the hardware. While the total number of solutions mostly increases with the search space, the sixth column shows that the relative number of solutions shrinks with an increased number of candidates. An exception to this is *Capsule Routing*. Compared to the other workloads, the number of possible combinations in the masking operation explodes due to the higher number of sequential computations, and resulting in more of *input* nodes consuming the result of previous computations. At the same time, the possible number of *fusions* is lower, due to the lower dimension count and less regular output layout.

The median number of transformations required to achieve full algebraic coverage shows only a modest growth compared to the absolute number of generated candidates, which can grow exponentially. This suggests that the combination of transformations is more important than their total amount. One consequence is the need for careful pruning of the search space. Cutting a branch too early could potentially discard a whole class of candidates.

3.4.3 Transformation Impact

Different configurations of transformations were used to study the impact of transformations on the search space and the resulting candidates. The configurations are described in Table 3.4. *Swap* and *Mask* (Config. No. 1) are always included because they are necessary to generate any match. The data in the top row of Figure 3.15 shows that the configuration with the most transformations creates the most operators. The relative differences between the configurations remain similar across kernels, regardless of the absolute number of candidates. Except for $GEMM$ and $Attention$, no workload can produce a solution with full algebraic coverage only relying on configuration 1. This is explained by how $GEMM$ natively matches the matrix-vector product and $Attention$ natively contains the matrix-vector and scalar-vector operators.

Adding *fission* (Config. No. 2) creates solutions for all operators with a convolution as part of the workload because the standard convolution can be separated into a sequence of vector and reduction operations. This increases the

number of intermediate results, which in turn increases the number of inputs dimensions in intermediate computations. Thus, more candidates are generated because each input dimension leads to a higher number of masking options.

The number of candidates by Config. No. 3 depends on the number of compute statements in the original operators, as it only performs fusion and inlining operations. $Conv2D$ and $GEMM$ create the same number of candidates as the first configuration. They have only one subgraph in the original specification and therefore no option for fusion. $Conv2D + Relu$, $Capsule$ and $FC + Sigmoid$ offer some options for fusions, but for neither it is helpful to actually find a full-covering solution.

Config. No. 4 uses the full search space design as proposed in 3.4.1, leading with fission, then subsequent fusion. The high number of candidates for this config across all workloads is explained by the dynamic between these two transformations. Each fission creates additional computation subgraphs, which in turn creates opportunity for fusion. More interesting is the fact that for all original operators, most candidates with full algebraic coverage are found by this configuration. This suggests that finding the right composition of algebraic operations is important to generate valid solutions. From the original formulation, the workload must be transformed, such that the granularity of the individual computations match the operations available in the intrinsics set. Generating a set of transformation that creates the required granularity is an important goal during search space design. While the exhaustive approach eventually reaches a solution, it does not scale, neither with more transformations nor with workload complexity.

3.4.4 Data-Layout Transformations

In addition to changing the structure of a computation, the data layout is also a crucial part of the search space. For example, TVM and TCG try to match innermost tensor to the instructions. Therefore, not only the loop structure, but also the memory layout is relevant for an embedding. This section studies the effects of memory layout transposition on the search space and solutions using *transpose* on $Conv2D$ and $GEMM$ as an example.

$Conv2D$ has one output and two input tensors, with four dimensions each. The search space from Section 3.4.1 is extended by preceding the fission step with a transpose of the input and output dimensions. This way, no transposition of intermediate results is necessary. In the case of $Conv2D$, many deep learning

Table 3.5 Effect of tensor transpose operations on the search space. Hit rate is the ratio of solutions to candidates.

Kernel	#candidates	#solutions	Hit rate	candidates rel to Table 3.3	solutions rel. to Table 3.3
GEMM	73	5	6.8%	2.08	1.25
Conv2D	1,372,653	3,810	0.27%	23.06	5.5

frameworks offer predesigned tensor layouts, mostly image major (*NCHW*) and channel major (*NHWC*), for all tensors in the operator. However, to explore the more general case of matching any workload to an arbitrary instruction, knowledge about the semantics of tensor dimensions cannot be assumed. This means we must explore all possible layout permutations of the four dimensions. All inputs and outputs are transposed in the same way, since there are 24 different permutations of four dimensions. Transposing them individually would create 13824 base cases for the rest of the search space evaluation. Considering the already expansive search space, this would be computationally intractable.

Table 3.5 shows the number of candidates in the extended search space, the found solutions, and the percentage of solutions creating full coverage. The number of candidates increased roughly linear with the number of possible tensor permutations, with a factor of 23.06 for $Conv2D$ and 2.08 for $GEMM$. The rightmost column shows the factor of how the absolute number of solutions changes compared to Table 3.3. The increase of solutions is much lower than the increase of candidates. This adds to the observation that for an increase in the complexity of the problem, the number of viable solutions to the embedding problem increases in absolute numbers but shrinks in relative terms.

3.5 Discussion

The main goal of this chapter was the evaluation of the complexity to embed accelerator intrinsics into workloads. An important factor in the design was to solely rely on an algorithmic application description, without any knowledge about the hardware embedded into it. The hardware knowledge should purely reside in the intrinsics and embedding process, both of which should be relatively easy to describe and adaptable to other hardware targets. Creating a solution with low overhead, that allows the efficient deployment of new operators on existing hardware or existing operators on new accelerators was a motivating factor in the experiment design. This stands in contrast to the existing deep learning deployment research, where for established operators a plethora of

work on performance improvements exists for general purpose hardware targets but support for more exotic operators or hardware targets is often lacking.

The embedding problem was approached with a custom, graph-based IR for deep learning workloads and methods to automatically map these to one or more intrinsic available in an accelerator. Through graph rewrites and pattern matching, multiple valid solutions for common ML operators can be produced with this method. The choice to implement a new data structure instead of relying on existing ones comes from limited support of some rewrite operations and embedding features in TVM [17] and the TVM-based UNIT [111] implementation, as well as no publicly available source code of ISAMIR [93]. Specifically, TVM has no direct support to embed multiple instructions into the same tensor expression and no support for arbitrary operation fusion and fission. This work focussed on the complexity of the embedding problem, investigating how a transformation driven process scales and which issues occur from such an approach.

3.5.1 Experimental Results

The evaluation in Section 3.4.2 showed that a relatively small number of transformations is enough to find an embedding, but selecting the right combination of transformations is important. The naive approach presented in this chapter, without any sophisticated methods to select possible transformation sequences, shows sufficient ability to find possible embedding for small operators, like *Dense* and its variations like $Attention$ or FC. Even with further complexity like memory layout permutations, multiple legal solutions are produced with reasonable effort.

With increasing kernel complexity, the process becomes more expensive, as demonstrated by Table 3.3. At the same time, diminishing returns on the results are observed. Especially $Conv2D$ variations or sequential kernels like $CapsuleRouting$ show this effect. While the latter can be broken down into sequences of simpler operations, complex kernels like $Conv2D$ cannot be simplified further. It is to assume that this problem is amplified by more complex memory access patterns and transformations.

3.5.2 Limitations of Transformation-Driven Embedding Methods

The exploration shows major challenges in the search space design. Even the limited set of transformations presented in this chapter can enlarge the search space to intractable sizes. This goes hand-in-hand with a shrinking relative number of solutions, visible in Tables 3.3 and 3.5. Operands with sequential computations, like $Con2D + ReLU$ or $CapsuleRouting$, are especially affected by this. Additionally, transformations like fission can introduce such sequences even in simpler operators.

Another difficulty is the top-down decision making for the transformations, without knowledge about how this will affect the search space and produced solutions. Using more transformations can ultimately not help to solve the embedding problem due to the growing complexity.

Solving this decision problem is vital to make any embedding process feasible. Glenside, and ISAMIR partially, present possible solutions to this problem. Glenside is based on equality saturation, a powerful tool to handle rewrite problems and ISAMIR can infer some rewrites, mainly transpose and loop ordering, from the embedding result. On the other hand, UNIT is restricted to the provided data layout. For *Conv2D* on GPU, im2col must be performed before the deployment is presented to the embedding process. TVM as no facilities for embedding through automated program transformations, at all.

For Glenside, as well as ISAMIR, it is not described how to integrate different hardware architectural considerations into the transformation process. For example, how to treat dimensions that could be mapped between intrinsic and operator, but aren't even divisors or too small to fit the intrinsic. TVM and UNIT will fail if the workload dimensions are too small. The post-embedding padding used in this work enables embeddings in such cases, but it is also potentially limiting to the performance. None of the existing frameworks, including the one presented in this chapter, can fully capture loop and data-layout transformations, as well as hardware architecture details. An example for this are embeddings where multiple workload dimensions could be fused into a single dimension that matches a single intrinsic dimension. They need to be explicitly created before the pattern matching, since existing solutions always evaluate a one-to-one mapping between intrinsic and workload dimensions.

A last practical problem in the presented approach is detecting the possibility for a specific rewrite. When reusing the same matching algorithm than the one

used during the intrinsic embedding, different patterns for the same rewrite are necessary, since this mapping algorithm only looks for exact matches. This can be especially difficult when it comes to the analysis of access functions. When aiming to be as workload agnostic as possible, subtle differences can create missed rewrite opportunities. While this issue was not specifically investigated in the experiments, it can pose a potential issue when industrializing such a method. A possible solution would be a more powerful algorithm detecting variations, for example fuzzy matching.

The IR presented in this chapter, as well as TVM, ISAMIR, UNIT or Glenside approach embedding on a level that is easy to reason about for a human, in a tensor or loop-based representation. They operate more or less directly on the handmade specification of a workload, by applying loop rewrites like split or unroll. The languages to program the workloads enable freedom in the expression for users, which can lead to multiple different specifications for the exact same task. Further, they usually express the task in an inherently sequential way, as nested sequences of linear loops where one computation is done after another. Parallelism or reductions are then extracted from context and language constructs, i.e. by assuming that loops created by the output tensor are inherently parallel or using specifically annotated reduction loops. All this makes it easier for humans to express themselves, but the embedding process now has to account for all of this to detect similar computations.

3.5.3 Outlook

The insights from this chapter lead to the conclusion that transformation driven processes on high-level, human-readable representations have limitations when it comes to finding embeddings. $Dense$-like operators are often simple enough to find solutions with reasonable effort. For complex operators like *Conv2D* scalability is limited, especially when a wide range of transformations is available.

Instead of more powerful transformations processes, a detailed analysis between prospect instructions and the workloads could be an alternative path forward. From such an analysis, the necessary transformations are then derived, with an emphasis on loop structure and data layout. This is supported by the low, actual number of transformations necessary to create a kernel where an embedding is possible. Another aspect of limiting the complexity is to find an embedding locally and only for a limited number of instructions. For example, only between operators and instructions that share the same arithmetic

operations embeddings are attempted. And instead of operating on human-readable representation, a solution should directly model arithmetic and logic operations, their dependencies, and which subset of data is accessed to compute a result without indirections over loops, access functions and iterators. The next chapter will discuss such an approach by representing the problem as a scalar dataflow graph and using constraints to describe the solution space.

Chapter 4

Joint Program and Data-Layout Transformation through Dataflow Embedding

The previous chapter discussed issues with the existing practice of instruction embedding on the iteration domain level through program transformation. Three issues were observed during the analysis, the first of which is the opaque search space. Whether a graph rewrite leads to an implementation is not known before the transformation is performed and an embedding is attempted. The second issue is the combinatorial complexity of the problem. Each new transformation increases the search space of potential solutions. The third issue is the mapping of data layouts and tensor access functions. While transpose operations can be handled gracefully [93], more complex operations like im2col are harder to enable [89]. And dilated or depthwise convolutions require additional rewrite strategies to create embeddings. A top-down process, as discussed in the previous chapter, requires checks to handle each of these cases individually, increasing the complexity of the search space design.

This chapter will introduce a novel method to compute possible mappings between hardware intrinsics and workloads, specifically addressing these issues. Principally, the problem is solved on scalar operations instead of loops. This removes arbitrary implementation variations like loop ordering, data layout or access functions from the embedding problem. Further, the representation contains all possible loop and tensor permutations for the problem in a single dataflow graph, without the need for transformations. Also, the approach in

The content of this chapter was published in ACM's Transactions on Architecture and Code Optimization [75]

Chapter 3, as well as existing work like ISAMIR or TVM [93, 17] directly transform their IR structure to produce novel implementations. The approach presented in this chapter will use features of a found embedding to directly generate or manipulate code, without modifying the underlying DFG based IR and thus, decoupling these issues.

Section 4.1 will introduce the method on a theoretical and conceptual level, mainly how dataflow graphs (DFGs) are utilized to solve the embedding problem. It introduces the construction rules and properties of a DFG, defines the instruction embedding problem and identifies the challenges of this method. The subsequent Sections 4.2 and 4.3 describe the specific implementation to overcome these challenges. Specifically, how polyhedral program representation and constraint programming can be used to define a search space containing all possible solutions to the embedding problem, without explicit transformations. After that, Sections 4.4 and 4.5 demonstrate how the embedding results are combined with program and data transformation rules to generate code for the *Versatile Tensor Accelerator* (VTA) [67] hardware.

4.1 Solving the Embedding Problem on the Dataflow Level

This section will introduce the DFG-based embedding problem on the conceptual level. Implementation details are discussed in Sections 4.2 and 4.3. The fundamental challenges of going from a high-level workload representation, like Figure 4.2a, to an implementation executable with available hardware intrinsics remain. However, the approach in this chapter reverses the order of Chapter 3. First pattern matching determines the embedding and then the final program and data layout is derived from it. This is enabled by a fine-granular program representation working on scalar operations, namely dataflow graphs. In a DFG there are no dimensions, access functions or other implementation details. This reduces the complexity of the underlying matching algorithm and the IR itself.

This chapter implements a three-layer deployment stack. First, the network is partitioned into individual operators. To do this, each operator is checked, if an embedding attempt is feasible. This is done by comparing the arithmetic operations and operator arity of the kernel to the instruction. If they do not match, an embedding attempt is futile. If the operator matches the intrinsic on this superficial level, an embedding is attempted, which is the second step

of the deployment. Lastly, after a successful embedding, a target specific code generation performs data layout and program transformation to enable deployment. The result is the deployment *strategy*. Strategies are meta-patterns for deploying an operator on a specific hardware, containing both data layout and algorithm information. It is not as generic as the algorithm, but also lacks the details of a specific schedule. A popular example for this is im2col for convolution processing, but even simpler processing variations like the transposition of operands can be seen as a strategy. What they all have in common is combined transformation of both data layout and operator in one way or another, while maintaining the same computational properties, except for floating point associativity. This is handled with a rule-based process. The found embedding is matched with a set of transformations, which are then applied. Section 4.4 shows how a set of transformation rules defined over computational patterns like parallelism, reductions or stencils can generate code.

This chapter will focus on the second and third step, embedding and code generation. The second step is the main contribution of this chapter, describing a new approach to the problem. Code generation is always hardware specific, and an example for the third step is given on the VTA hardware. Research towards the first step, the general problem of ideal network decomposition, is covered by TASO [48] on the practical side, as well as an extensive theoretical analysis in [29]. Therefore, we choose not to further pursue this research direction and instead rely on the existing partitioning tools of TVM.

4.1.1 Data Flow Graph and Subgraph-Embedding

The basic workload and intrinsic representation are dataflow graphs. A DFG represents every scalar operation necessary to perform a computation as a directed graph. Nodes represent operations and edges the data dependency from a producing to a consuming operation, also called dataflow. Formally:

Definition 4.1. *A DFG is a labelled, directed graph, defined as* $\mathcal{G} = (\mathcal{N}, \mathcal{E}, l)$*, where* N *is the set of nodes, the set of directed edges is* $\mathcal{E} \subseteq \mathcal{N}x\mathcal{N}$ *and* $l()$ *is a function assigning labels from the set* $\mathcal{L}_N \cup \mathcal{L}_E$*.* $\mathcal{L}_N = \{\{Operation\}, \{Data\}\}$ *is the set of node label classes, holding tensor shapes, data types and arithmetic operations.* $\mathcal{L}_E = \{spatial, sequential\}$ *is the set of edge labels.*

Further, a DFG has the following properties:

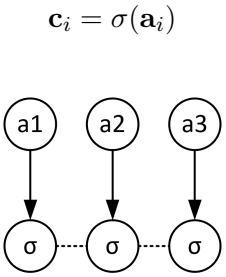
- Nodes only carrying $data$ labels generate outgoing $sequential$ edges for each operation consuming the data. They have no incoming edges. They represent the input values of the computation modelled by the DFG.
- Nodes with $operation$ labels perform a scalar operation, consuming the data of the incoming $sequential$ edges and produce one or more outgoing $sequential$ edges.
- Commutative reduction operations are modelled by a $sequential$ self-edge. This optimization can reduce the number of nodes and edges in a DFG significantly.
- Nodes with $operation$ labels and only an outgoing self-edge, or no outgoing edges, represent the computation result.
- Nodes labelled $operation$ can have bidirectional $spatial$ connections to other nodes, performing the same computation, but for a different output element in the original tensor expression. These connections indicate parallelism in the computation. Potentially, $spatial$ edges lead to a set of k fully connected nodes. The number of edges can be reduced by pruning the connections to create a graph with one internal node and $k-1$ leaves. In this *star* subgraph, the transitive property maintains the parallelism information.

Embedding an intrinsic DFG $\mathcal{G}_i$ into an operator DFG $\mathcal{G}_o$ is the subgraph isomorphism problem:

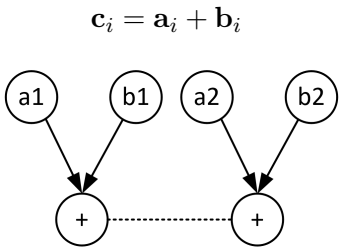
Definition 4.2. *An intrinsic graph $\mathcal{G}_i = (\mathcal{N}_i, \mathcal{E}_i, l_i)$ is an isomorph subgraph of operator graph $\mathcal{G}_o = (\mathcal{N}_o, \mathcal{E}_o, l_o)$ if there exists and injective function $f : \mathcal{G}_i \to \mathcal{G}_o$ where $\forall (s,t) \in \mathcal{E}_i \mapsto (f(s), f(t)) \in \mathcal{E}_o$ while maintaining the node labelling, such that $\forall s \in \mathcal{N}_i : l_i(s) \equiv l_o(f(s))$*

Thus, solving the embedding is to find a mapping between $\mathcal{G}_i$ and $\mathcal{G}_o$ that describes a distinct subset of nodes and edges in $\mathcal{G}_o$ which exactly match $\mathcal{G}_i$.

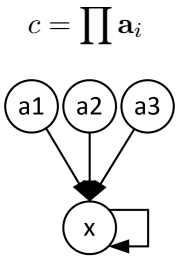
Figure 4.1 shows multiple examples of DFGs created from the respective tensor expression. In the first row Figures 4.1a, 4.1b and 4.1c are examples for tensor expressions performing a single operation, while Figure 4.1d demonstrates how multiple operations are linked together. Figures 4.1a, 4.1b and 4.1e are examples for tensor expression with multiple output elements, while 4.1c and 4.1d show how inputs can be reduced to a single result. Example 4.1e also demonstrates how access patterns from stencils are represented in DFGs. One



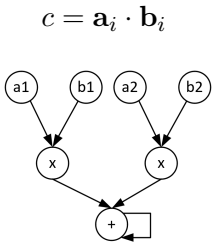
(a) Example DFG of injective functions.



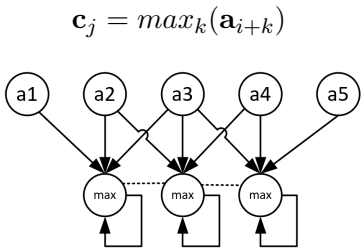
(b) Example DFG of a vector add operation.



(c) Example DFG of a reduction operation.



(d) Example DFG of a dot product. There are no spatial edges, because only a single output element is computed.



(e) Example DFG of a maxpool operation.

Figure 4.1 Examples for dataflow graphs for common operations in DNN operators. The solid lines render the *sequential* edges, the dashed lines render the *spatial* edges. Input nodes are denoted with $a1$, $b1$ and so forth. Operation nodes are described by their respective operation, such as $+, \sigma$ or max.

input element is consumed by multiple operations, which is an example for the *non-functional* relations that can exist between subsets of nodes.

A property of all DFGs derived from tensor expressions is that each subgraph computing one output element shares a set of input nodes with its neighbours but no operations or intermediate results. This property removes the need to cover the full operator DFG with a sequence of instructions, but instead allows solving the problem for a single intrinsic instance and then use further analysis to compute the structure and data layout of the full program. This reduces the embedding problem complexity. For large operators, embedding a comparatively small intrinsic becomes increasingly difficult, because after the first embedding has been found, a decision of where to continue the embedding would be necessary. This would then implicitly define an execution order through the embedding process, and thus, introduce scheduling decisions into the embedding. But for the commutative workloads in DNNs, the number of possible schedules is nearly endless. By extrapolating from a single embedding, the execution order problem

can be handed to a scheduling tool, like AutoTVM. The drawback is the need to specify additional rules what kind of embeddings are possible. More constraints shrink the solution space down further. This is the opposite to iteration domain embedding, where additional transformations increased the search space.

In this approach the matching problem itself is not hindered by implicit implementation decisions like loop ordering, data layout of tensors or access functions. This removes the need for transformations in the search space design to account for these decisions. From the result of a found isomorphism, a new data layout and program structures can be derived. For this new strategy, code can be generated. The exact code generation process is hardware and programming interface specific. The intermediate representation, program structure and data-layout transformation are decoupled from the code generation and individual components can be reused. A demonstration of this is found in Chapter 4.4, where an existing code generation infrastructure is used to program a hardware accelerator with novel strategies.

The presented method is generally not bound to a specific representation of the DFG or algorithm to solve the subgraph isomorphism problem. The following two issues need to be solved in order to make the method applicable in practice:

1. With a space complexity of $O(|\mathcal{N}|^2)$ and $O(|\mathcal{N}| + |\mathcal{E}|)$, adjacency matrices and lists are not suited to represent a full DFG for operators like Conv2D with billions of operations.

2. Solving the embedding only with subgraph isomorphism is no sufficient solution condition. Often, there are additional restrictions to the available hardware and its software interface. A possible implementation needs to be inside this restricted space in order to be valid.

These challenges are addressed in the following sections. Section 4.2 explains how a polyhedral program representation is used to abstract the workload while maintaining the detailed information of the basic dataflow graph. Section 4.3 explains how constraint programming is leveraged to model additional restrictions in the problem.

4.2 Polyhedral Representation of Dataflow Graphs

The first challenge is addressed with a more concise DFG representation, inspired by the polyhedral model, a powerful compiler methodology capable of expressing

computations in quasi-affine loop nests. At its core, it is a mathematical abstraction of the original algorithm, enabling program analysis and manipulation. The polyhedral model was first presented by Karp et. al. in 1967 [51] and is mostly used to optimize numerical programs operating over loop nest. Specialized tools for various domains and problems, such as scheduling [30, 31, 107], vectorization [100, 32, 53] or loop tiling [85, 68, 11] are available. Also, solutions for hardware targets like GPUs [108, 40] or FPGAs [110, 24] have been developed over years. Several tools in the deep learning community leverage this representation to generate efficient DNN kernels. For example, *Tensor Comprehensions* (TC) [102], which is based on [108] uses the polyhedral representation for performance improvements on GPUs and MLIR [57] provides a polyhedral dialect for loop optimization passes.

Polyhedral program representation is an abstraction of loop programs, based on integer sets and relations, modelled with $\mathbb{Z}$-polyhedra. Each nested loop defines the bounds of one polyhedra dimension. Thus, n loops form an n-dimensional $\mathbb{Z}$-polyhedra and each iteration of the loop nest maps to a point in the $\mathbb{Z}$-polyhedra. Loop iterations are represented as an ordered integer n-tuple. Data dependencies between operations and array access functions are modelled as set relations [105].

In the context of this work, sets are described in Presburger notation [105]. Instead of explicitly enumerating every element in a set, the set content is described symbolically through the combination of conditions. For example, the set

$$\mathcal{A} = \{1, 2, 3, 6, 7, 8\} \tag{4.1}$$

will be described as

$$\{\mathcal{A}[i] : 0 < i < 4 \vee 5 < i < 9\}. \tag{4.2}$$

Each n-tuple element in the set, as well as the name of each dimensions, is described by list $[i]$. Since dimensions are identified by their list index, different names do not alter the content of the set. Therefore,

$$\{\mathcal{A}[i, j] : i = 1 \wedge j = 2\} = \{\mathcal{A}[a, b] : a = 1 \wedge b = 2\}. \tag{4.3}$$

Following the colon is the collection of conditions for each of the dimensions in the n-tuple. Parameters for conditions are declared in a list before the set

expression:

$$[x, y] \to \{\mathcal{A}[i] : 0 < i < x \wedge (i\%y) = 0\}. \tag{4.4}$$

So, the parameters $x = 10$ and $y = 2$ create the set

$$\mathcal{A} = \{2, 4, 6, 8\}. \tag{4.5}$$

The order of parameters is irrelevant.

Code snippet 4.2b shows the tensor expression in Figure 4.2a as a loop nest. The statement inside the loop is split into individual two-operand instructions. This is unusual for a polyhedral representation. Typically, the content of a statement is of secondary concern, as most targets are general purpose architectures and the statement can be broken down into individual operations. Here, the split is performed to match the definition of the dataflow graph, where each node is a single operation, like add or multiply. The split exposes the individual dependencies of the scalar multiply and subsequent sum operation in the representation. This order of statements is treated as an additional dimension in polyhedral representation. Each execution of a statement is called *dynamic instance*. All executions of a statement form the dynamic instance set. The sets

$$[I, J, K] \to \{\mathcal{S}_1[i, j, k, t] : 0 \leq i < I \wedge 0 \leq j < J \wedge 0 \leq k < K, t = 0\} \tag{4.6}$$

$$[I, J, K] \to \{\mathcal{S}_2[i, j, k, t] : 0 \leq i < I \wedge 0 \leq j < J \wedge 0 \leq k < K, t = 1\} \tag{4.7}$$

describe the dynamic execution instances of $\mathcal{S}_1$ and $\mathcal{S}_2$ in Figure 4.2b. Basic set operations can be applied to polyhedral sets as well. Thus, all operations in the program are contained in the set

$$\mathcal{S}_1 \cup \mathcal{S}_2 = [I, J, K, T] \to \{\mathcal{S}[i, j, k, t] : 0 \leq i < I \wedge 0 \leq j < J \wedge 0 \leq k < K \wedge 0 \leq t < T\} \tag{4.8}$$

where t is the textual ordering of the two statements, in ascending order. This example shows how the semantics of tensor expressions or loop programs can be transformed into a concise, mathematically sound description that has scalar resolution without enumerating every operation individually.

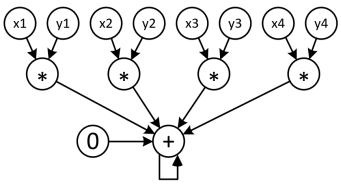
In Program 4.2b, statement $\mathcal{S}_2$ consumes the result of $\mathcal{S}_1$, creating a *read-after-write* (RAW) dependency, denoted as $\mathcal{S}_2 \Rightarrow \mathcal{S}_1$. The instance sets themselves do not contain any order of execution or dependency information. This is modelled by binary relations between the sets, which are also expressed in a symbolic

$T : \mathbf{a}_{i,j} = \sum_k \mathbf{x}_{i,k} \cdot \mathbf{y}_{k,j}$

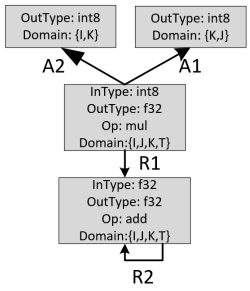
(a) Tensor Expression for a matrix multiplication

```
for i in I:
  for j in J:
    for k in K:
      tmp = x[i,k] · y[k,j] # S1
      a[i,j] += tmp # S2
```

(b) Tensor Expression (a) as a naive loop nest



(c) Partial DFG for a single output element with a contracted add operation



(d) Dependence graph in set and relation based representation

Figure 4.2 A more detailed example how a matrix multiplication, breaks down into several different representations. A loop nest, a part of it's dataflow graph and the used representation with polyhedral sets and relations

notation. The relation $\mathcal{S}_2 \Rightarrow \mathcal{S}_1$ is modelled by

$$\mathcal{R}_1 = [I, J, K] \rightarrow \{\mathcal{S}_2[i, j, k] \Rightarrow \mathcal{S}_1[i', j', k'] : i = i' \wedge j = j' \wedge k = k'\}, \tag{4.9}$$

where the $'$ marks the indices of the right-hand set in the relation ($\mathcal{S}_1$). In an arbitrary instance set $\mathcal{S}$, there is a dataflow between two instances $(s1, s2) \in \mathcal{S}$ if there exists a relation for which $s1 \rightarrow s2 \neq \emptyset$. The second relation in the program is created by the sum operation of $\mathcal{S}_2$. Each dynamic instance reads the value in $\mathcal{A}$, stored by the previous dynamic instance. This creates a cyclic dependence:

$$\mathcal{R}_2 = [I, J, K] \rightarrow \{\mathcal{S}_2[i, j, k] \Rightarrow \mathcal{S}_2[i', j', k'] : i = i' \wedge j = j' \wedge k' = k + 1\}, \tag{4.10}$$

It dictates how instances of the sum operation modelled by $\mathcal{S}_2$ need to be ordered to prevent a violation of the data dependency. Since sums are commutative operation it is possible to relax this constraint to $k \neq k'$, allowing an arbitrary order of $\mathcal{S}_2$ instances.

Memory access functions are modelled in the same way. The example program accesses matrices $\mathcal{X}, \mathcal{Y}$ and $\mathcal{A}$, defined by the parametric sets

$$[I, K] \to \{\mathcal{X}[i, k] : 0 \leq i < I \wedge 0 \leq k < K\} \tag{4.11}$$

$$[K, J] \to \{\mathcal{Y}[k, j] : 0 \leq k < K \wedge 0 \leq j < J\} \tag{4.12}$$

$$[I, J] \to \{\mathcal{A}[i, j] : 0 \leq i < I \wedge 0 \leq j < J\}. \tag{4.13}$$

A read or write operation from a specific dynamic instance to the array is then defined by a relation. In the matrix multiplication example, the read accesses to $\mathcal{X}$ and $\mathcal{Y}$ are described by the following equations:

$$\mathcal{A}_X : \mathcal{S}_1 \Rightarrow \mathcal{X} = [I, K] \to \{\mathcal{S}_1[i, j, k] \Rightarrow \mathcal{X}[i', k'] : i = i' \wedge k = k'\} \tag{4.14}$$

$$\mathcal{A}_Y : \mathcal{S}_1 \Rightarrow \mathcal{Y} = [J, K] \to \{\mathcal{S}_1[i, j, k] \Rightarrow \mathcal{Y}[j', k'] : k = k' \wedge j = j'\} \tag{4.15}$$

$$\mathcal{A}_A : \mathcal{S}_2 \Rightarrow \mathcal{A} = [I, J] \to \{\mathcal{S}_2[i, j, k] \Rightarrow \mathcal{A}[i', j'] : i = i' \wedge j = j'\} \tag{4.16}$$

Notice how not every index from the $\mathcal{S}_1$ or $\mathcal{S}_2$ is used by the access functions. For changes to these unused indices, another access to the same memory element is performed.

With the presented polyhedral sets and relations, any tensor expression can be expressed as a program in polyhedral form by the tuple $\mathcal{P} = \langle \mathcal{S}, \mathcal{D} \rangle$:

- The instance set $\mathcal{S}$ is the union of all dynamic execution instances and tensor elements. $\mathcal{S}$ is described by a set of integer tuples. Each tuple $s \in \mathcal{S}$ describes exactly one dynamic execution instance.
- The data dependence relation $\mathcal{D}$ is the union of all binary relations between pairs of instances in $\mathcal{S}$.

Reflecting this on the DFG representation, the node set $\mathcal{N}_o$ of $\mathcal{G}_o$ is contained in $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{X} \cup \mathcal{Y} \cup \mathcal{A}$, while the *sequential* edge set $\mathcal{E}_o$ is contained in $\mathcal{D} = \mathcal{A}_X \cup \mathcal{A}_Y \cup \mathcal{A}_A \cup \mathcal{R}_1 \cup \mathcal{R}_2$. Not all the relations in this representation are *symmetric*, which means they are the same in both directions. The relation $\mathcal{S}_1 \to \mathcal{X}$ is *surjective* and *functional*, meaning that every multiplication can exactly map to the one tensor element in $\mathcal{X}$ it consumes. However, the inverse relation $\mathcal{X} \to \mathcal{S}_1$ is *non-functional*. Every input element in $\mathcal{X}$ is used by multiple multiplications, but not all of them. The relation of $\mathcal{A}_X$ and $\mathcal{A}_Y$ reflects this, as they contain no term with properties for i' or j', respectively. As a result, for one specific input value in $\mathcal{X}$, the relation describes the subset of all multiplications using this value.

All this information is stored in a structured way by a coarse-grained *dependence graph*, as displayed in Figure 4.2d. A node is an expression of the original tensor

algorithm, like an arithmetic operation or a memory read. Each dependence graph node contains the polyhedral description for all DFG nodes with the same label as the dependence graph node. Edges describe the access and dependency relations between the individual sets. Such graphs are a known concept in polyhedral representations [9]

Unlike graph representations based on adjacency matrices or lists, the representation is of constant size and does not scale with number of nodes in the graph, but only with number of dimensions and statements in the original tensor expressions. Further, it enables a detailed analysis of the program, among others:

- Determine the input scope of specific output elements i.e., computing the bounds of array accesses for a single or set of output elements. In a graph representation this would require expensive graph traversal. In the polyhedral analysis this can be achieved with a set operation sequence.
- Effective manipulation of the individual sets and relations with basic set operations, for example to remove elements not interesting for a given analysis with an intersection.
- Computing convex hull shapes for tensor slicing and nesting. This enables the deconstruction of high-dimensional computations into regular, lower-dimensional convex hulls. Through piece-wise affine constructs, regularly strided input sets can be modelled in detail.

While the polyhedral model is a powerful abstraction, some restrictions apply. Foremost, the representation is aimed at loop programs with bounds known, or computable, at compile time. Further, bounds and relations need to be expressible as quasi-affine functions, only depending on iteration variables and constant values [106]. Data-dependent accesses and branches are not supported. However, none of these restrictions pose a problem for the workloads targeted in this work. While other methodologies with similar capabilities exist, they are often less detailed, like interval analysis [66, 38], or require more extensive correctness proofs [73].

4.3 Embedding as a Constraint Satisfaction Problem

Based on the concise yet detailed program representation from Section 4.2, this section will now discuss how constraint programming (CP) solves the second challenge presented in Section 4.1. CP is a method of solving constraint

satisfaction problems (CSPs). By describing the space of legal embeddings in terms of constraints, a solver can automatically generate solutions belonging to this space, if any exist. Modifying the constraint setup also allows different degrees of exploration in a solution space. For example, at first only solutions that require a few, simple transformations are allowed by using a tight constraint setup. If this fails, a different constraint setup allows solutions that require more expensive transformations. The choice of CP over techniques like SAT is motivated by its expressiveness in the program formulation and customizable propagation and search algorithms.

Definition 4.3. *A CSP is formally defined as triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where*

- $\mathcal{X} = \{x_j | 0 \leq j \leq n\}$ *is a set of variables, for which a value needs to be found.*
- $\mathcal{D} = \{d_j | 0 \leq j \leq n\}$ *is the set of value domains, from which a value is assigned to the respective variables. An assignment $Asn(d_j, x_j) : x_j = v$ selects value $v \in d_j$ for x_j to take. Variable x_j can only ever receive values from domain d_j, not from any other domain in $\mathcal{D}$.*
- $\mathcal{C} = \{c_i | 0 \leq i \leq m\}$ *is the set of constraints. A constraint c_i is formed over a subset of variables $g_x \subset \mathcal{X}$ and evaluates if all assignments $Asn(g_d, g_x)$ with $g_d \subset \mathcal{D}$ are valid.*

The CSP is satisfied when all assignments are performed, and no assignments violates the conjunction of $\mathcal{C}$.

Once the problem is modelled with variables, domains and constraints, a solver begins the process of assigning values and evaluating the constraints. Evaluating every possible assignment is infeasible in most practical applications. Instead, every CP constraint comes with a propagator, which removes values from the domain that cannot be part of a valid solution. The propagator is a monotonic filtering algorithm: it only removes values from a domain, but never adds any. It is specific for each constraint. The propagator infers which values to remove from a domain, based on the domains and assignments of other variables under the same constraint. To find a solution, the solver uses a search algorithm to systematically perform assignments and propagate the assignments through the domains. A backtracking-based search algorithm can find all possible solutions in a given problem. Variable selection determines which $x \in \mathcal{X}$ is assigned a value next. Value selection is the specific implementation of $Asn(d, x)$. Variable and value selection impact the time-to-solution and need careful consideration when designing the constraint program.

4.3.1 Embedding Problem Definition

This section will discuss how to represent the embedding as a CSP in $GeCode$[1], the solver used for this work.

Definition 4.4. *The full space of the embedding problem is:*

- $\mathcal{X} = \{x | \forall x \in \mathcal{N}_i\}$ *For every node of the instruction DFG a variable is created. This means every scalar operation and data element in the instruction is represented by a variable.*
- *The set of domains is defined as* $\mathcal{D} = \{d | d \subseteq \mathcal{S}_{operator}\}$. *Every domain is a subset of the operators instance set in the polyhedral representation. The subset is determined by the node type. The domains of nodes labelled* $data$ *are the polyhedra bounded by the respective tensor shapes. The domain of nodes labelled* $operation$ *is the instance set.*

Definition 4.4 describes an embedding on the scalar level. Since every potential assignment between nodes in the instruction and nodes in the workload is described by this formulation, it also contains every possible solution to the embedding problem between this specific workload and instruction.

Constraints specifying solution properties can be imposed over this space. The most important constraint is matching the dataflow between operator and instruction. Additional constraints further narrow the solution space.

The next sections present the constraints imposed upon the problem space. First an introduction to the used general-purpose constraints, mainly graph isomorphism and hyper-rectangle detection. Section 4.3.5 focusses on more specialized constraints, modelling memory access behaviour.

4.3.2 Subgraph Isomorphism

Solving the subgraph isomorphism problem with CP is a well-researched problem [119, 120, 56] and solutions range from direct formulations to highly optimized implementations. This work directly models of the instruction DFG $\mathcal{G}_i$ we want to discover in the operator's target graph $\mathcal{G}_o$, as shown in Figure 4.3a. As described in Definition 4.4, every node of $\mathcal{G}_i$ is a variable. Every edge $(s, t) \in \mathcal{E}_i$ is then modelled with a pairwise constraint. The combination of all pairwise constraints describes the dataflow (line 3). For better propagation, the spatial edges (line 5) are modelled as well. To fully express isomorphism, a global *AllDiff* constraint

[1] https://github.com/Gecode/gecode

```
1 for (s,t) in E_i:
2   if label(s,t) ∈ sequential:
3     edge(s,t, etype ⇐ sequential)
4   else:
5     edge(s,t, etype ⇐ spatial)
6
7 AllDiff(N_i)
```

(a) Describing the instruction graph as pairwise $edge$ constraints.

```
1  if l(s) ≠ l(s.val)
2    return Failed
3  # evaluate data dependence
4  rel ⇐ eval(l(s)→l(t), s.val)
5  if rel = ∅:
6    return Failed
7  t.domain ⇐ t.domain ∩ rel
8
9  if t.size = 1:
10   t.val ⇐ t[0] # assign solution
11   return Finished
12 return # value of t selected by
       solver
```

(b) Propagation of edge constraints using the data dependence relation

Figure 4.3 Algorithms for describing subgraph isomorphism and computing the propagation based on the polyhedral data dependence relations

is used, such that every node can only occur once in the solution (line 7). The *AllDiff* constraint enforces that each value assigned to the variables under the constraint must be pairwise different.

Now that the problem is described, the solving process can begin. During this, the solver eventually assigns a variable a value from its domain. Assigning a value means to select one node of $\mathcal{N}_o$ as a possible candidate to match a node in $\mathcal{N}_i$. The propagation algorithm in Figure 4.3b then checks the label of the variable against the node assigned from the domain (lines 1-2) and if possible, also filters the other node's domain (lines 4-9). The propagator is filtering values directly based on the data dependence relations in the dependence graph. It evaluates the relation (line 4) and removes values from the partner node's domain, where no connection exists (line 7). If the relation between the pair is *functional*, it can directly assign a solution (line 9-11). Even if this is not the case, the propagation is powerful enough to *subsume* the domain. This means that only valid solutions for this constraint remain in the domains and no further propagation is necessary. The remaining domain values are evaluated with respect to the other constraints over their variable. If evaluating the relation leads to an empty domain, the assignment fails (line 6). Finally, when values are assigned to both s and t, the constraint checks for correctness by verifying there is an edge in $\mathcal{G}_o$ connecting the pair, or formally $Asn((s,t),(d_s,d_t)) \in \mathcal{E}_o$.

To further aid the propagation, the constraint also models the parallel edges in the dataflow graph (line 5 in Figure 4.3a). Since fully expressing this would result in a large number constraints, the transitive property of pairwise constraints is leveraged by picking an arbitrary node in the DFG and adding a constraint to

```
1  m_k ⇐ v_1 − v_0 # base step for every move in the hyper rectangle
2  if m_k/|m_k| ∉ V_B: return Failed # see if base step is axis aligned
3  V_B ⇐ V_B ∩ m_k/|m_k| # remove m_k from vector base set V_B. Each direction is only
       allowed once.
4
5  V ⇐ V ∩ v_0 # remove first element from list
6  # Initialze counter tables
7  dimTable ⇐ ∅ # table keeping track of already determined dimenision sizes
8  liveTable[m_k] ⇐ 1 # table with active counters
9  strideTable[m_k] ⇐ |m_k| # stride along each axis
10 for v_n ∈ V:
11   move ⇐ v_n − v_{n−1} # compute current step
12   if move ∈ liveTable:
13     # increment table counters in liveTable
14     liveTable[move] ⇐ liveTable[move] + 1
15     # verify counter states in dimTable and reset them, if necessary
16     verify(dimTable, liveTable)
17   # check if this is a dimension jump
18   else if (v_n−v_0)/|v_n−v_0| ∈ V_B ∧ move = (v_n − v_0) + (v_0 − v_{n−1}):
19     # Add the current outermost dimension size to the dimTable
20     dimTable[m_k] ⇐ LiveTable[m_k] # size of d_{k−1}
21     # Add new counter for new outermost dimension d_k
22     liveTable[move] ⇐ 1
23     # track latest diagonal move as
24     m_k ⇐ move
25     # remove the directional vector from the vector base V_B
26     V_B ⇐ V_B ∩ (v_n−v_0)/|v_n−v_0|
27   else:
28      return Failure # if the move is not a jump or already known
29   Store latest m_k in dimtable
30     dimTable[m_k] ⇐ liveTable[m_k]
31 bounding_box ⇐ bbox(DimTable) # compute bounding box from dim table
32 return bounding_box
```

Figure 4.4 Algorithm for hyper rectangle inference. $\mathcal{V}$ is the list of variables, each variable holding a point and $\mathcal{V}_B$ is the vector base of the domain's shape (e.g. shape of the input tensor).

every parallel node it has. If now any node parallel to the first node gets assigned a value, the domain of first node is pruned to only contain nodes parallel to the assignee. This, in turn, propagates to all other nodes parallel to the first node. While this introduces a degree of indirection in the propagation, the number of pruned values remains the same.

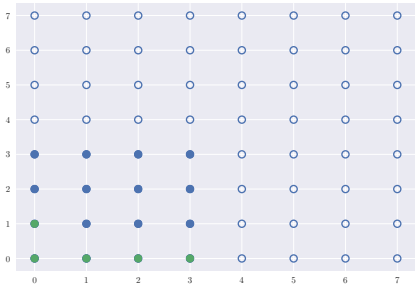
4.3.3 Axis-Aligned Hyper-Rectangle Constraint

For this work, but also for DNN workloads and accelerators in general, regular memory access patterns are a sensible restriction on the solution space. Many DNN workloads operate in high-dimensional, rectangular spaces, also called hyper-rectangles. Selecting an axis-parallel subset of this domain enables common data-layout transformations, like transposing, fusing or tiling. At the same time, this helps reducing the size of the solution space. This is enabled by a constraint to detect, verify and propagate the shape of a rectangle with any number of

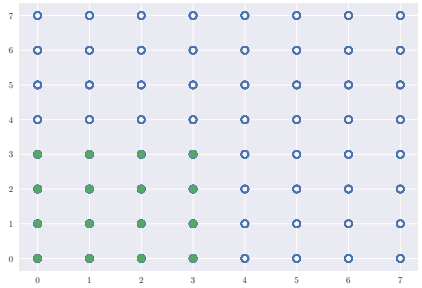
(a) Assignment of first and second variables

(b) Assignment third variable

(c) Assignment of fourth variable



(d) Assignment of fifth variable and propagation of constraint



(e) Full assignment of all variables after propagation

Figure 4.5 The plots above show how propagation and assignment happens in the rectangle constraint. Blue dots represent domain values, green dots assigned variables and white dots with blue outlines are ones the values removed by propagation.

dimensions $n > 0$ from an ordered tuple of points $\mathcal{V} = [v_0, ..., v_m]$. After only a few decisions, the propagator can infer a bounding box from the selected points and the total number of points in $\mathcal{V}$. Intersecting this bounding box with the domain efficiently removes values that can never lie within the rectangle. The constraint supports regular strides in any dimension.

Algorithm 4.4 works by iterating over the variables, tabulating the distance between two elements in the $strideTable$, the size of a dimension in $dimTable$ and tracks the current location in a $liveTable$. Using the values of these counters, the algorithm determines if $\mathcal{V}$ forms an axis aligned rectangle with regular strides.

The following example explains the process shown in Figure 4.5. The blue points represent the domain in two-dimensional tensor $\mathbf{a}_{y,x}$ with shape $s_{\mathcal{A}} = (X, Y)$, represented as the following set:

$$[X, Y] \to \{\mathcal{A}[y, x] : 0 \leq x < X \wedge 0 \leq y < Y\} \tag{4.17}$$

During a search, the constraint solver assigns values to one of $M = 16$ variables, which are represented by the green points. Each step in the example

corresponds to one assignment of a point in the domain to a variable and the resulting call of the hyper-rectangle constraint.

Assignments 1,2 Corresponding to Figure 4.5a: The initial value assignments are $v_0 = \mathbf{a}_{0,0}$ and $v_1 = \mathbf{a}_{0,1}$. Lines 1-9 in Algorithm 4.4 determine the innermost dimension of the rectangle and its stride. The innermost *step* of the hyper-rectangle search is $m_k = v_1 - v_0 = [0, 1]^T$ and the stride is $stride_x = |v_1 - v_0| = 1$. Since no further variables are assigned, the algorithm is done and skips to line 28. The bounding box in this two-dimensional space is set to $[Y, M \cdot stride_x]$. The values for the x-axis can be bounded to the number of variables M, multiplied by the stride between the first two elements, since the stride for every dimension needs to be regular. The constraint now removes values from the search domain according to this new bound:

$$[Y, M, stride_0] \rightarrow \{\mathcal{A}[y, x] : 0 \leq x < M \cdot stride_x \wedge 0 \leq y < Y\} \tag{4.18}$$

Assignment 3 Corresponding to Figure 4.5b: Assignment $v_2 = \mathbf{a}_{0,2}$ is checked if the step from v_1 to v_2 matches the initial step v_0 to v_1. This happens in lines 10-17 of the Alg. 4.4, where the counter tables keep track of the steps in each dimension. After computing the bounding box, no further possibilities for propagation present themselves here. The next selections could either continue along the x-axis which has already been constrained or skip to the y-axis with any stride. Thus, no more values can be pruned safely.

Assignment 4 Corresponding to Figure 4.5c: Assignments $v_3 = \mathbf{a}_{0,3}$ is checked if the step from, v_2 to v_3 matches the initial step v_0 to v_1. Again, the bounding box offers no new opportunity for propagation.

If an element from another dimension had been selected, the algorithm would have failed as it would have meant a side of length 3 in the rectangle. Since 3 is no even divisor 16, a legal rectangle could not have been formed.

Assignment 5 Corresponding to Figure 4.5d: The next significant step happens with assignment $v_4 = \mathbf{a}_{1,0}$. Moving from v_3 to v_4 is not a step along the innermost dimension, but a *jump* adding a second dimension in the rectangle. A jump is a diagonal move between the last element in previous dimension and the first element in the following one. In this case, from $\mathbf{a}_{0,3}$ to $\mathbf{a}_{1,0}$. Whether a move is actually a jump is determined by computing if it is the hypotenuse

of a triangle formed by v_0, v_{n-1} and v_n:

$$move = (v_n - v_0) + (v_0 - v_{n-1}) \tag{4.19}$$

Further, we need to ensure the *jump* does not violate the axis alignment by checking if its normal vector is part of the vector base $\mathcal{V}_B$:

$$\frac{v_n - v_0}{|v_n - v_0|} \in \mathcal{V}_B \tag{4.20}$$

For the bounding box, the following new pieces of information can be inferred: First, for the solution to remain a valid rectangle, the innermost dimension needs to always have $d_x = 4$ elements. Second, the y dimension of $\mathbf{a}_{y,x}$ with $stride_y = |v_n - v_0|$ can be bounded to $\frac{M}{d_4} \cdot stride_y$. Together this leads to a two-dimensional bounding box

$$[M, d_x, stride_y] \rightarrow \{\mathcal{A}[y, x] : 0 \leq x < d_x \wedge 0 \leq y < \frac{M}{d_x} \cdot stride_y\} \tag{4.21}$$

Since $|\mathcal{A}| = 16$ and steps and jumps recorded so far dictate the ordering, only one legal variable-value selection remains, meaning the constraint is resolved after only five steps. See Figure 4.5e.

For an abitrary number of k dimensions, the verification in line 13 checks if for a *jump* into dimension d_k, all counters of the inner dimensions $d_{k-1}...d_0$ are zero. If one counter is non-zero, the *jump* breaks the regular structure of the hyper-rectangle.

The sequential assignment can lead to situations where multiple valid assignments are performed, but a full assignment would not result in a correct hyper-rectangle. Correctness can be assured early-on by computing the integer factorization of M and see if all dimensions fit one of the solutions. For example, with $M = 12$ dimensions of size 2 and 4 would be legal by themselves, however, they can never appear together.

The example mentioned bounding box computations multiple times. The process to compute bounding box shape s_{box} is detailed in Equation 4.22.

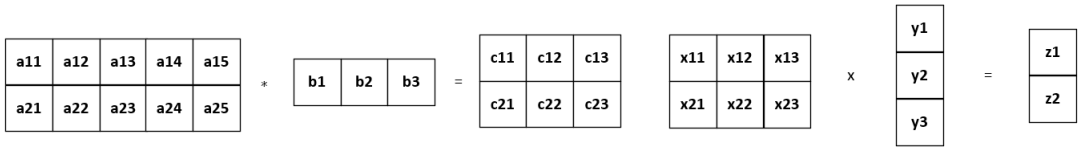
$$s_{box} = \sum_{\mathcal{V}_B}^{\mathbf{b}} \mathbf{b} \cdot \begin{cases} strideTable[\mathbf{b}] \cdot dimTable[\mathbf{b}] & \text{if } \mathbf{b} \in dimTable \\ strideTable[\mathbf{b}] \cdot \frac{M}{\prod dimTable} & \text{else if } \mathbf{b} \in liveTable \\ domain[\mathbf{b}] & \text{else} \end{cases} \tag{4.22}$$

$\mathbf{b}_{h,w} = \sum_0^{kw} \mathbf{a}_{h,w+kw} \cdot \mathbf{b}_{k,w}$

(a) Stencil Workload

$\mathbf{z}_i = \sum_0^{k} \mathbf{x}_{i,k} \cdot \mathbf{y}_k$

(b) GEMV intrinsic



(c) Graphical representation of the stencil workload and the GEMV instruction.

Each base vector $\mathbf{b}$ is multiplied by the smallest possible domain extend. The sum of these vectors is the shape of the bounding box. For example, in Assignment 5 of the above example:

$$s_{box} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \cdot d_y + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \cdot d_x = \begin{pmatrix} 4 \\ 4 \end{pmatrix} \tag{4.23}$$

If the size of a dimension is known, meaning Algorithm 4.4 already found a jump to an outer dimension, the size is multiplied by the stride. If a dimension's size is not yet known, but it is known to be a part of the rectangle, the largest still possible extend is multiplied with the already known stride. Otherwise, the original domain size is used.

4.3.4 Padding

To support a wider range of workloads, zero-padding can be added to individual dimensions of the iteration domain. This happens when either a) an individual dimension is below a specified threshold or b) the dimension is not an even divisor of dimensions in the hardware intrinsic. This gives the CSP solver more freedom in terms of the search and possible candidates. By penalizing padding in the later candidate selection, solutions with unnecessary padding are discarded early.

4.3.5 Memory Access Constraints

Subgraph isomorphism alone is not sufficient to ensure computational correctness. Memory access patterns that match for each individual output element of a workload can be unsuitable for code generation. This creates the necessity for constraints that ensure correct output element selection, given the graph isomorphism present in the instruction. This can be resolved by analysing the properties of accessed memory elements between the workload and the hardware

intrinsic. By comparing how the accessed data overlaps between the computation of two output elements, it is possible to determine whether a mapping holds, or not. To do this, the concepts of *access sets* and *overlap sets* are introduced.

Definition 4.5. *An access set $\mathcal{A}_x^{z_1}$ is defined as the union of data elements from input tensor $\mathbf{x}$ necessary to compute the output element $\mathbf{z}_1$.*

Definition 4.6. *An overlap set is defined as union of pairwise intersections*

$$\mathcal{O}_{\mathcal{N}}^{\mathbf{z}} = \bigcup_{n=1}^{N} \mathcal{A}_{\mathbf{x}}^{\mathbf{z}_{n-1}} \cap \mathcal{A}_{\mathbf{x}}^{\mathbf{z}_n}$$

over a collection of access sets $\mathcal{A}_{\mathbf{x}}^{\mathbf{z}_n}$ over indices in $\mathcal{N}$.

For tensor expressions, the access sets and resulting overlap sets can be computed conveniently through the polyhedral representation. Each tensor expression output element location also determines the loop iterators for the non-reducing loops. And to compute a full output element value, every reduction-loop needs to be fully iterated. Thus, the access set for one output element is determined by a polyhedral set containing all input elements. The example in Figure 4.6c illustrates this when matching a GEMV instruction 4.6b to a stencil workload 4.6a. The workload accesses three elements to compute one output element. For the next output, two of the inputs overlap with the one from the previous output element while one is new. For simplicity, the example focusses on the relation between $\mathbf{x_{i,k}}$ and $\mathbf{a_{h,w}}$. For the workload, all access sets are described the function

$$\mathcal{A}_{\mathbf{a_{h',w'}}}^{\mathbf{b}_{h,w}} = [h, w, K] \rightarrow \{\mathbf{a}[h', w'] : h' = h \wedge w' = w + k \wedge k = 0 < k < K\}. \tag{4.24}$$

When the mapping function $f()$ behaves such as $f(z_1) = b_{1,1}$ and $f(z_2) = b_{1,2}$, the workload access set is

$$\mathcal{A}_{\mathbf{a}_{h',w'}}^{\mathbf{b}_{1,1}} = \{\mathbf{a}_{1,1}, \mathbf{a}_{1,2}, \mathbf{a}_{1,3}\} \tag{4.25}$$

$$\mathcal{A}_{\mathbf{a}_{h',w'}}^{\mathbf{b}_{1,2}} = \{\mathbf{a}_{1,2}, \mathbf{a}_{1,3}, \mathbf{a}_{1,4}\} \tag{4.26}$$

and the resulting overlap set is

$$\mathcal{O}_{h=1,w=1..2}^{a} = \{\mathbf{a}_{1,2}, \mathbf{a}_{1,3}\} \tag{4.27}$$

in our example. Using the following function for the instruction,

$$\mathcal{W}_{\mathbf{x}_{i',k}}^{\mathbf{z}_{i,j}} = [i, K] \rightarrow \{\mathbf{x}[i', k] : i' = i \wedge k = 0 < k < K\}. \tag{4.28}$$

```
for i in z_i:
  no_overlap(x_{i,k}, z_0, z_i)
  no_overlap(x_{i,k}, z_i, z_{i+1})
  full_overlap(y_i, z_0, z_i)
  full_overlap(y_k, z_i, z_{i+1})
```

Figure 4.7 Defining the memory access behaviour through pairs of overlap set rules

the access sets are

$$\mathcal{A}^{\mathbf{z}_{1,1}}_{\mathbf{x}_{i',k}} = \{\mathbf{x}_{1,1}, \mathbf{x}_{1,2}, \mathbf{x}_{1,3}\} \tag{4.29}$$

$$\mathcal{A}^{\mathbf{z}_{1,2}}_{\mathbf{x}_{i',k}} = \{\mathbf{x}_{2,1}, \mathbf{x}_{2,2}, \mathbf{x}_{2,3}\} \tag{4.30}$$

with an intersection of

$$\mathbf{O}^{z}_{i=1..2,j=1} = \emptyset. \tag{4.31}$$

This makes the assignment $f(z_1, z_2) = (\mathbf{b}_{1,1}, \mathbf{b}_{1,2})$ infeasible, at least without memory layout transformations. The alternative assignment $f(z_1, z_2) = (\mathbf{b}_{1,1}, \mathbf{b}_{2,1})$ is correct, since the access sets

$$\mathcal{A}^{\mathbf{b}_{1,1}}_{\mathbf{a}_{h',w'}} = \{\mathbf{a}_{1,1}, \mathbf{a}_{1,2}, \mathbf{a}_{1,3}\} \tag{4.32}$$

$$\mathcal{A}^{\mathbf{b}_{2,1}}_{\mathbf{a}_{h',w'}} = \{\mathbf{a}_{2,1}, \mathbf{a}_{2,2}, \mathbf{a}_{2,3}\} \tag{4.33}$$

create the overlap set,

$$\mathcal{O}^{b}_{h=1..2,w=1} = \emptyset. \tag{4.34}$$

The assignments for the right-hand side operands $\mathbf{y}_k$ and $\mathbf{w}_{kw}$ are valid for either choice, since they do not change for different output elements.

With these concepts, the polyhedral models helps creating pairwise, constraints describing memory access behaviour between pairs of inputs for the hardware intrinsics. Generalizing the example above for arbitrary large GEMV instructions leads to a chain of pairwise constraints, similar to the used subgraph isomorphism constraint. In Figure 4.7, `no_overlap` describes an empty overlap set and `full_overlap` an identical overlap set. By posting another constraint from the first to any other input element in the same column takes care of the non-transitive property of the constraints.

Similar to the hyper-rectangle constraint from Section 4.3.3, the formulation is invariant to the dimension ordering of the domain. In a scenario where the sliding window w would be applied in vertical, instead of horizontal direction of

the workload, the posted constraints would still produce the correct behaviour for this hardware intrinsic.

4.3.6 Candidate Selection

Padding and other data transformations necessary for an embedding can create an overhead in the number of multiply-accumulate operations (MAC) and data movement. To handle this, an optimization term to the constraint problem is added to the CSP. It minimizes the MAC and data movement overhead O_{MAC} and O_{Data}, compared to the theoretical minimum:

$$O_{MAC} = MAC_{Total} - MAC_{min}$$
$$O_{Data} = Data\ movement_{Total} - Data\ movement_{min}$$

When treating O_{MAC} and O_{Data} as elements of a vector $\mathbf{o}_n = [O_{MAC}, O_{Data}]^T$ with $n = 2$, the CSP optimizes the term $min(||\mathbf{o}_n \cdot \mathbf{w}_n||)$, where $\mathbf{w}_n$ is a user-defined weight vector. It can be used to shift the impact of the overheads on the candidate selection, depending on the used hardware and technology. This eliminates solutions with padding or other overheads, when possible.

4.4 Experiment Setup

This section describes the experimental setup to evaluate the method described in the previous sections. Section 4.4.1 contains an overview of the used accelerator hardware, followed by Section 4.4.2 describing how the original TVM compilation flow was modified to integrate the methods introduced in this chapter. After this, the target-specific parts of the method are presented. A detailed description of the constraint setup is given by Section 4.4.3 and the rule-based code generation is described in Section 4.4.4.

4.4.1 Hardware Setup

The evaluation is performed on the *Versatile Tensor Accelerator* (VTA) [67], a hardware accelerator providing a matrix-multiply (GEMM) intrinsic, as well as a vector unit for activation and scaling tasks. The hardware is instantiated on a Zynq UltraScale+ FPGA. It uses the default configuration provided by the authors with a 256kbyte weight buffer, a 128kbyte data buffer and a GEMM core with $8bit$ multiplications and $32bit$ accumulation. The GEMM unit computes

$\mathbf{c_{x,y}} = \sum_z \mathbf{a_{x,z}} \cdot \mathbf{b_{z,y}}^T$ with $(x, y, z) = (1, 16, 16)$ and its result can be processed by a vector unit for element wise activation and quantization operations. Notice that matrix operand $\mathbf{b_{zy}}$ is transposed. The hardware has a load/store direct-memory access (DMA) unit for independent memory accesses of matrix operands. It can read and write full 2D operand matrices stored consecutively in memory. The architecture and programming model follows the Decoupled-Access-Execute model [90]. Other than traditional von-Neumann architectures, data access and operations are handled by independent hardware units. In the case of VTA, interactions between the two are handled by a queue system. The whole process is software-managed i.e., a full program consists of load, store, execute and queue management operations. This principle simplifies the hardware while shifting complexity into software.

4.4.2 TVM Compilation Flow Extension

As shown by Figure 4.9a, code generation for VTA is handled by TVM and interfacing on the relay level, as well as the algorithm and schedule level of each offloadable operator. First, the relay graph representing the DNN is manipulated to perform additional data transformations on both the parameters and the activations of the DNN to accustom deployment to VTA. For each operator a specialized schedule template is provided, where loop transformations and *tensorization* are performed, replacing part of the loop nest with a specialized hardware intrinsic.

The *Conv2D* reference embedding of TVM maps the three axes x, y, z of the GEMM unit statically to the batch $n = x$, output channel $oc = y$ and input channel $ic = z$ dimensions of the convolution. Each of these dimensions is split and moved to be the innermost dimensions of the *input*, *weight* and *out* tensors. This process is specified in a static template, applying the necessary transformations during code generation. Figure 4.8 shows the loop and data-layout transformations for a convolution with a 2D memory tiling, where each tile is a matrix operand that can be loaded/stored by the DMA. Step 1 is the baseline *Conv2D*. The loops and tensor are tiled as explained before (Figure 4.8, step 2). The resulting implementation reorders the new loops n_i, oc_i, ic_i to be the innermost parts of the loop nest. These loops are then replaced by a call to the GEMM instruction. The DMA load and store operations are then placed around the operation to continuously load values until an output segment is complete (Figure 4.8 step 3). Implementing other workloads follows a similar

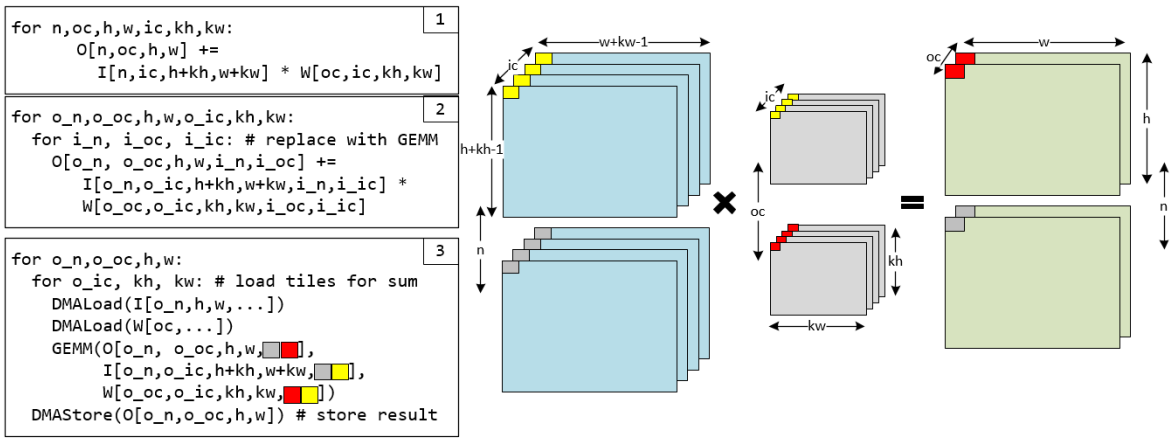
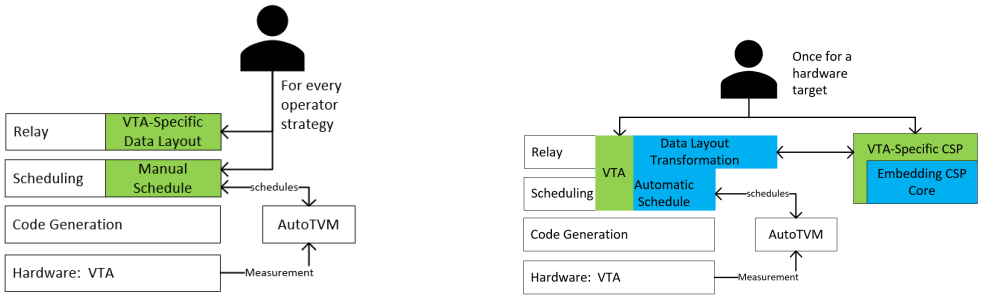


Figure 4.8 2D Convolution and the reference GEMM implementation



(a) Existing VTA compilation stack.

(b) Extended VTA compilation stack.

Figure 4.9 Original and modified deployment stack of VTA accelerator.

pattern; input and output dimensions are tiled into matrices and moved to the inside. In the original TVM version (see Figure 4.9a), every transformation necessary for deployment is determined by a human expert. Strategy variations require new relay transformations, as well as new schedule implementations, each specifying layout transformations, loop orderings and tensorization. As of this date, only *Conv2D* in $NCHW$ layout with no dilation, *Conv2D*-Transposed as well as *dense*-like operations are supported.

To utilize the VTA accelerator in the experiments, its compilation flow is extended with the dataflow embedding. This allows reuse of all software components not affected by the embedding, as seen in Figure 4.9b. Mainly, the auto-tuning and JIT code generation facilities can be used as before. In relay, the restrictions for specific operators and their memory layout are lifted. Instead, operators with multiply-accumulate operations are marked as potential candidates for an embedding. After a successful embedding, a deployment strategy is extracted from the result. This strategy is then used to a) generate

the appropriate data layout rewrites and b) generate a schedule/kernel for auto-tuning and code generation. Sections 4.4.3 and 4.4.4 provide details on how the solver finds embeddings for VTA and how the code is generated.

4.4.3 Constraint Program for VTA

The space of legal embeddings for a hardware target is defined by a specific combination of constraints. Important to note is that each program is constructed from constraints, which are by themselves hardware independent. Conceptually this comparable to auto-tuning tools, where different combinations of elemental rewrite operations can form many different implementations of the same program.

The instruction DFG $\mathcal{G}_i$ is generated from the hardware configuration. Currently, this is still manual step. From there, a constraint program as explained in Section 4.3 is generated. The nodes of $\mathcal{G}_i$ become the variables and the workload's dynamic instance sets the domain. For VTA, the following constraints are used to generate mappings:

Subgraph Isomorphism: Match the dataflow of the GEMM instruction. This is the central constraint of the embedding problem. It is used as described in Section 4.3.

Disjoint/AllDiff: Prevent the same dynamic execution instance or input value from appearing multiple times in the same instruction call. This constraint is imposed over all variables in the problem. To allow overlapping access patterns, this can be relaxed for input data.

Hyper-Rectangle: Ensure all input and output elements are mapped into an axis aligned shape. For VTA's inputs $\mathbf{a}_{x,z}, \mathbf{b}_{y,z}$ and $\mathbf{c}_{x,y}$, this is applied to the constraint variables subsets $\mathcal{N}_i^{\mathbf{a}_{x,z}}$, $\mathcal{N}_i^{\mathbf{b}_{y,z}}$ and $\mathcal{N}_i^{\mathbf{c}_{x,y}} \subset \mathcal{N}_i$. The sets contain the variables modelling the input and output nodes of $\mathcal{G}_i$. This allows simpler memory transformations based on transpose and reshape operations. The constraint is posted for input and output tensors. This constraint allows additional control over a choice of embedding parameters. These parameters are part of the constraint program's input and different solving strategies and workloads can vary these parameters. The following parameters are adjustable:

- *stride*: How much stride is allowed in each workload dimension. Depending on the available code generation, different strides can be possible.

- *dimensions*: To how many different dimensions in the workload is the instruction allowed to match. With this, the fusion of dimensions can be controlled in the code generation.

Fixed Origin : The first match of all input and output tensors is fixed to the origin of the respective domain. This helps in structuring the problem and reducing the search effort and has no direct influence on the correctness.

Memory Access Only allow matches with the same memory consumption pattern as the instruction. These concepts are explained in Section 4.3.5. VTA has the following overlap set definitions

$$\forall i,j \in \mathbf{c}_{x,y} : \mathcal{O}^{\mathbf{a}_{x,z}}_{[i,j],[i,j+1]} = \emptyset \tag{4.35}$$

$$\forall i,j \in \mathbf{c}_{x,y} : \mathcal{O}^{\mathbf{b}_{y,z}}_{[i,j],[i+1,j]} = \emptyset \tag{4.36}$$

to prevent overlapping input sets for the rows in matrices $\mathbf{a_{xz}}$ and $\mathbf{b_{yz}}^T$ and

$$\forall i,j \in \mathbf{c}_{x,y} : \mathcal{O}^{\mathbf{a}_{x,z}}_{[i,j]} = \mathcal{O}^{\mathbf{a}_{x,z}}_{[i,j+1]} \tag{4.37}$$

$$\forall i,j \in \mathbf{c}_{x,y} : \mathcal{O}^{\mathbf{b}_{y,z}}_{[i,j]} = \mathcal{O}^{\mathbf{b}_{y,z}}_{[i+1,j]} \tag{4.38}$$

for identical inputs between the columns.

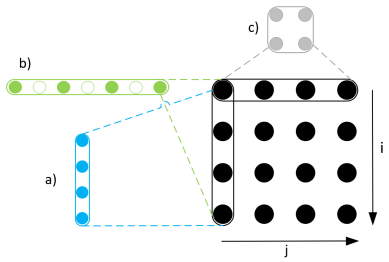
The candidate selection uses the weight vector $\mathbf{w}_2 = [1,1]^T$ in the experiments. However, future work can investigate these hyper-parameters further, especially when optimization of specific metrics is the goal. This constraint program and search strategy can be used to embed the VTA instruction into any workload, not just convolutions. This removes specific program transformations from the search space, that would only be applicable to certain workloads. Instead, it formulates a solution space within the constraints of both the hardware and the code generation facilities. Parametrization of the solution space, like stride in the hyper-rectangle, enables control over the exploration degree in the search. For example to limit the solutions to instances the following code generation steps can handle gracefully.

4.4.4 Customized Code Generation

Code generation in the polyhedral model is usually a complex process, generating an abstract syntax tree (AST) and then lowering this AST to a language the

```
1   diagonal = v_n - v_0
2   if n_dims(diagonal) == 1  # simple case
3     diag, l = _unit_vector(diagonal)
4     _, stride = _unit_vector(_v_sub(match_locations
          [0], match_locations[1]))
5     if stride == 1.0
6       dims = [(diag.index(1), int(l) + 1, stride)]
7     else
8       dims = [(diag.index(1), (l + stride) // stride
            , stride)]
9   else
10    kd = {}
11    keys = []
12    for pos, point in enumerate(match_locations[1:])
13      new_p = _v_sub(match_locations[0], point)
14      dir, l = _unit_vector(new_p)
15      if n_dims(dir) == 1  # collect all unit-dim
            movements
16        if dir not in keys
17          kd[dir] = []
18        keys.append(dir)
19        kd[dir].append(l)
20
21    for dir in keys  # now create a tuple of axis,
22      p = kd[dir]
23      dims.append((dir.index(1.0), len(p)+1, int(p
            [0])))
24  return dims
```

(a) Algorithm extracting embedding dimensions.



(b) Examples for different mappings.

$$a = [(1, 4, 1)]$$
$$b = [(2, 4, 2)]$$
$$c = [(1, 2, 1), (2, 2, 1)]$$

(c) Result of dimension mapping analysis.

Figure 4.10 Examples for the extraction of embedding dimensions from a match. Examples a, b and c demonstrate different, possible mappings between workload and instruction, supported by our framework. The black 4×4 block represents an instruction input matrix, while the coloured projections are possible mappings from a workload. Note that the embeddings are in column i are mutually exclusive and represent different solution options. Lines 2 to 8 analyse the 1D case and is applicable to examples a and b. Example c, mapping the instruction to multiple workload dimensions is analysed by lines 9 to 24.

compiler back-end can interpret. For example, the popular ISL library [105] can lower code to LLVM IR. However, since VTA is programmed via TVM using tensor expressions and loop manipulations to generate a program executable on the VTA, a different code generation path is possible. From the result of matching a single intrinsic in the dataflow graph, the loop splitting, ordering and required layout transformations can be inferred, in order to generate a new tensor expression. This new expression then uses the existing deployment path for VTA. In addition to this process, data-layout rewrites are issued to accommodate the new tensor expression. The following sections will explain the VTA-specific process for relay and tensor expression generation.

4.4.4.1 Embedding Feature Extraction

For VTA, feature extraction focusses on the mappings found for the input and output operands, since they directly determine the mandatory tiling and reordering dimensions of the workload. By iterating over the values assigned from the workload nodes N_o to the intrinsic's input and output groups $N_i^{\mathbf{axz}}$,

$N_i^{\mathbf{bzy}}$ and $N_i^{\mathbf{cxy}}$, the mapped dimensions and their ordering is determined. Since the solver already guarantees a hyper-rectangular, axis aligned set of values, the extraction is reduced to iterating over one of the intrinsic's row and column each to determine the values in Equation 4.10c, as illustrated in Figures 4.10b and 4.10a.

The first step is computing the diagonal $v_n = j_n - j_0$, the vector from the first mapped element in the operator, to the last. In the simple case of examples a and b, where v_n is parallel to any of the operator axis, the mapping is a single, linear dimension. The extraction of mapping properties in the workload is then straight forward. Since the embedding only extends over a single dimensions, $v_n = j_n - j_0$ results in a vector that is zero in all positions, except one. The non-zero element's index is the mapped dimension in the operator. The length is $||v_n||$ of the same vector. The stride is determined by the distance between any two subsequent nodes in the mapping.

For the more complex mapping c, involving multiple workload dimensions, the analysis is slightly more challenging. By computing the vector for each vector formed by mapping indices $[(i, j) : i = 0, 0 < j < J]$, vectors from the origin to each mapped element are computed. Each vector's direction that is part of the vector base determines one mapping set dimension. The length of each dimension is determined by the number of vectors pointing in the same direction and the stride is the length of the shortest vector of each direction.

4.4.4.2 Kernel Generation through Program and Memory Transformation

Using the results from the process described in Figure 4.10, a new, target compatible strategy is generated from the original workload. This new strategy is manipulating both the tensor expression and relay program to generate a new implementation. In the tensor expression, the mapped axes are split according to the embedding and a new dimension for each of the mapped instructions is added at the innermost location, such that the tensorization can be performed. This process is modelled after the manual embedding in Figure 4.8. Parallel to this, the relay program is modified by injecting data-layout transformations for the determined embedding. Both processes are rule-based and each rule is tied to features of workloads and intrinsics.

The data-layout transformations presented in this section were selected after running the embedding procedure on all benchmarking workloads for the used hardware configurations. A manual analysis of the results determined the access

patterns found by the solver. From these, common transformations have been derived and implemented.

First, the program rewrites on relay level are explained. After this, the tensor expression generation is discussed. Table 4.1 has the list of all possible rewrites derived from the mappings. Most operations are already available in relay and only require insertion into the program. The first three are more complex and enable more flexibility with stencil computations, the rest are standard tensor modification routines, available in many DNN frameworks, like TensorFlow, TVM and even numpy.

The first column in Table 4.1 contains the rewrite rule ordering. The application of the first three rules is mutually exclusive for each workload dimension. The column *feature* describes what is looked for in the embedding result and workload to trigger this rewrite for kernel and data layout. The next two columns contain short examples how this affects the data layout and access functions of the workload. relay functions implement all required rewrites prior to the workload and, if necessary, a reverse transformation to recreate the original data layout after the operator. The functions *Dilate Pack*, *Stencil Unroll* and *Image Pack* use `relay.take`, a gather operation copying values in an index list, as no direct implementations are available in relay. All rewrite operations are issued individually. Then, TVM's operator fusion automatically creates kernels combining the transformations into a single kernel.

Stencil Unroll This is a common practice, especially for AI workloads on GPUs and is better known as im2col [14]. Unrolling both image and stencil dimensions enables processing of a *Conv2D* as a matrix multiplication. Advantage of this practice is the rich body of optimization targeting GEMM and generally more robust performance for different convolution operators [22]. A drawback is the increased data footprint. While architectures like GPUs can do this in local memory only, the VTA accelerator has to perform this operation in its DRAM memory before processing. Nonetheless, this stencil unrolling proved to be useful to process convolutions with a low ic count. Figure 4.11c demonstrates this process for a 1D convolution.

Image Packing This operation packs the data elements into individual, parallel channels for processing. This enables greater parallelism during stencil computations, at the cost of a slight overhead to maintain the data overlapping data areas, or halos. The cost of halos is determined by both, stencil and image size.

Figure 4.11a demonstrates this process for the 1D case. For convolutions, these dimensions are directly fused with the batch dimension.

Dilate Pack Dilated convolutions [117] are a primary example of simple algorithm modifications creating immense embedding challenges. The image data of the convolution is now accessed with a regular stride. One effect of the dilation is that neighbouring output elements have no overlapping input data, thus the data reuse of the workload is reduced. However, the missing overlap enables the packing into multiple sub-images, each processed as a dense convolution with no dilation factor. As demonstrated in Figure 4.11b shows the regular stride introduced by a dilation factor in the image access function. Now the whole operation can be decomposed into parallel computations with no data overlap. Both, the image and dilation packing create multiple, parallel dimensions in the workloads. For convolutions, these dimensions are directly fused with the batch dimension.

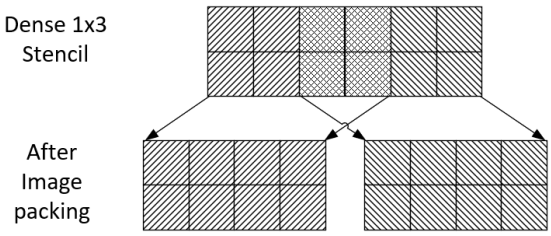
Pad/Mask Hardware intrinsics often work with a specific input data width, however, workloads are not always integer multiples of this value. For VTA *pad* inserts the minimal padding necessary to support the embedding. Implemented with `relay.nn.pad`

Split After the pad operation, each mapped dimension is split by a factor determined during the embedding. This is implemented with `relay.reshape()`. For example, with split list $[(1, 2, 1), (2, 2, 1)]$ and tensor $\mathbf{t}_{4,8,10}$, the following transformation is created: `relay.reshape(t, (4,4,2,5,2))`

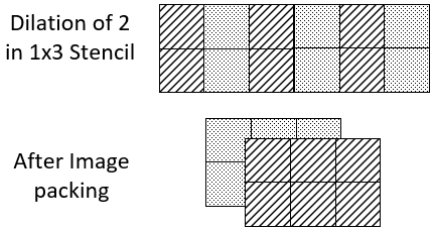
Reorder The dimensions prepared by the previous rewrites are now ordered as required by the VTA hardware:

- For the workload inputs, dimensions involved in the sum (mapped to z) are the innermost dimensions.
- Dimensions x and y are moved as the second innermost dimensions, in order of the found mapping.

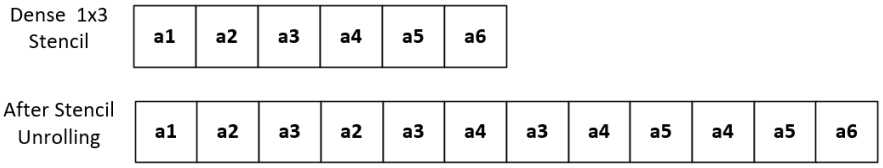
In the case of multiple workload dimensions mapped to the same intrinsic axis, the reorder operations maintains the order of axis mappings determined by the solver. The operation is implemented with `relay.transpose`



(a) Image packing for stencils to enable parallelism in the image dimension.



(b) Memory packing for dilations to enable dense processing of independent sub-images.



(c) Stencil unrolling to enable linear, data-parallel access to sliding window operations.

Figure 4.11 Data-layout transformations for convolution deployment.

Fuse Fuse multiples axes mapped to the same intrinsic dimension, implemented with `relay.reshape`.

After the relay transformations, specialized schedules are generated. By combining knowledge about the hardware and the embedding results, the process is straight forward. First, the transformation on data layout and access functions from Table 4.1 are applied to the original workload. Then, according to the hardware specification, the loops determined in the embedding are split and moved, forming the innermost part of the loop program. This part is then replaced using the `tvm.tensorize` operation. DMA read and write operations are placed dynamically at the outer-most reduction loop in the program. This all is realized in a *schedule template*, where the transformations necessary for tensorization are fixed, but other loop manipulations like *tiling* are parametrizable for optimization with AutoTVM.

4.5 Evaluation

This section evaluates the presented method. First, it compares the new method to existing implementations. To allow for direct comparisions, only embeddings and program rewrites equivalent to ones used by TVM are allowed. This is also in line with the reported abilities of UNIT [111], LIFT [94, 65] and ISAMIR [93]. After this, Section 4.5.2 demonstrates how the program transformation setup

Table 4.1 Data-layout rewrites for dynamic convolution strategy generation. The order and implementation of rewrites is manually defined for specific hardware targets and workloads. Stencil unroll and image pack are mutually exclusive per mapped dimension, which is ensured by the CSP. The hardware intrinsic dimension is d, workloads dimensions are n, i and k. s is the extend of d in n or i. The notation d_e is the access to element e in d.

Order	Strategy	Feature	Data Layout	Access Function
1	Stencil Unroll	$d_e^{out} \cap d_{e+1}^{out} \neq \emptyset \wedge stride(d_e^{out}, d_{e+1}^{out}) = 1$	$[N, I] \Rightarrow [N, I, K]$	$[n, i+k] \Rightarrow [n, i, k]$
1	Dilate Pack	$d_e^{out} \cap d_{e+1}^{out} \neq \emptyset \wedge \|\|d_e^{in} - d_{e+1}^{in}\|\| > 1$	$[N, I] \Rightarrow [N \cdot s_{in}, I/s_{in}]$	$[n, i+(d \cdot k)] \Rightarrow [n, i+k]$
1	Image Pack	$d_e^{out} \cap d_{e+1}^{out} = \emptyset \wedge stride(d_e^{out}, d_{e+1}^{out}) > 1$	$[N, I] \Rightarrow [N \cdot s, \frac{I}{s}]$	$[n, i+k] \Rightarrow [n, i+k]$
2	Pad	$d = i \| i\%split \neq 0$	$[N, I] \Rightarrow [N, I + pad]$	$[n, i] \Rightarrow [n, i]$
3	Split	$d = i \| i\%split = 0$	$[N, I] \Rightarrow [N, \frac{I}{s}, s]$	$[n, i] \Rightarrow [n, i_{out}, i_{in}]$
4	Reorder	<hardware specific>	$[N, I] \Rightarrow [I, N]$	$[n, i] \Rightarrow [i, n]$
5	Fuse	$d = (i, j)$	$[N, I] \Rightarrow [N \cdot I]$	$[n, i] \Rightarrow [f_{ni}]$

can generate code for previously unsupported operators and gracefully handle hardware architecture variations. Finally, in Section 4.6 the embedding effort itself, as well as possible optimizations, are discussed and evaluated.

Evaluation workloads are from the Baidu DeepBench Inference Benchmark Suite[2], providing 108 convolution operators from a range of domains, like image and speech processing. Twelve convolutions cannot be executed on the VTA accelerator without additional zero padding. Section 4.5.2 will discuss possible solutions for this problem in detail. Furthermore, 28 layers in the convolution benchmark cannot be processed by *GeCode*, the solver we implemented our approach in. It uses the default C++ int data type representation of the used compiler, which is 32 bits in our case. Large convolutions can yield domains substantially larger than this limit. Additional 6 layers caused various errors in the TVM compiler, like a VTA instruction buffer overflow. This leaves 62 layers to benchmark. Functional correctness is ensured through numerical comparisons.

4.5.1 Validation Experiments with Static Strategy Generation

The method is validated by comparing the performance of a micro-benchmark with the TVM reference implementation. The benchmark performs tensor packing, convolution, activation, and tensor unpacking. It is implemented as a *relay* program. Packing and unpacking are performed by the ARM host CPU on the Zynq board. For the convolution operator, the TVM reference uses an handmade implementation template that specifies which memory and loop transformations are necessary, as shown in Figure 4.8. This template expects a *NCHW* tensor layout. The new dataflow embedding method automatically generates TVM code based on the found embedding and a specified target

[2]https://github.com/baidu-research/DeepBench, accessed 05.2022

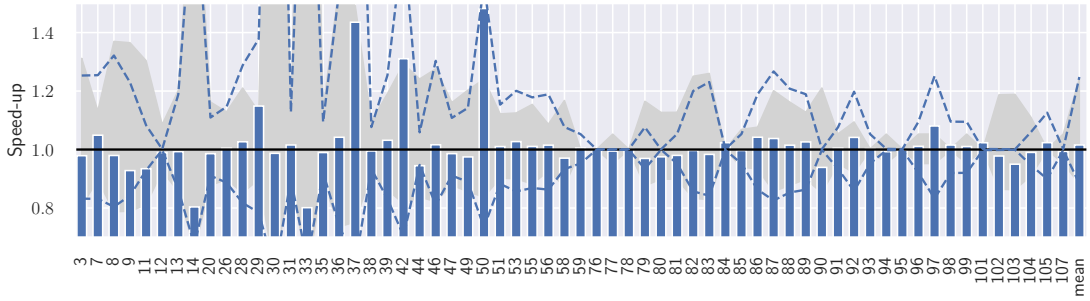


Figure 4.12 Speed-up as ratio of time of the TVM reference to time of this work, for the *Conv2D* layers of the Baidu DeepBench Inference Benchmark. The grey envelope and dashed blue lines are the normalized standard deviation over the reference and this work, respectively. Results show that the baseline of this work based on strict mapping and standard *NCHW* data layout is of comparable performance to the TVM reference, with all but two layers being inside one standard deviation of the reference.

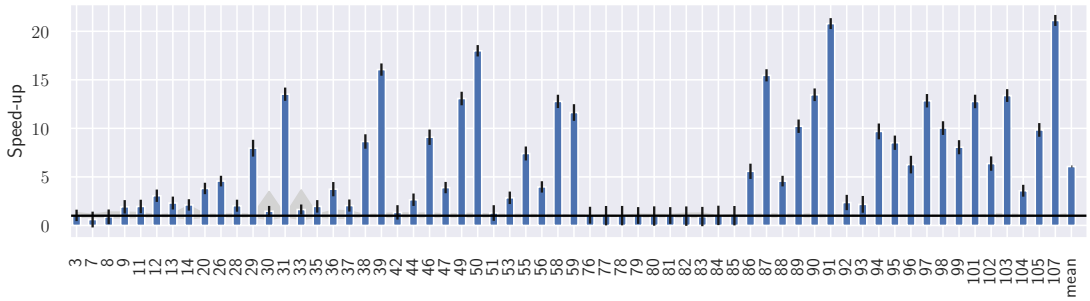


Figure 4.13 Speed-up as ratio of time of TVM reference (*NCHW*) to time for a generated *NHWC* data layout, for the *Conv2D* layers of the Baidu DeepBench Inference Benchmark. Except for four layers, results demonstrate consistent performance advantages and thus indicate the potential of flexible code generation for DNN operators.

memory layout. As intended by the constraint program design, implementations found by the solvers map the instruction dimensions to the same workload dimensions as the reference. From this, a relay program for the tensor packing is generated automatically. Both the reference and our generated solutions use AutoTVM to optimize performance. To ensure comparability both versions use the same optimization parameters found by AutoTVM.

The presented method achieved performance competitive with the TVM reference, as demonstrated in Figure 4.12. After a warm-up, the evaluation averaged over 200 measurements for each layer. Across the benchmark, all but two layers perform within the reference's standard deviation (σ) envelope. The mean σ is $26ms$ for the reference as well as our solution. The absolute differences between the two approaches range from $0.1ms$ to $90ms$. These results show that the automatic embedding can compete with existing, handmade implementations for a hardware accelerator target.

4.5.1.1 Dynamic Data Layout

Dynamically changing the tensor layout of a DNN for global performance optimization, such as changing from *NCHW* to *NHWC* is a topic of interest and already supported some by existing tools [124][63]. However, for accelerators like VTA, only some of the dimensions are free to be rearranged, as only the packed, innermost dimensions are necessary for the embedding. Since the solver determines which dimensions are necessary, and thus which ones are free, the memory layout of the free dimensions can be changed during code generation.

For example, this allows code generation for *Conv2D* with *NHWC* layout, including code for tensor packing and unpacking. Figure 4.13 compares the results of this experiment to the TVM reference, based on *NCHW*. Over all layers of this benchmark, the $NCHW$ layout outperforms the $NHWC$ layout in only four cases. This speed-up effect correlates with the ratio between channels and image size, as layers with larger channels show better performance in the *NHWC* layout. This can be explained with the data movement during the layout transformation. The packing moves data from the ic and oc to be the innermost tensor dimension. When dimensions are closer to their target position, the read access has less stride, yielding better CPU cache utilization. Since not every layer in a network is unpacked and then repacked after processing, the performance gain on a full network would not be as large.

4.5.2 Case Studies on Dynamic Strategy Generation

The previous section demonstrated how dataflow embedding can achieve performance competitive to the existing reference, without having to specify a data layout manually. This section explores scenarios where static strategies can become inefficient due to mismatches between hardware intrinsic and deployment strategy and how dynamic rewrite strategies based on embedding results help improve latency and data movement. The first scenario explores workload variations like dilated convolutions or convolutions with a low number of input channels. The second scenario investigates how changes to the architecture of VTA's processing element influence the deployment.

To enable this in the constraint program, the following modifications are performed:

- **Hyper-Rectangle:** The parameter *stride* is relaxed to allow arbitrary values. This way, any stride is potentially possible. Parameter *dimension* is set to the number of dimensions in the workload.

Table 4.2 All convolutions used for the evaluations in this sections. They all have one or multiple characteristics that reduce the utilization in the VTA reference implementation.

ID	N	IC	H	W	OC	OH	OW	KW	KW	Pad	Stride	Dilation
1	1	1	700	161	32	334	78	20	5	0	2	1
2	2	1	700	161	32	334	78	20	5	0	2	1
3	4	1	700	161	32	334	78	20	5	0	2	1
4	1	1	480	48	16	480	48	3	3	1	1	1
5	1	3	108	108	64	54	54	3	3	1	2	1
6	1	3	224	224	64	224	224	3	3	1	1	1
7	2	3	224	224	64	224	224	3	3	1	1	1
8	1	3	224	224	64	112	112	7	7	3	2	1
9	2	3	224	224	64	112	112	7	7	3	2	1
10	1	1	151	40	32	73	20	20	5	8	2	1
11	1	1	700	161	64	349	79	5	5	1	2	1
12	2	1	700	161	64	349	79	5	5	1	2	1
13	1	304	18	18	448	8	8	3	3	0	1	2
14	1	208	72	72	304	16	16	3	3	0	1	3

- **Memory-Access:** Parallel searches with and without the memory access constraint are posted. This way, solutions with and without im2col are found in the same search process.

4.5.2.1 Low-Channel and Dilated Convolutions

In the previous sections, twelve convolution layers in the benchmark could not be executed on the VTA without zero padding the ic dimension. They are referred to as "low-channel" convolutions. Depth-wise convolutions pose a similar problem to VTA, as they also lack the necessary number of input channels. To this set of workloads, two dilated convolutions are added. The full collection of workloads is listed in 4.2.

As explained in Figure 4.8, the ic dimension is split with a factor the size of dimension z in the instruction. If $ic < z$, padding ic with $z - ic$ additional elements is necessary to generate code. However, padding results in lower effective hardware utilization and a larger memory footprint. For the evaluation, the five best implementations based on the selection metric in Section 4.3 were chosen as candidates for auto-tuning.

Tables 4.3, 4.4 and 4.5 show the performance of solutions found by the solver-based approach, optimizing for operator inference time, combined time for operator and transformation and the memory footprint. All is reported relative to a naive padding strategy. Memory transformations and operator speed-ups are reported individually and combined. The tables report no variance for dilated workloads, since only a single implementation was found, respectively.

Table 4.3 Generated implementations with the best *operator* performance. All numbers are reported relative to the TVM reference with padding.

	Op[a]		Transf.[b]	Combined	Memory		
ID	S[c]	σ	S	S	Data	Weights	Tot.[d]
1	×194.148	28.813	×0.002	×1.750	×2.796	×0.120	×2.722
2	×171.977	27.810	×0.002	×1.332	×2.796	×0.120	×2.758
3	×238.761	59.906	×0.001	×1.608	×2.151	×0.090	×2.137
4	×1.139	0.491	×0.038	×0.272	×0.977	×0.111	×0.972
5	×1.634	0.092	×0.014	×0.231	×1.000	×0.444	×0.974
6	×1.192	0.293	×0.017	×0.286	×3.964	×0.444	×3.924
7	×1.193	0.422	×0.013	×0.244	×3.964	×0.444	×3.944
8	×10.793	3.150	×0.053	×1.137	×0.513	×0.327	×0.502
9	×3.310	0.023	×0.046	×0.876	×0.513	×0.327	×0.508
10	×13.599	3.702	×0.463	×6.973	×0.786	×0.100	×0.549
11	×11.338	0.089	×0.089	×3.380	×0.963	×0.160	×0.952
12	×11.355	3.780	×0.066	×2.737	×0.963	×0.160	×0.958
13	×2.550	-	×0.139	×1.829	×1.0	×0.360	×0.378
14	×23.770	-	×0.486	×18.369	×1.0	×0.074	×0.189
Geo Mean	×10.234		×0.019	×1.005	×1.385	×0.197	×1.328

[a] Operator, [b] Transformation, [c] Speed-up, [d] Total

This overview shows that it's possible to improve memory footprint and overall performance, but often not at the same time. Therefore, optimizing for another objective often leads to a different implementation. A more detailed view reveals that the trade-offs between the implementations the solver generated versus simple padding are complex and need detailed consideration for individual cases.

One of the main drivers of better inference performance is an effective hardware utilization, controlled by the padding. The largest speed-ups are achieved in layers with $ic = 1$. For $ic < z$, only $\frac{ic}{z} \cdot (h \cdot w)$ elements in the input image meaningfully contribute to the result. This drives down the effective utilization.

However, padding in ic and the resulting change in the number of operations alone does not give the full picture when discussing performance. Computing the number of operations in a convolution is the product of all its loops. Since the reference only pads the ic dimension, the maximal possible factor increasing the number of operations is 16 with the used VTA instance. This does not explain the extreme speed-ups in the Layers 1-3. Increasing ic does not only increase the size of the data tensor, but also generates larger weight tensors. In Layers 1-3 and 8-12 the padding creates weight tensors that exceed the capacity of the accelerator's on-chip weight buffer. Implementations generated with the new approach mainly affect the size of the data tensor, so the accelerator can hold the

Table 4.4 Generated implementations with the best *combined* performance, i.e. time for transformation and operator. All numbers are reported relative to the TVM reference with padding.

	Op.[a]	Transf.[b]	Combined		Memory		
ID	S[c]	S	S	σ	Data	Weights	Tot.[d]
1	×117.697	×0.095	×48.089	17.998	×2.330	×0.100	×2.268
2	×104.199	×0.070	×35.411	14.042	×2.330	×0.100	×2.299
3	×170.802	×0.020	×27.686	10.109	×0.974	×0.100	×0.968
4	×1.139	×0.038	×0.272	0.080	×0.977	×0.111	×0.972
5	×1.485	×0.053	×0.618	0.153	×1.000	×0.444	×0.974
6	×1.023	×0.037	×0.466	0.126	×1.004	×0.444	×0.998
7	×1.193	×0.013	×0.244	0.058	×3.964	×0.444	×3.944
8	×10.793	×0.053	×1.137	0.301	×0.513	×0.327	×0.502
9	×3.310	×0.046	×0.876	0.213	×0.513	×0.327	×0.508
10	×13.599	×0.463	×6.973	2.474	×0.786	×0.100	×0.549
11	×11.338	×0.089	×3.380	1.046	×0.963	×0.160	×0.952
12	×11.355	×0.066	×2.737	0.904	×0.963	×0.160	×0.958
13	×2.550	×0.139	×1.829	-	×1.0	×0.360	×0.378
14	×23.770	×0.486	×18.369	-	×1.0	×0.074	×0.189
Geo Mean	×8.788	×0.069	×2.813		×1.121	×0.188	×0.0882

[a] Operator, [b] Transformation, [c] Speed-up, [d] Total

full weight tensor in the on-chip buffer. This effect is less pronounced for the input images, since except for Layer 10, they always exceed the buffer capacity. Ultimately, this also explains why layers with memory footprints equal or larger than the reference with padding still perform better. There is no competition for memory bandwidth capacity by weight transfers. This means data is loaded faster and results are written sooner, which in turn clears the partial result buffer quicker.

Due to the effect of the weight buffer, the data memory footprint is almost negligible as a performance proxy in the overall system. While the stencil unrolling produces data tensors exceeding the padded tensors by factors of up to $\times 2.299$ or $4.6Mbyte$, the same implementations can also provide a speed-up of up to $\times 104$, or $663ms$, versus the reference. Likewise, in Layers 8 and 10, the best operator performance is not achieved by the implementations with the smallest data footprint. Comparing Tables 4.3, 4.4 and 4.5 shows that the size of the weight buffer is also not directly correlated with performance of the memory transformation operation. The implementations with the lowest footprint are rarely the fastest ones.

Generally, the size of the data footprint does not directly relate to the operator performance or transformation performance. It is to assume that the expanding of the stencils causes this. For example, in Layer 1 the data footprints between

Table 4.5 Generated implementations with the smallest *memory footprint*. All numbers are reported relative to the TVM reference with padding.

	Op.[a]	Transf.[b]	Combined	Memory			
ID	S[c]	S	S	Data	Weights	Tot.[d]	σ
1	×116.952	×0.049	×30.692	×0.974	×0.160	×0.952	0.650
2	×103.393	×0.047	×26.665	×0.974	×0.160	×0.963	0.655
3	×170.802	×0.020	×27.686	×0.974	×0.100	×0.968	0.622
4	×1.139	×0.038	×0.272	×0.977	×0.111	×0.972	0.694
5	×1.398	×0.046	×0.558	×0.509	×0.444	×0.506	0.774
6	×1.023	×0.037	×0.466	×1.004	×0.444	×0.998	1.061
7	×0.201	×0.037	×0.168	×1.004	×0.444	×1.001	1.181
8	×10.793	×0.053	×1.137	×0.513	×0.327	×0.502	1.138
9	×3.310	×0.046	×0.876	×0.513	×0.327	×0.508	1.098
10	×3.041	×0.242	×2.191	×0.352	×0.160	×0.286	1.145
11	×11.329	×0.023	×1.112	×0.963	×0.160	×0.952	1.127
12	×1.816	×0.015	×0.569	×0.963	×0.160	×0.958	1.105
13	×2.550	×0.139	×1.829	×1.0	×0.360	×0.378	-
14	×23.770	×0.486	×18.369	×1.0	×0.074	×0.189	-
Geo Mean	×6.068	×0.053	×1.938	×0.765	×0.209	×0.643	

[a] Operator, [b] Transformation, [c] Speed-up, [d] Total

different solutions found by our solver differ by factor of up to $\times 2.865$, or by $3.2Mbyte$, the inference performance difference is only $\times 1.672$, or $2.5ms$.

The dilated convolutions perform reasonably better than the original with padded stencils, even considering the expensive preprocessing and postprocessing of the data. Similar to the low-channel workloads, a main benefit of this method the reduction in weight buffer pressure. The weights are not artificially inflated, reducing the necessary data movement and improve the data reuse.

The memory transformations are between $1ms$ and $60ms$ for most implementations. The simple padding + tiling of the reference is usually one to two orders of magnitude faster. There are two reasons for poor performance. First, the `relay.gather` index list takes up cache space and bandwidth that could be utilized for data in the stencil unrolling. Second, it does not consider cache locality effects during the linear list traversal. This makes further discussion of the memory transformation performance moot.

4.5.2.2 Hardware Intrinsic Variation

This experiment changes the VTA hardware architecture from an $i, j, k = 1 \times 16 \times 16$ to a $i, j, k = 8 \times 8 \times 8$ layout. This increases the arithmetic intensity. As long as $i \leq n$, the new hardware layout is no issue for the embedding process. However, inference with batch size *n = 1* is common for many DNN use cases, so the hardware needs to support this efficiently. Now the static VTA template needs

Table 4.6 Speedup of dynamically generated strategies relative to padding on an 8x8x8 VTA architecture.

	Op.[a]	Transf.[b]	Combined	Memory		
Data, Weight , Pad, Stride	S[c]	S	S	Data	Weights	Tot.[d]
(1, 32, 8, 8) (64 32, 3, 3), 1, 1	×1.60	×0.89	×1.13	×0.60	×1.00	×0.77
(1, 32, 16, 16)(64 32, 3, 3), 1, 1	×6.64	×3.31	×4.27	×0.19	×1.00	×0.33
(1, 32, 32, 32)(64 32, 3, 3), 1, 1	×12.65	×6.20	×7.71	×0.16	×1.00	×0.21
(1, 256, 8, 8)(256, 256, 3, 3 1, 1	×3.85	×1.47	×2.47	×0.60	×1.00	×0.90
(1, 128, 16, 16)(256, 128, 3, 3), 1, 1	×12.00	×4.42	×8.22	×0.19	×1.00	×0.57
(1, 128, 32, 32)(256, 128, 3, 3), 1, 1	×8.04	×11.93	×9.47	×0.16	×1.00	×0.32
(1, 72, 56, 56)(96, 72, 1, 1), 0, 1	×5.62	×9.10	×8.44	×0.14	×1.00	×0.14
(1, 256, 7, 7)(512, 256, 1, 1), 0, 1	×4.60	×4.12	×4.32	×0.22	×1.00	×0.57
(1, 8, 224, 224)(24, 8, 3, 3), 1, 2	×10.27	×6.39	×6.73	×0.13	×1.00	×0.13
(1, 72, 56, 56)(96, 72, 3, 3), 1, 2	×6.95	×5.14	×5.46	×0.16	×1.00	×0.18
Geo Mean	×6.26	×4.21	×4.99	×0.21	×1.00	×0.33

[a] Operator, [b] Transformation, [c] Speed-up, [d] Total

padding in the batch to deploy the workload, reducing the effective hardware utilization and increasing data movement. The next paragraphs will compare the padded solutions to dynamically generated strategies from the rewrite rules in Table 4.1. In this experiment, only the activation tensor is affected, and the parameters stay untouched.

As in the previous section, the top five implementations based on the selection metric in Section 4.3 were chosen as candidates for auto-tuning. The candidate with the best overall performance is used for comparison to the reference. The results are in Table 4.6. Overall improvements for an operator range from $\times 1.13$ to $\times 9.47$, with the operator inference alone showing improvements up to $\times 12.56$. Except for the first, and smallest, workload, the data transformations are now also faster, ranging from $\times 0.89$ to $\times 11.93$. While the new data-layout transformations are more expensive to perform, the increase in batch size makes the default transformations much more expensive, due to the large memory access strides. This is a similar effect as in the experiment in Figure 4.13. The achievable speed-up of this work increases with the image size in the activation, as well as the total size.

The theoretical speed-up from $n = 1$ to $n = 8$ should be limited to 8, for the transformations as well as the execution. The geo-mean speed-ups in all three categories are within this limit. The most probable cause for individual workloads with faster inference is a weak auto-tuning result for the reference. For the two transformations with a speedup > 8, the authors suspect micro-architectural effects. The ARM core performing the layout transformation has a total of *1MByte* of L2 cache. The inputs without padding still fit into the cache,

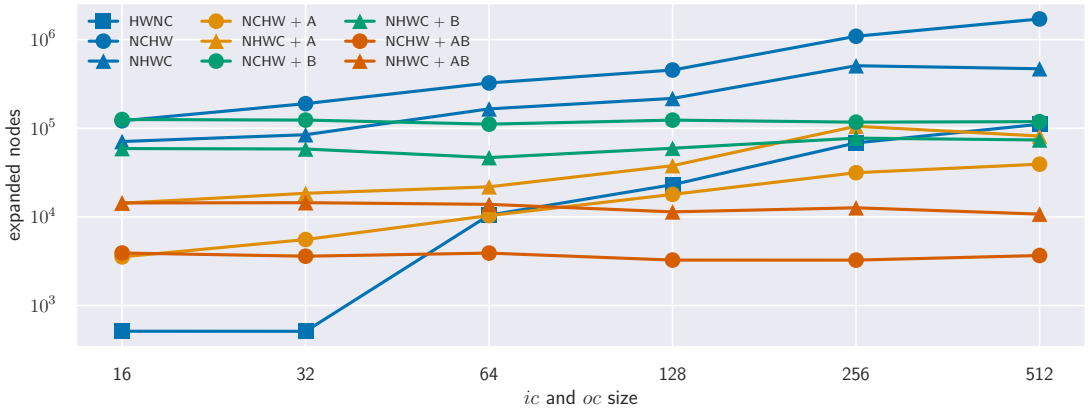


Figure 4.14 Search effort for different *Conv2D* ic and oc channel sizes with various search strategies and domain layouts. A is asset search, B is domain size reduction. AB combines both strategies.

while the padded version exceeds the capacity. But not only processing latency, also, the memory footprint benefits from these measures. Over all workloads it only uses $\times 0.21$ as much memory as the reference. The full factor of $\times 0.125$ is not achieved, since additional padding or data replication is sometimes necessary to enable embeddings, especially for workloads with smaller image sizes. There, the split in the image can't fully capture the parallelism needed for the batch parallelism in hardware, making some padding necessary.

4.6 Search Robustness and Branching Strategy Optimization

The evaluation in Sections 4.5.1 and 4.5.2 showed promising results on the functionality of the proposed method. It can reproduce existing implementations, applies to new workload variations and completely new implementations can be generated by relaxing the search-space constraints. However, a weak point of this method is that the complexity scales directly with the number of operations in both instruction and workload. This section proposes and explore possible search space optimizations.

In constraint solving, variable and value selection strategies determine how the problem space is explored by determining which variable is assigned which value next. To better fit the underlying problem these strategies can be customized. Examining the number of *expanded nodes* gives an indication of how successful

the used strategies are. They are the number of value assignments to variables that have been explored during the search.

This experiment uses a variable selection strategy based on groups of node types in $\mathcal{N}_i$. From the example in Figure 4.2d, all multiplications are in group $\mathcal{N}_i^{\cdot}$. Changing the order of groups can result in varying degrees of propagation. When starting with a group of input nodes, less propagation is possible due to the non-functional relations to their consuming nodes. Starting with $\mathcal{N}_i^{\cdot}$, every node assigned a value automatically propagates to its inputs as well as to the following add operation. The implementation used for all experiments in this chapter begins with the output variables and propagates backwards through the DFG, which proved to be a robust heuristic for short solver runtimes. The value selection strategy implements a lexicographic search through the domain. The performance of constraint programs for different problems is sensitive to the used search strategy. Changing strategies for value and variable selection can lead to exponential differences in runtime for the same problem statement. The presented method amplifies this, as the variable count and domain sizes change for different problems and instructions.

Exploring various operator layouts and search strategies for *Conv2D* demonstrates how these parameters affect the search effort for embedding the VTA GEMM instruction with the strict solution space from Section 4.5.1. In Figure 4.14, the number of search tree nodes expanded by the solver is a measure for effort to find a solution. Without additional search strategies (A, B or AB) a large difference between different operator layouts and an upwards trend for all operator layouts is clearly visible. The ideal layout *HWNC* has a very low initial search effort. This is an artefact, as for small channel sizes (16 and 32), most initial value and variable selections in lexicographic order are correct. For larger layers, the effect dissipates, and the upwards trend is the same as *NCHW* and *NHWC*. While the propagation is efficient at removing large parts of search space, not all of it can be excluded, leading to a linear search through the pruned space. The filtering is limited by the *non-functional* behaviour of some data dependence relations and the commutative nature of the convolution. However, this behaviour is not inherently bad, as it is caused by input value reuse and commutative operations. Without these, different implementation strategies would not be possible in the first place.

4.6.1 Strategies for Improved Robustness

The following paragraphs discuss two methods to improve the search robustness. The first one is a straightforward way to reduce the domain size. A group of constraint variables $\mathcal{N}^d$ with the same domain, for example all variables describing the output operation, can be assigned an *unary* pruning constraint. This constraint limits the size of all dimensions in m-dimensional domain $d_m \subset \mathcal{S}$ for all variables in $\mathcal{G}^d$ by

$$\forall e_i \in d_m | 0 \leq i \leq m : e_i = \begin{cases} e_i & \text{if } b_g \cdot stride \leq e_i \\ b_g \cdot stride & \text{otherwise} \end{cases} \tag{4.39}$$

where b_g is the upper bound for all dimensions e_i in g. Figure 4.14 reports how this method (B) stabilizes the search effort for growing domains. For example, b can be limited to the largest dimension size in the instruction. Since b removes large parts of search space, it can also remove potential solutions. Therefore, a more conservative or adaptive strategy for setting b can help with exploration. Because the solver posts this constraint for every variable individually, the domain propagation happens before the solver begins the search. Therefore, this is equal to simply presenting a smaller problem to the solver. The drawback of this approach is the reduced efficiency with an increasing b and $stride$, and no filtering for dimensions smaller than the threshold value. The propagation done by this constraint overlaps with some work the hyper-rectangle constraint is performing.

The second method changes the order in which the n dimensions of the instance set $\mathcal{S}$ are traversed. This is motivated by Figure 4.14, showing how different dimension orderings for *Conv2D* affect the search. The operator layout (order of dimensions) from the workload specification is traversed in lexicographic order. The layer with an ideal layout *HWNC*, where the dimensions used in strict mapping are traversed first, arrives at a solution faster. To improve the robustness, this work proposes an increased domain exploration diversity instead of a fixed dimension order. A portfolio search [39] uses multiple assets, each searching through the n dimensions of $\mathcal{S}$ in a different order. Every asset is a copy of the problem space, executed concurrently. Applying the portfolio search to the order of dimension traversal in $\mathcal{S}$ yields a more robust search strategy. However, one asset for every possible of the $n!$ permutations of $\mathcal{S}$ would be infeasible.

The portfolio can leverage hardware intrinsic and operator properties to reduce the number of assets. The instance set $\mathcal{S}$ is split into a number of spatial

dimensions n_s and reduction dimensions n_r. For every intrinsic with k_s spatial and k_r reduction dimensions, where $k_s < n_s$ and $k_r < n_r$, only

$$\#assets = \frac{n_s!}{(n_s - k_s)!} \cdot \frac{n_r!}{(n_r - k_r)!} \tag{4.40}$$

assets are necessary to create one asset with an ideal instance set layout for a lexicographic search. This strategy also helps in relaxed search scenarios, since it can potentially increase the exploration diversity. The fact that more assets are created for instructions with more dimensions is not necessarily a drawback. Since all assets can be searched in parallel, a more complex problem could potentially be assigned more resources.

4.6.2 Results

Figure 4.14 reports how asset-based searches (A), domain bounds (B) and their combination (AB) reduce the embedding effort. The asset-based strategy shows a clear reduction in the total effort for both operator layouts. With increasing channel size, the performance becomes comparable to a search with an ideal operator layout. Also, the absolute difference between different memory layouts is reduced. The domain bound (B) limits the effects of increasing the search effort for growing channel sizes. Its effect is especially pronounced for large channels, operators with small domains would not benefit as much from this strategy. Combining both strategies (AB) ultimately leads to a stable and fast search. The difference between data layouts in AB is solely an artefact of the asset creation and execution order.

4.7 Discussion

This chapter presented and explored *dataflow embedding*, a methodology of intrinsic embedding where program rewrites are constructed after a detailed analysis of both workload and hardware intrinsic on dataflow graphs. This deployment technique was developed from the previous chapter's insights.

First and foremost, an embedding process on scalar value granularity was introduced. The complexity of the problem was made manageable by employing the polyhedral model and constraint programming as tools for problem representation and solving. The polyhedral model allows the concise representation of tensor expressions, while maintaining a scalar resolution on data dependencies, memory access and execution order. Constraint programming is a flexible tool

to design search spaces from a set of basic, problem agnostic constraints. The search space formulations enables embeddings without explicit program transformations before the matching. The constraint-based approach then prunes the search space in a structured manner, which is the opposite of the transformation driven approach in Chapter 3, where the search space is a direct product of the available program rewrite operations. The whole process to generate code is based on an embed-and-extrapolate design, where the regularity of tensor workloads is utilized to scale a single instruction embedding to a full workload deployment. This is achieved by deriving the necessary data transformations, loop orderings and intrinsic embeddings from the dataflow embedding.

Besides the process and tools used, a set of constraints for embedding problems were presented by this work, most notably the axis-aligned hyper-rectangle constraint, polyhedral based subgraph-isomorphism, and memory access analysis. All three enable strong propagation properties in the CP's domain. The constraints are agnostic to the target hardware and specific embedding problem and can be reused and combined as necessary for new tasks. This is similar to the transformation from Chapter 3, where individual transformations are independent in their application and effect.

4.7.1 Results

Evaluating the introduced method on the VTA hardware, all but two automatically generated embedding strategies for workloads from the DeepBench Benchmark achieved performance within one standard deviation to the embedding strategy provided by the hardware developers. For operators that could only be realized with additional padding by the existing deployment stack, our solution produced multiple different implementations strategies, enabling trade-offs between memory footprint and operator performance. In many cases, an expensive data-layout transformation improves the processing efficiency drastically, resulting in an overall speed-up. When optimizing for overall performance, a geo-mean speed-up of factor $\times 2.813$ was achieved. However, some cases also demonstrated no obvious benefit at all, or only on individual metrics like the size of the weight tensor. Thus, even naive padding can be viable in some scenarios and should not be completely discarded from the solution space.

Two relatively simple search optimization improved the embedding robustness. An asset-based search helps to scale the search in parallel, as well towards different strengths of diversity in the solution space. For example, multiple, fast

assets in a portfolio can search for the simpler solutions in a very constrained space, while a second group of assets can explore a wider solution space with more complex embeddings.

By principle the method also enables the mapping of multiple workload dimensions to a single intrinsic dimension. The resulting *split* and *fusion* of multiples dimensions are simply derived from the matched input or output values. This level of insight is given by solving the problem on the dataflow level. An iteration domain-based embedding process would have to explicitly formulate and explore such strategies.

The method enables data-layout transformations, from simple tensor transpositions to complex transformations like dilation packing without relying on non-deterministic, rewrite-driven processes. Simple problems, like supporting arbitrary tensor layouts in convolution were enabled without changing the deployment process or constraint program. The only difference was the operator description. For specialized rewrites like dilation packing, the exact conditions for when it is possible can be specified in the constraint program.

A further benefit is the potential to explore an implementation space without providing a set of transformations. By controlling the degree of freedom via constraints, a wide array of solutions can be found and even potentially new transformations could then be derived from the found embedding. On the other hand, undesired parts of the solution space can be removed. For example, by excluding solutions that would require an im2col style unrolling, if no such transformation is available.

4.7.2 Potential Benefits and Limitations

Introducing new transformations is more challenging in this approach. First, the scalar embedding has to bridge back to a coarse-grained transformation primitives. Second, instead of just adding to the stack of transformations, like in the Top-Down process, the constraint program has to be adjusted to make the new transformation a valid solution, without introducing any invalid solutions. This requires an understanding of both the CP program, the hardware and its compiler stack.

On the other hand, this approach should be easier to industrialize. Once a deployment stack, with a fixed set of transformations and constraints is specified, the dataflow embedding should be more robust than the Top-Down approach. In Top-Down, variations in the operator specification need to be accounted for

when designing the pattern-matching and embedding system. In the dataflow embedding, the lowering to dataflow graphs removes the need for such a more elaborate system. This can justify the higher engineering effort.

A drawback of scalar representations is the scaling of the variable domains with size of workload, but it can be mitigated by means of customizable branch-and-bound strategies, like asset-based search. Loop-level representation and embedding used in existing works, like the one in Chapter 3, TVM or ISAMIR do not have this problem, since their representation is decoupled from the workload dimensions sizes.

The scaling of the CP's number of variables with the instruction size cannot be mitigated as easily. Here, the constraints need to offer enough propagation to remove large parts of each variable's domain as early as possible. This is harder to achieve in practice and while this issue did not occur in the performed experiments, it is a potential issue for this method in practice.

With more complex data-layout transformations, more operators can be supported. However, which strategy is the best on a given hardware and operator is not always clear and requires the evaluation of multiple strategies to find the best candidate. This is a challenging problem in general, since optimization towards one aspect, say memory consumption, does not guarantee the best improvements in other aspects as well. Ultimately, this is a system-level decision and needs consideration in the overall application design and is a potential topic for future research.

4.7.3 Related Work

Constraint programming and similar methods like ILP and SAT also serve as the basis for other work in DNN and accelerator deployment, as discussed in Section 2.4. However, there they are mainly used for performance optimizations and not for embedding problems, such as in DORY [12] or the work of Chaudhuri et. al. [13]. Optimization frameworks for GPUs also adopted CP as a vehicle, for example Telamon [10]. While it seems attractive to integrate the performance optimization aspect directly into the embedding, if and how they can be included is an open question. When adding decision variables for optimizations like unrolling and tiling of loops to the problem formulation, an analysis would have to determine which are free to manipulate after the embedding. This implicitly introduces a sequential problem-solving again, instead of a unified approach. And while tiling could be solved by directly modelling value placement in

buffers with scalar variables, it is not clear how questions like the loop unrolling or ordering can be handled on the scalar granularity without losing the local scope of a single embedding used in this chapter.

This chapter relied on existing auto-tuning tools for each found implementation strategy. And while the candidate selection term in Section 4.3.6 helps to filter, the final latency of an operator can only be evaluated after auto-tuning. This leads to prolonged search times for the best strategy, since auto-tuning can be very time-consuming. A speed-up of this process would benefit the overall deployment process, regardless of how the embedding is solved. Therefore, Chapter 5 will investigate auto-tuning on the VTA hardware accelerator and how robustness and tuning time can potentially be improved.

Chapter 5

Hardware-Aware Initialization of Auto-Tuning on Hardware Accelerators

The previous chapters were concerned with finding deployment *strategies*. For any given operator, the strategy is only half of the deployment process. The other half is finding well-performing *schedules*, which are hard to derive by principle. High performance of operators, regardless of the specific target metric, is often enabled by manually implemented libraries, written in low-level languages like C or assembly for a specific hardware target, especially for *Conv2D* [22, 84, 103]. They aim to optimally utilize the hardware architecture and memory hierarchy of the system, often through complex rewrites, which allow the mapping to well understood BLAS [59] operators. In essence, the load, store and arithmetic operations of a specific operator are reordered in such a way, that cache misses, bank conflicts, unnecessary memory transfers, pipeline stalls, register spills or other performance issues are avoided, or at least minimized. Such implementations are time-consuming to write and require a deep understanding of the targeted system.

Hardware-in-the-loop systems, or auto-tuning, help automate this process. Auto-tuning probes a search space of different implementations for the operator. This space is generated from the combination of different, individual loop optimizations. The combinatorial property of this search space prevents brute-force attempts for any meaningful problem.

The content of chapter is accepted for publication on the 2022 Workshop on Accelerated Machine Learning @ HiPEAC 2022 [77]. The data and experiments of this chapter were created during the supervision of the Master's Thesis by Moritz Reiber and have been submitted to examination as part of said Thesis at the University of Tübingen.

Hardware accelerators can have tight resource budgets and more rigid code execution mechanisms. For example, in VTA it is not possible to define a tiling scheme that would overflow one of the local buffers. While a CPU would be less efficient with such code, the VTA software pipeline refuses to compile such code entirely. Many parameter configurations can cause this issue, as well as different operator instances or deployment strategies. The consequence are large subspaces of *invalid configurations* in the auto-tuning search space. Since many auto-tuning tools have at their core a statistical model to aid in the search, investigating these effects is of interest. The statistical models are specialized towards performance prediction and it is not known how well they can adapt to the task of discerning valid from invalid configurations and accurately predict the performance at the same time. This chapter presents the following contribution:

- An analysis of occurrence and effect of such invalid configurations in a search space for *Conv2D* on the VTA accelerator. First, the distribution in different search spaces is analysed. Then, how invalid configurations influence the performance prediction model at the core of AutoTVM, as well as the auto-tuning process itself.
- From these findings, a procedure to prevent the negative influence of the invalid configurations on the auto-tuning process is developed and evaluated.

Chapter 2 already introduced auto-tuning with AutoTVM. In Section 5.1 occurrence, distribution and impact of invalid configurations are explored. This exploration is the basis for possible improvements, which are described in Section 5.2. Evaluation of said improvement over a series of experiments is discussed in Section 5.3.

5.1 Auto-Tuning Space Analysis

The auto-tuning process on hardware accelerators can produce configurations for which no code can be generated. Possible reasons for this can include data- or instruction-buffer overflows, loop ordering issues or wrong loop splitting. Their result are invalid configurations, which distort the performance evaluation at best, and deliver false results at worst. Thus, they waste resources that could have been used to evaluate a valid candidate.

When faulty candidates are used to train the performance prediction model inside the auto-tuner, multiple questions arise. Mainly, how to present these con-

Table 5.1 Workloads set: The workload ID is the respective position in the Baidu DeepBench benchmark suite. The search space sizes shown here are determined with the modified template. The valid ratio is the number of valid candidates, divided by the total number of candidates in the optimization space.

Workload ID	batch	out-channel	image	kernel	in-channel	stride	pad	search space size	valid ratio
3	1	32	79×341	10×5	32	[2 2]	[0 0]	768	0.06
5	4	32	79×341	10×5	32	[2 2]	[0 0]	3072	0.068
8	1	64	12×120	3×3	32	[1 1]	[1 1]	9216	0.067
17	1	256	56×56	3×3	128	[1 1]	[1 1]	20480	0.027
42	1	64	56×56	3×3	64	[1 1]	[1 1]	9216	0.047
48	1	1024	14×14	1×1	256	[2 2]	[0 0]	2240	0.151
53	2	128	28×28	3×3	128	[1 1]	[1 1]	18433	0.035
59	2	512	7×7	1×1	2048	[2 2]	[3 3]	6145	0.008
76	1	64	112×112	1×1	64	[1 1]	[0 0]	14400	0.088
78	1	64	56×56	1×1	256	[1 1]	[0 0]	15361	0.099
92	2	64	112×112	1×1	64	[1 1]	[0 0]	28800	0.082
106	2	2048	14×14	1×1	1024	[2 2]	[0 0]	7169	0.122
107	2	512	7×7	1×1	2048	[1 1]	[0 0]	6144	0.231

figurations towards the model? And subsequently, how does this representation influence the model? With respect to representation, two scenarios are possible. The first is to ignore invalid candidates and only feed back valid performance results. This is useful to avoid a distortion of the model and focusses on the performance prediction task. However, this poses the question if it also makes the model blind to potential invalid configurations. AutoTVM implements a second option, where invalid configs are fed back into the model with a heavy performance penalty. The intention is to train the model towards avoiding the invalid configurations. However, it is not known how the model reacts to this additional complexity. Can the model discern valid configurations and predict performance well enough?

To analyse this impact, a diverse subset from the Baidu DeepBench Inference Benchmark [1] was picked for further experiments. The selected convolutions are presented in Table 5.1. The ID in the table refers to the ID in the benchmark suite. The original auto-tuning template was modified to include additional loop permutations, which increases the search space. The changes added valid, as well as invalid configurations. For these workloads, exhaustive search space explorations were performed. The VTA hardware setup from 4.4.1 was used to gather performance data. This dataset serves as baseline for the analysis in Section 5.1.3 and the experiments in Section 5.3. All experiments rely on the *knob* features, like presented in Table 2.1. They represent the selected configuration of loop tiling, permutation and parallelization factors of the specific candidate in a flattened vector.

[1] https://github.com/baidu-research/DeepBench, accessed 05.2022

(a) Baseline optimization grid.

(b) Grid after validity classification and edge removal.

Figure 5.1 Cluster of grid nodes. Grey nodes are not yet checked for validity. White nodes are invalid, black nodes are valid configurations in the grid.

The right-most column in Table 5.1 shows the share of valid configurations for each search space as $ratio = \frac{v}{t}$, where v is the amount of valid configurations and t is the total search space size. Across all workloads, the overwhelming majority of configurations is invalid. The mean ratio over all workloads is 0.083, with 0.008 and 0.231 being the minimum and maximum, respectively. The largest part of invalid configurations is generated by tiling or loop ordering issues. However, in every evaluation a small fraction of runtime issues occurred, for example connection interrupts between VTA and the host machine. They were not treated differently from actually invalid configurations.

5.1.1 Spatial Organization

The indices of the search-space configurations form a regular, Cartesian grid. In this grid, two configurations are connected if the Manhattan distance [54] between their indices is exactly one. The Manhattan distance is computed by Equation 5.1.

$$d_M(\mathbf{a}_n, \mathbf{b}_n) = \sum_{i=1}^{n} |\mathbf{a}_i - \mathbf{b}_i| \tag{5.1}$$

Removing edges between a valid and an invalid candidate yielded the graphs in Figure 5.1. They show how the grid is split into two independent segments. A segment with valid configurations, and another only with invalid ones. For all researched workloads, exactly two independent segments formed after cutting the edges. This indicates a spatial relationship of valid configurations in the search space.

5.1.2 Stand-Alone Model Evaluation

The AutoTVM model learns to rank the throughput in $GFLOp/s$ of different candidates, invalid candidates are scored with 0.0 $GFLOp/s$. As AutoTVM adds invalid configurations with this penalty to the model training set, it poses the question, how well the model can handle this. Specifically, how treating low-performing and invalid configurations similarly distorts the prediction ability of the model.

AutoTVM's performance model ranks the configuration among themselves, instead of predicting the absolute performance. In every tuning epoch, the tuner optimized the ranking loss over n configurations, thus predicting the relative performance. These top-n candidates are selected for evaluation on hardware.

For the following experiments, workloads $3 - 57$ [2] were randomly sampled for configurations and their performance measured on hardware. Based on this dataset, the model was evaluated in isolation. The configurations from every workload were randomly split into training data (75%) and testing data (25%).

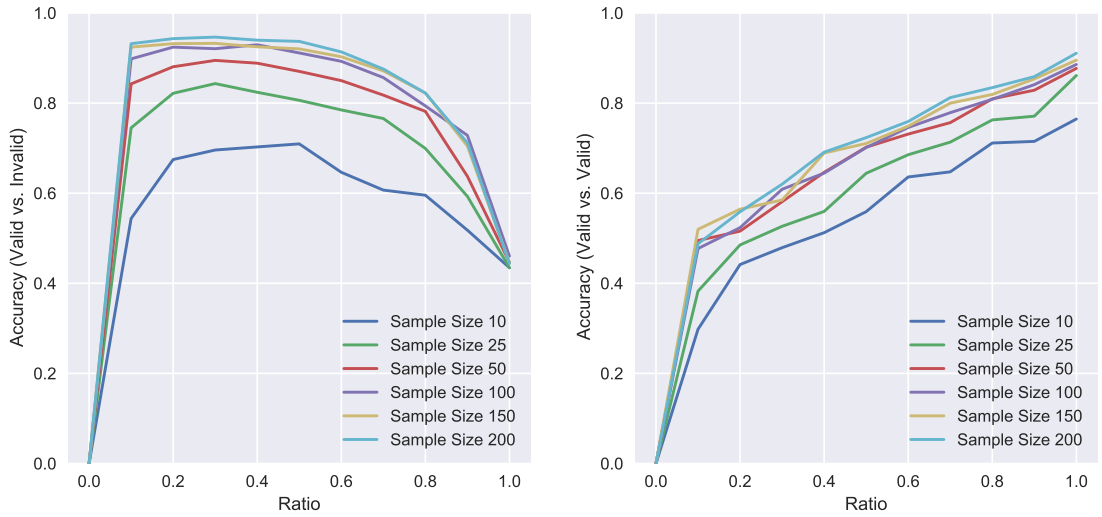
Experiments for the presented metrics were done with a controlled ratio of valid configurations in the training set. A ratio of 0.3 means that 30 out of 100 samples in the training set are valid. This set was used to train the performance model. The model's behaviour was then evaluated on an unmodified training set. For the experiment the default model of AutoTVM, as presented in [18] was used. It is a *gradient boosted tree* [33], implemented in the XGBoost library [16] and using the parameters of configuration space as the input data. This feature-style is known to converge fast for individual operators, but does not transfer to other workloads. Repeating this process for every workload and averaging over the results yielded the data in Figures 5.2 and 5.3.

Pairwise Ranking Accuracy Function $f(x_i)$ is the predicted performance of configuration i over features x_i and measured performance c_i. Invalid configurations are scored with $c_i = 0$. The accuracy of pairwise comparisons is then defined as

$$\text{accuracy} = \frac{1}{n^2 - n/2} \sum_{i>j} \begin{cases} 1 \text{ if } \text{sign}(c_i - c_j) = \text{sign}(f(x_i) - f(x_j)| \\ 0 \text{ else} \end{cases} \tag{5.2}$$

Increasing training sample size improved the performance in both experiments. A low ratio of valid samples in the training set improves the model's ability to

[2] except workloads 6, 10, 15, 21, 27, 34 and 41, which are not executable on VTA



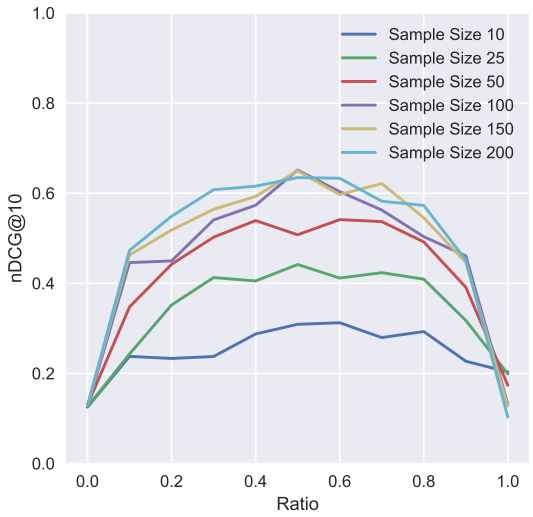
(a) Accuracy Valid-Invalid (b) Accuracy Valid-Valid

Figure 5.2 Cost-model accuracy evaluation.

distinguish between valid and invalid configurations, especially in the accuracy of valid-invalid metric (Figure 5.2a). This is an expected behaviour, as in search spaces with a high proportion of invalid configuration the model's ability to distinguish should improve with more knowledge about the invalid configurations. This, however, comes at the cost of losing the ability to rank the valid samples, as shown by Figure 5.2b, where valid-valid rating ability of the model shows a monotone increase with the valid samples in the training set, and inversely, with a low number of valid samples in the training set, the ability to rank is diminished. Since this evaluation ignores the invalid configurations entirely, the more the model learns about valid samples from the training set, the better is the ability to rank them.

While Figure 5.2b shows a linear increase with a higher frequency of valids, the valid-invalid (5.2a) performance drops significantly. When the model learns less about invalid configurations, the harder it becomes to distinguish them from valid configurations. Even further, the diminishing ability to separate valids from invalids outweighs the theoretically better ranking performance in models trained on higher ratios.

Normalized Discounted Cumulative Gain Accuracy is only a metric for the ability of the model to distinguish between the performance of individual configurations. It does not give indication of ranking quality in the top-n candidates. The normalized discounted cumulative gain ($nDCG$) is a metric to quantify this ranking ability. This includes the ability to only retrieve valid



(a) nDCG

Figure 5.3 Cost-model ranking evaluation.

configurations from the search space into the top-n ranking, since any valid candidate is better than an invalid one. The nDCG cumulative gain is computed by the following formula:

$$\text{nDCG}\,@n = \frac{\sum_{i=1}^{n} \frac{c_{r_i}}{\log_2(r_i+1)}}{\sum_{i=1}^{n} \frac{c_{r_i^*}}{\log_2(r_i^*+1)}} \tag{5.3}$$

First, it computes the sum of measured performances c_{r_i}, discounted by their respective position r_i produced in the ranking. This true score is then normalized by dividing it by the score of an ideal ranking r^*.

Each time a candidate among the top-n is ranked above another candidate with better measured performance, the score decreases. This quantifies the ranking quality. And since the invalid configurations in the top-n will always have a worse measured performance than any valid configuration, the metric can also be used to measure the ability of the model to discern valid and invalid configurations. The score of nDCG should not be interpreted in same way as precision or accuracy. A score of 0.9 does not mean, that 90% of decisions were correct. It only allows relative comparisons in ranking ability between two nDCG scores, where a higher score means a better overall performance.

The nDCG performance is displayed in Figure 5.3a, where across the full ratio scale a bell-shaped curve appears. Since nDCG includes both the ability to rank and to distinguish between valid and invalid configurations, this makes sense. At a low ratio, the model fails at ranking the valid samples, correctly. This

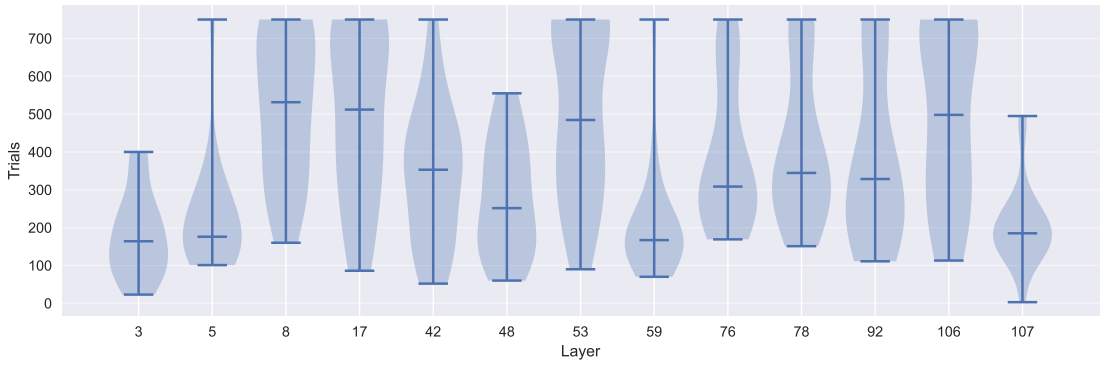


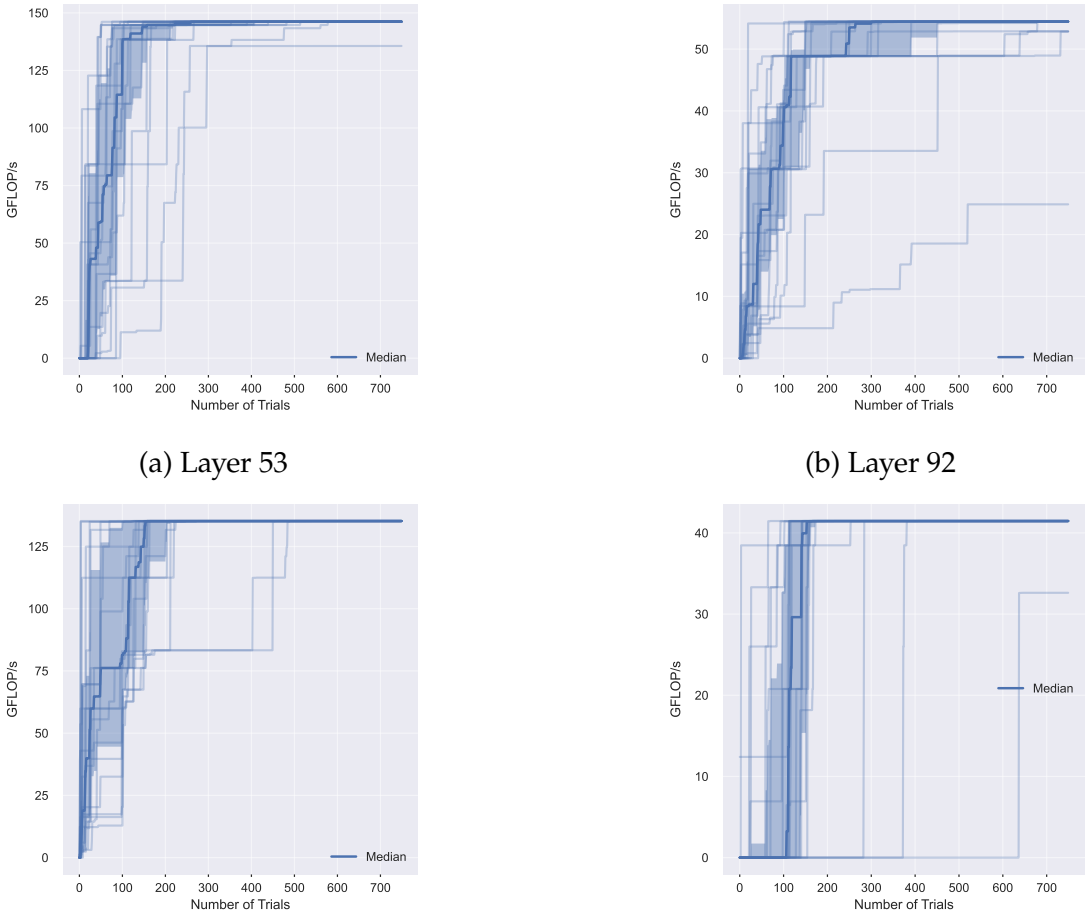
Figure 5.4 Distribution of trials to find the best configuration for each workload over 20 runs with the auto-tuner.

improves until an equilibrium at a ratio of 0.5 is reached. A further increase in valid samples then leads to a dropping score, as the ability to filter the invalids in the ranking correctly is reduced.

The observations on accuracy and ranking ability confirm the hypothesis that ignoring invalid configurations distorts the performance model. When treating invalid configurations the same way as low-performance configurations, the model's ranking quality is affected. Further, it seems that identifying invalid candidates has more influence on the ranking than the ability to rank among valids (see Figures 5.2a and 5.2b). Interesting is the equilibrium reached in the areas of balanced valid/invalid ratios, as the model seems to reach a sweet spot that, given enough data, has satisfactory ranking ability as well as the ability to distinguish between valid and invalid configurations.

5.1.3 Tuning Robustness

In the previous section, the performance model was analysed in isolation. Thus, only the effects on the model were observed, but not how this affects the full auto-tuning process. For the following experiments, we assume an untrained performance model, as is the default in AutoTVM. As a result, the first measurement epoch is randomly sampled from the search space. We use an epoch size of $e = 50$ and a total tuning length of $t = 750$ trials. This means that in total 750 candidates are measured on the actual hardware, in batches of 50. After each epoch, the measurement data is used to further train the performance prediction model. This improved model is then used by simulated annealing algorithm to traverse the search space, ranking unmeasured data points. The top-e configurations found are then selected for measurement on hardware.



(a) Layer 53

(b) Layer 92

(c) Layer 107

(d) Layer 59

Figure 5.5 Progression of multiple auto-tuning runs: Each curve shows the best performance found in tuning run so far. The median over all runs is plotted bold and the shaded area is the space between the first and third quartile.

Additional experiments were conducted on a pre-trained performance model, which eliminates the uniform sampling in the initial epoch. This so-called *transfer learning* from different workloads on the same hardware platform can help to reduce the randomness, especially in the beginning of the search.

5.1.3.1 Tuning from Scratch

Figure 5.4 reports the number of trails necessary to find the best-performing configuration, aggregated over all 20 runs for each layer in Table 5.1. In this chapter, this will be called the *convergence* of a tuning run. The first and most important observation is the large spread across most workloads. This spread determines the *robustness* of the search algorithm. Workloads 8, 17,42, 48, 53 and 106 have a relatively even distribution over all runs, which indicates towards low

robustness. The other workloads are unimodal with a longer tail-end. This is an indication for higher robustness with some outliers. While most only have slower outliers, layer 107 has both, faster and slower ones. Although workloads 59 and 107 have a narrower spread, they also have multiple outliers where it took significantly longer to find the best configuration. Still, they are consistently below 200 trials to find the best configuration. Layers 3 and 5 perform well when looking that the absolute number of trials, however, with 768 and 3072 they have relatively small search spaces and the spread to find the best configurations is wide.

Search space size alone, however, cannot be an explanation, as layers 106 and 107 have similarly dimensioned search spaces (7169 and 6144) and differ significantly in their robustness. A possible explanation is the 23.4% share of valid configurations in the search space of 107, compared to 12.2%. However, layer 59 is the polar opposite with 0.8% valid configurations and a comparable search space size of 6145. A possible explanation for this behaviour: once the model learned how a valid candidate looks like, they are more often included in the top-e samples, which makes finding the best configuration a question of time. Multiple candidates with the same top performance could be another explanation for this, especially when the absolute performance in $GFLOP/s$ is low, compared to layers 107 or 53 (see Figures 5.5c and 5.5a).

Figure 5.5 shows all runs for layers 53, 92, 107 and 59, which offers additional information on the behaviour of individual runs. For example, in layer 59 and 92 some runs did not find the best configuration during the 750 trials. It also shows that the initial set of 50 randomly selected configurations greatly influences how long it takes the tuner to converge towards a good solution. Bad initial configurations can lead to plateaus in the search, which can last over multiple epochs. It is possible, that the improvements found after several hundred trials can be attributed to random candidate selection, rather than model quality.

5.1.3.2 Tuning on Pretrained Model

Starting with an already trained model promises to reduce the convergence time by reducing the randomness in the search and give the model a first "lay of the land". Transfer learning was conducted on 30 000 samples from all workloads in Table 5.1. This relatively large amount of data is necessary due to the more complex feature set used. However, only 63,04% of the samples are ultimately used for training, as some, but not all, invalid samples are already discarded during feature extraction. TVM cannot feed this information into

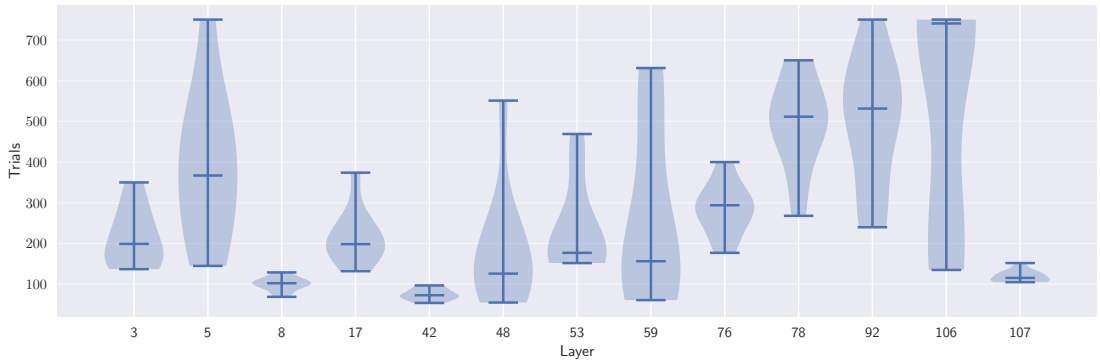


Figure 5.6 Distribution of trials to find the best configuration for each workload over 10 pre-trained runs with the auto-tuner.

Table 5.2 Tuner overhead in seconds.

Table 5.3 The left-hand side and middle show the overhead of knob and context relation features at runtime. This includes feature extraction, simulated annealing and model training on new measurements. The right-hand side column is the initial model training time for transfer learning.

Layer	Knob	Context-Relations Features	Initial Model Training
78	77.8	656.8	512.8
8	77.5	1118.7	839.9
92	88.4	1139.7	653.7

the performance model, as it could not retrieve a full set of features from these candidates. Thus, the performance model only sees a partial representation of the invalid candidates. The training data is sampled randomly from all workloads under investigation. For the training of a model for a specific layer, its own data is excluded from the training set.

The features in the previous sections were simply the knob values in the configuration space. However, these features are specific to each workload. In this section the auto-tuning uses input features called *context relation features*, which create a detailed but abstract workload profile, enabling a degree of portability. It represents an aggregation of memory accesses and arithmetic operations over various loops in the program. More details can be found in [18]. The expectation is that the pre-trained model performs better initial sampling. The experiments in Figure 5.6 however show, that this is not always the case. For workloads 3, 8, 17, 42, 48, 53, 76, 78 and 107 performance and convergence improved, as expected. Although the model already can filter some of the invalid candidates through training and errors during feature extraction, workloads 5, 53, 92 and 106 performed worse than training from scratch when looking at

the median. Possibly, the latter workloads have features that cause a different performance behaviour. For example, all have a $stride > 1$. Together with the skewed training data, this can lead the model to learn the wrong things, distorting the search. Robustness improved as most workloads now exhibit a mostly unimodal distribution around the median.

Other issues are features and training themselves. Extracting them from a workload is expensive, as shown in Table 5.3. Both the non-measuring related overhead and the initial model training are time-consuming to perform. When focussing on the trials, the model actually often finds the best solution faster than the baseline starting with an untrained model. However, the cost of extracting the features and initializing the model can produce longer wall-clock times. During a single run, extracting the features can add over 20 minutes to the total time. Training the performance model can also extend well over 10 minutes with 30.000 samples. While using less samples would reduce the training time, it would also impact model quality.

5.2 Validity Driven Model Initialization

Following the insights from the data in Section 5.1, a better initialization process for the performance model seems like a promising path to improve robustness and reduce the number of necessary hardware measurements. The main problem to address is the uniform sampling over a non-uniformly distributed solution space, as shown by the locality analysis in Figure 5.1. Transfer learning helps with this but can also fail when operators are too different from the learned data. Also, the current implementation is inefficient and increases wall-clock time spent. The new process is aiming to improve the substantial robustness issues of auto-tuning runs by training a fresh model with a better initial data set to achieve a balance in valid/invalid classification as well as performance prediction. Data in Figure5.3a shows that this is achievable with a balanced data set and relatively few samples for the model, starting from 50 to 100 training examples.

Sampling valid candidates from the search space is the first step in this process. From the found valid and invalid configurations, a balanced initial measure batch is created. Secondly, the valid configurations are stored and used to assist the simulated annealing exploration when selecting later measure batches. Section 5.2.1 and 5.2.1 have details the on the former and 5.2.3 on the latter.

Algorithm 4 Algorithm for locality driven search space sampling.

```
1:  procedure PRESAMPLE(n_samples, n_parallel, S)
2:      N ← ∅
3:      P ← randomly sample n_parallel points from the search space S
4:      while |N| < n_samples do
5:          C ← ∅
6:          for p in P do
7:              r_p ← evaluate validity of p with validity check
8:              N ← N ∪ {(p, r_p)}
9:              if is-valid(r_p) then
10:                 C ← C ∪ { neighbors of p ∉ C ∪ N}
11:             else
12:                 C ← C ∪ { random new point p ∈ S∧ ∉ C ∪ N}
13:             end if
14:         end for
15:         P ← randomly sample n_parallel points from the candidates C
16:     end while
17:     return N
18: end procedure
```

5.2.1 Neighbourhood-based presampling

Instead of selecting the initial measurement batch at random, a balanced set of valid and invalid configurations is used. This information is obtained by a presampling algorithm which explores the search space and stores information about the validity of individual configurations.

In search spaces with only a fraction of valid configurations, random sampling is not sufficient to find enough valid samples in a reasonable amount of time. Aiming for a ratio of 0.5 valid configurations in the first measure batch of size 50, a purely random presampling would have to evaluate thousands of configurations. By leveraging the spatial locality of valid candidates from Section 5.1.1, the randomness is reduced and more samples are retrieved from the search space.

The first task is to identify the valid configurations. Since the compilation process of TVM already captures a large share of invalid candidates, it can also be utilized in the presampling phase. A drawback of this approach is the relatively long latency of around 500ms per evaluation.

The data in Section 5.1.1 revealed that valid configurations are often spatially adjacent. Algorithm 4 describes how this feature is used to find more valid samples in the search space. In a first step, a randomly sampled set of configurations $\mathcal{P}_0$ from search space $\mathcal{S}$ is classified as valid or invalid (line 3). Each configuration

$p \in \mathcal{P}$ is added to output set $\mathcal{N}$, together with the validity information. If p is valid, locality is exploited by adding all its neighbours to the set of potential candidates for the next evaluation step $\mathcal{C}$ (line 10). Two configurations are neighbours if their Manhattan distance is 1. The set of neighbours has a higher chance of containing additional valid configurations. To maintain exploration, each invalid configuration adds a random new candidate from $\mathcal{S}$ to $\mathcal{C}$ (lines $11 - 12$). After processing all members of $\mathcal{P}$, a new set $\mathcal{P}$ is sampled from $\mathcal{C}$, as shown in line 15. These steps are repeated until the limit of $n_{samples}$ is reached. From $\mathcal{N}$, the valid and invalid points to generate the balanced initialization set for AutoTVM is generated.

All evaluated workloads in Section 5.1.1 only had a single cluster. This however, is not guaranteed. Random sampling at the beginning and during the search ensures a degree of exploration, opposed to the exploitation of locality once a valid sample is found.

5.2.2 Sample Selection for Distance Maximization

presampling, as described in the previous section, generates a set $\mathcal{N}$ of configurations and information on their validity. From $\mathcal{N}$, the initial measurement epoch $\mathcal{E}_0 \subset \mathcal{N}$ is selected, such that a desired ratio of valid and invalid configurations is achieved. To do this, $\mathcal{N}$ should be larger than $\mathcal{E}_0$, to ensure a degree of diversity in the possible candidates for $\mathcal{E}_0$. A result of the presampling algorithm is that $\mathcal{N}$ can be defined as $\mathcal{N} = \mathcal{N}^{valid} \cup \mathcal{N}^{invalid}$. Therefore, $\mathcal{E}_0$ can be constructed from the configurations $\mathcal{E}_0^{valid} \subset \mathcal{N}_{valid}$ and $\mathcal{E}_0^{invalid} \subset \mathcal{N}^{invalid}$ as $\mathcal{E}_0 = \mathcal{E}_0^{valid} \cup \mathcal{E}_0^{invalid}$.

The neighbourhood-driven presampling fills N with a high number of valid configurations. Due to the neighbourhood-heavy sampling, $N^{valid} \in N$ are often very similar, with configurations that differ only in a single parameter. When sampling from N, diversity has to be ensured. Otherwise, the model created can struggle with generalizing in the later search [118]. Covering $\mathcal{N}$ as uniformly as possible is a sensible choice.

However, previous experience with VTA also showed that particularly good configurations usually reside on the edge of clusters, and thus on the edge to the space of valid configurations. Origin of this phenomenon could be the loop-tiling parameters. Configurations on the valid-cluster's edge fill the local hardware buffers as full as possible, before the next larger tile-size would overflow them. Therefore, the sampling method was designed to include this observation, while maintaining a uniform sampling strategy. Since measures like euclidean distance

Algorithm 5 Sample selection with distance maximization

```
1:  procedure DISTANCE-MAXIMIZING-SAMPLE-SELECTION(𝒩, num)
2:      ℰ ← { sample an initial point from 𝒩}
3:      while |ℰ| < min(num, |𝒩|) do
4:          δ*_mean ← 0                          ▷ maximal average distance
5:          δ*_min ← 0                           ▷ maximal individual min-distance
6:          k_new ← None
7:          for n in 𝒩 \ ℰ do
8:              Δ_{n,ℰ} ← {d(n, k) ∀ k ∈ ℰ}
9:              if mean(Δ_{n,ℰ}) > δ*_mean ∧ min(Δ_{n,ℰ}) > δ*_min then
10:                 δ*_mean ← mean(Δ_{n,ℰ})
11:                 k_new ← n
12:                 δ*_min ← min(Δ_{n,ℰ})
13:             end if
14:         end for
15:         ℰ ← ℰ ∪ {k_new}
16:     end whilereturn ℰ
17: end procedure
```

are hard to interpret in the search space, we use the neighbour distance used in Section 5.1.1, which can be interpreted as the Manhattan distance between configurations, similar to the analysis in Section 5.1.

The full procedure is defined in Algorithm 5. The first step is randomly selecting one configuration from N and add it to E. To select the next configurations to add (set k_{new}), for each $n \in \mathcal{N} \setminus \mathcal{E}$ the distance to every configuration $k \in \mathcal{E}$ is computed with equation (5.1) and added to set $\Delta_{n,\mathcal{E}}$. The minimal and mean distance of $\Delta_{n,\mathcal{E}}$ determine if $k_{new} \leftarrow n$ (lines 9 to 12). After updating $\mathcal{E}$ with k_{new}, the process is repeated until $|\mathcal{E}|$ reaches a predefined value or $\mathcal{N} \setminus \mathcal{E} = \emptyset$.

5.2.3 Validity Bias During Simulated Annealing

The validity data collected during presampling can also be used in other phases of the auto-tuning process. After the initial batch was measured and model was initialized, simulated annealing is used to select measurement batches from the search space. The SA algorithm selects these based on a score predicted by the performance model. However, as shown in the model evaluation, it can often happen, that invalid samples are ranked well, albeit they could not actually be executed in hardware.

Since the validity of a search space subset is already known at runtime, this information can be used to bias the simulated annealing. In experiments, the value

of performance model predictions were mostly in the interval $[-7, 7]$. Known invalids are set -10^6 to remove them from the top-n ranking. Known valids need to be handled with more care. Since we do not know their performance, a place in the top-n should not be guaranteed. A bias value of 1 showed good results in empirical evaluations. It boosts the ability to place in the ranking without overshadowing possible better implementations.

5.3 Evaluation

For the experiments, the original AutoTVM was adapted. It now uses the presampling algorithm from Section 5.2.1 to explore the specific search space and gather the data to replace random initialization of the first measure epoch E_0 with a set of configurations created by the process from Section 5.2.2. From there, the existing flow of AutoTVM is used: measurements in epoch $\mathcal{E}_i$ are used to train the performance model. This performance model is then used by the simulated annealing algorithm to select the configuration for the next measure epoch $\mathcal{E}_{i+1}$. This new batch of configurations is then evaluated on hardware and the cycle repeats. The original AutoTVM implementation has been altered in two locations: 1) The simulated annealing now uses the bias information from Section 5.2.3 and 2) hardware evaluations are replaced with querying the ground-truth data gathered once for the full search space. All experiments use an epoch size of 50 measurements, or trials, until a total of 750 evaluations on hardware are performed.

The *performance* of the auto-tuner is defined as the ability to find good configurations fast and consistently. The following experiments use the median and interquartile range as a measure of the former and latter criterion. The median gives a good indication on the *convergence* time of the auto-tuner. It is more robust against outliers than the mean, which can be common in the examined problem space (see Figure 5.4). The interquartile range (IQR) describes central half of all values and is useful to determine how robust the overall process is i.e., how good the median does actually approximate the auto-tuners convergence time. IQR is defined as:

$$IQR = Q_3 - Q_1 \tag{5.4}$$

where Q_1 and Q_3 are the first and third quartile respectively.

Table 5.4 Tuning performance relative to the baseline over 20 runs each: On the left-hand side is the median number of trials to converge to the best performing candidate is. On the right-hand side is the IQR over all tuning runs, as a measure of robustness.

	Convergence			Robustness		
Layer	Full	No SSDM[a]	No BSA[b]	Full	No SSDM[a]	No BSA[b]
3	0.305	**0.223**	0.308	0.345	**0.282**	0.452
5	**0.460**	0.668	0.540	0.362	0.426	**0.289**
8	0.433	0.548	**0.412**	0.375	0.453	**0.288**
17	**0.438**	0.479	0.537	0.580	0.593	**0.491**
42	0.319	**0.312**	0.455	0.177	**0.157**	0.408
48	**0.270**	0.390	0.284	0.333	**0.251**	0.350
53	**0.480**	0.491	0.571	**0.333**	0.367	0.599
59	0.120	**0.111**	0.120	**0.861**	1.238	0.879
76	**0.460**	0.530	0.493	**0.251**	0.364	0.325
78	**0.447**	0.557	0.457	**0.176**	0.353	0.271
92	**0.492**	0.518	0.604	0.269	0.453	**0.194**
106	**0.403**	0.720	0.456	**0.631**	0.938	0.788
107	0.786	**0.735**	0.857	**0.815**	1.164	1.178
Mean	**0.416**	0.483	0.469	**0.424**	0.541	0.501

[a] Sample Selection w. Distance Maximization, [b] Biased Simulated Annealing

5.3.1 Baseline Method

In the following Section, the effects of the improved model initializations, distance maximizing sample selection and simulated annealing bias are investigated. Runs with an unmodified AutoTVM serve as a baseline against a fully improved version, one without distance maximization and one without simulated annealing bias. For the baseline, the presampling was skipped and the first epoch $\mathcal{E}_0,\ |\mathcal{E}_0| = 50$ randomly initialized. In case there are no valid configurations in $\mathcal{E}_0$, random initialization was repeated.

In all experiments with the extended auto-tuning, the presampling set $\mathcal{N} \subset \mathcal{S}$ of each search space $\mathcal{S}$ is built from $min(1000, |\mathcal{S}|)$ configurations. Model initialization is then performed as described in Section 5.2, with an initial measurement epoch of 50. Selection of $\mathcal{E}^{valid}$ and $\mathcal{E}^{invalid}$ is done with sample distance maximization.

5.3.1.1 Convergence

For all described experiments, the median search times for the best configurations are presented in Table 5.4. All results are reported relative to the AutoTVM baseline. The improved methods outperform the baseline for every workload, with the mean over the median trials being $\times 0.416$, $\times 0.438$ and $\times 0.469$, relative to the baseline. However, two results stand out: layer 59 is remarkably better than the baseline ($\times 0.111$) and layer 107 shows weaker improvements ($\times 0.735$). These

are also the workloads with lowest (layer 59) and highest (layer 107) number of valids in the search space. For layer 59, if there are very few valids, finding any of these has much higher chance of being the best one. Layer 107 on the other hand, has the highest number of valid candidates, meaning that finding valids alone is not sufficient for good performance, making the search process harder. Nonetheless, the initialization method improved the convergence time by $\times 1.36$. The effects of distance maximization and biased simulated annealing (BSA) are visible, as the best values are most often found by the full experiment, but not dominant towards the overall performance. Turning the additional features off only leads to a slight decrease of convergence and IQR, but both are still in the same order of magnitude. The common ground of all trials is the presampling and model initialization, which gives a workload specific, validity driven head start to the performance prediction model.

5.3.1.2 Robustness

Similar results are achieved for the robustness of the search, also presented in Table 5.4. IQR reductions to $\times 0.424$, $\times 0.541$ and $\times 0.501$ of the baseline are found in the experiments. However, some additional observations can be made. First, only half of the time the full setup delivers the most robust results. Especially turning off BSA often improves the results. On the other hand, two experiments actually performed worse than the baseline when distance maximization was turned off during the sample selection. However, these were already stable workloads in Figure 5.4, so improving upon the baseline is rather hard, which also showed in the results of the full method. Only an IQR improvement of $\times 1.22$ was achieved, compared to the baseline. In other layers, like 3, 5, 8, 42, 48 or 92, the IRQ metric was improved significantly by factors of three to six.

5.3.1.3 Individual Workload Analysis

Figure 5.7 details the experimental results in absolute numbers. It compares the baseline and the experiments with the full improvement set. All the median values of the presented method are consistently below 300 trials, with only layers 8, 17 and 53 requiring more than 200 trials. A first overview shows improved robustness and convergence time for all workloads. Further, the robustness of the proposed method roughly follows the baseline, but always improves upon it. Except for 8, 17 and 106, all runs have unimodal or bimodal distribution, which is a sign of higher robustness. The bimodal distribution only appears

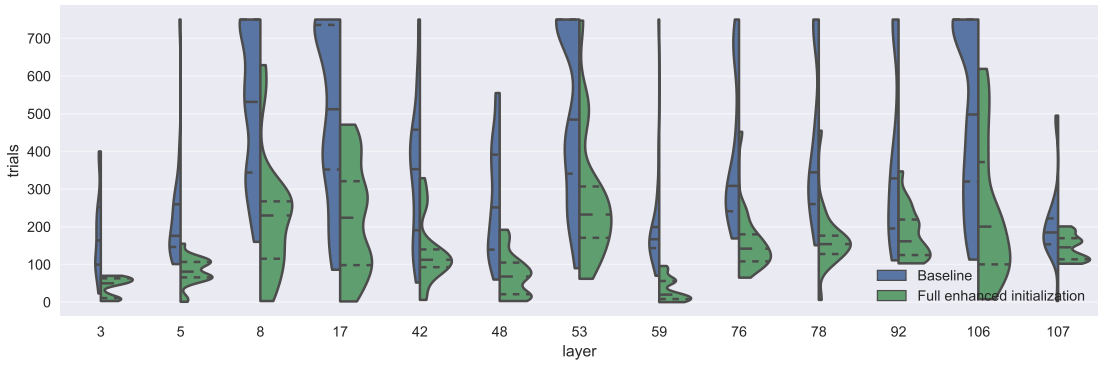
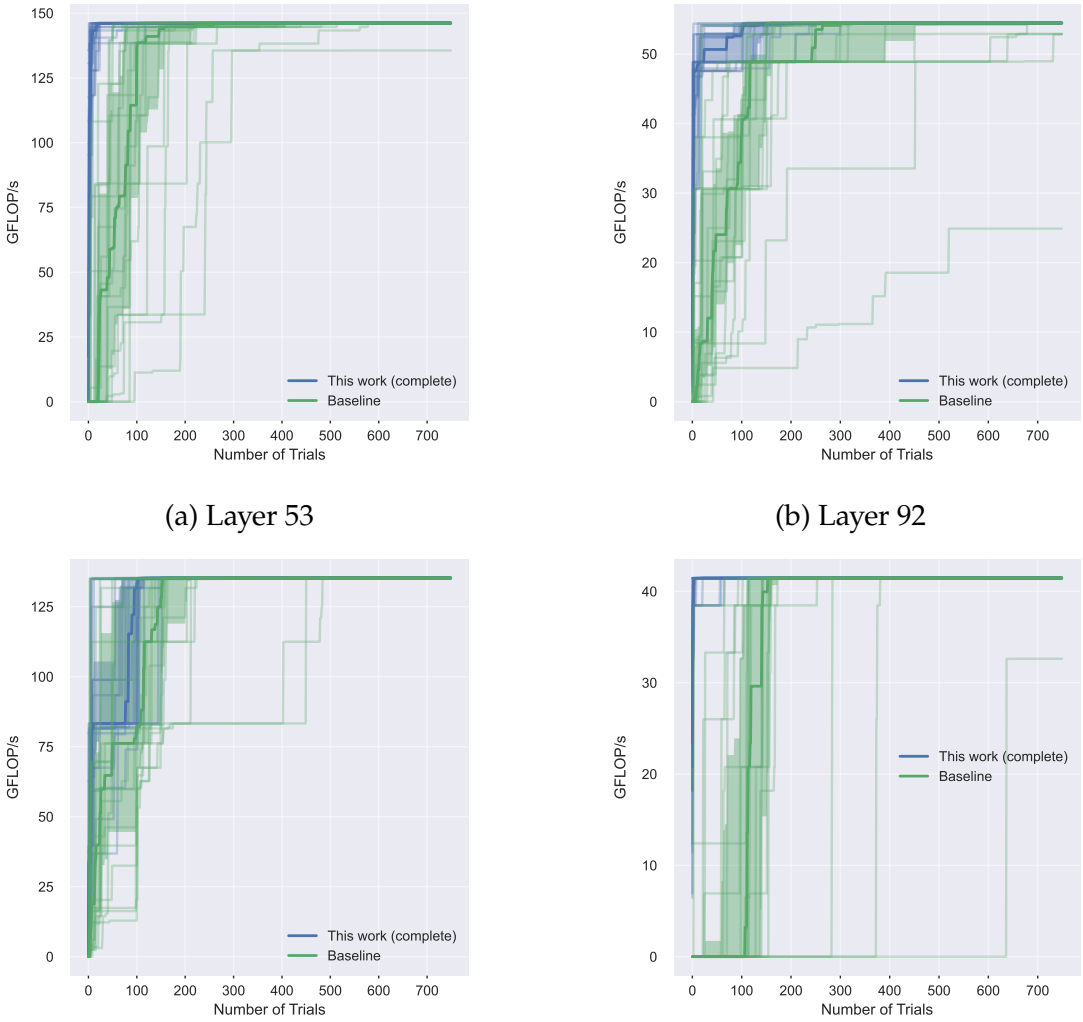


Figure 5.7 Distribution of trials to find the best configuration for each workload for the new method and the AutoTVM baseline over 20 tuning runs.

in workloads with very low tuning times. This is caused by the separation of tuning into batches, where the early batches often already contain the best solution. Longer tuning runs flatten this. Also, the new method produces fewer tails in the distributions. Meaning that a layer with narrow IQR in the baseline, will also have a narrow IQR with the improved initialization and a wide IQR in the baseline leads to a wider IQR in the new method. While the overall performance improved, some runs still produced outlier performing significantly worse than the rest, for example layers 8, 42 or 53. Interesting is layer 42, where the IQR with presampling is narrow, but several positive and negative outliers exist. The baseline, on the other hand spreads from 50 to 170 trials, with IQR between 200 and 450. This hints towards a workload, that is rather difficult for the performance model to learn and where the initial dataset has a large impact on the rest of the search. This is the complete opposite of layer like 3, 5 or 107, where spread is narrowed considerably and outliers are few or non-existent. The high robustness of 59 can again be explained with extremely low number of valid configuration in the search space.

A more detailed view on individual runs is provided by Figure 5.8, in which every run of layers 53, 92, 107 and 59 is plotted for both baseline and enhanced method. This reveals additional information on the convergence of individual runs. Previous analysis focussed on finding the maximum performance, while these plots also show the path towards these solutions. The most significant finding here that is before the very best configuration is found, candidates in the 90th percentile of performance are found earlier in the search. For example layers 53 and 92 show very good results within the first 100 trails, while the remaining search only improves the performance by around five to ten percent. The baseline ,however, often struggled to find good candidates, with some runs even ending



(a) Layer 53

(b) Layer 92

(c) Layer 107

(d) Layer 59

Figure 5.8 Tuning Runs: Full method against baseline.

at 750 trials without finding the best-performing candidate. A different picture can be seen in layer 107, where the baseline and improved method track each other closely, with the latter having a slight head start, which is kept throughout the run. This can be attributed to the high number of valid configurations, with diverse performance characteristics. Interesting is the outlier behaviour in this plot, where the baseline tends to have outliers that are worse than the average, whereas the improved method has outliers that perform better than the majority. Lastly, layer 59 with its low number of valid candidates requires only one or two epochs to find the best version, while the baseline struggles, only finding the first valid candidate after more than 200 trials, up to 600 trials in the worst case.

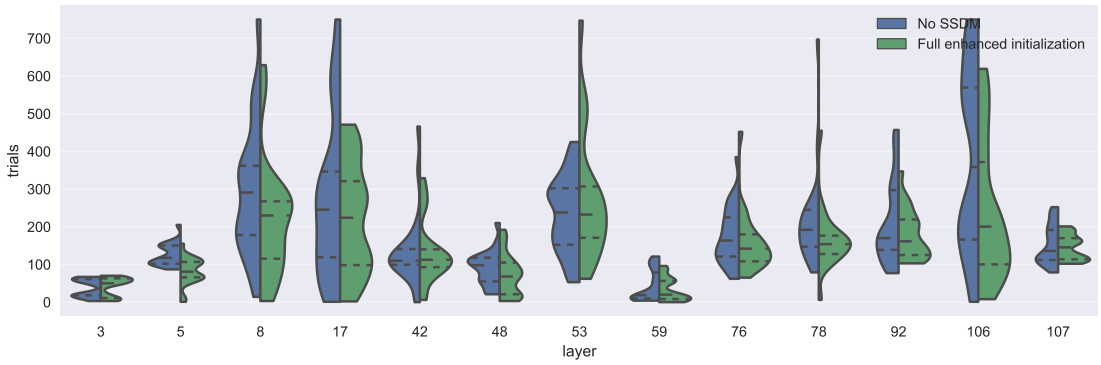


Figure 5.9 Distribution of trials without distance maximization.

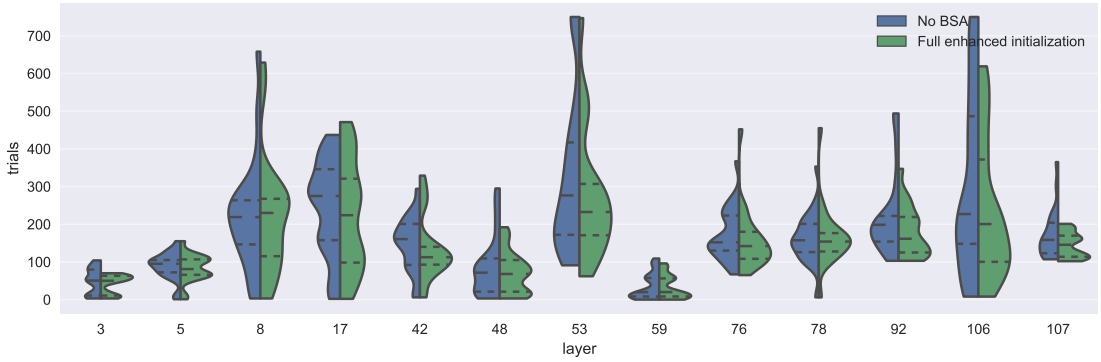


Figure 5.10 Distribution of trials without the simulated annealing bias.

5.3.1.4 Effects of BSA and Distance Maximation

The differences between the full improved method and deactivated distance maximization and BSA are displayed in Figures 5.9 and 5.10. For both, the results confirm what was already discussed in Figure 5.7 and in the relative comparison of Table 5.4 for median convergence time and IQR. Robustness and trials-to-best solution show a slight performance decrease, with layers like 8, 53, 92 or 107 being exceptions from this. No direct correlation between the size of the search space and the observed effects found. It is to be believed, that for large search spaces the total number of candidates seen during presampling is too small to meaningfully impact the overall tuning behaviour. And for small search spaces, the model learns about the workload's characteristics fast enough for the additional methods not to matter, either.

5.3.2 Pre-Trained Performance Models

In the previous experiments, the AutoTVM model used random sampling to start the search process and used an uninitialized model, whereas the presampling

Table 5.5 Comparison of the new initialization method and transfer learning to the AutoTVM baseline.

	Convergence		Robustness	
Layer	New Method	Pre-trained Baseline	New Method	Pre-trained Baseline
3	**0.305**	1.213	**0.345**	0.649
5	**0.460**	2.085	**0.362**	1.914
8	0.433	**0.193**	0.375	**0.027**
17	0.438	**0.388**	0.580	**0.164**
42	0.319	**0.207**	0.177	**0.065**
48	**0.270**	0.501	**0.333**	0.448
53	0.480	**0.365**	0.333	**0.227**
59	**0.120**	0.937	**0.861**	5.386
76	**0.460**	0.953	0.251	**0.171**
78	**0.447**	1.189	**0.176**	0.542
92	**0.492**	1.618	**0.269**	0.615
106	**0.403**	1.487	**0.631**	1.226
107	0.786	**0.624**	0.815	**0.302**
Mean	**0.416**	0.905	**0.424**	0.903

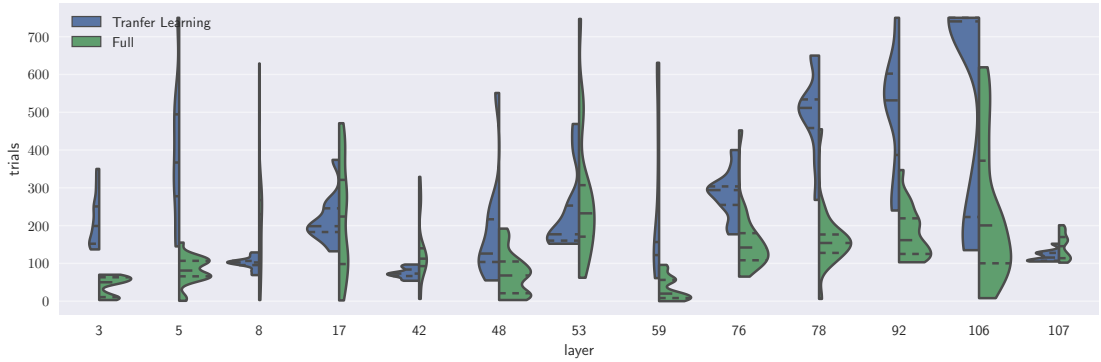
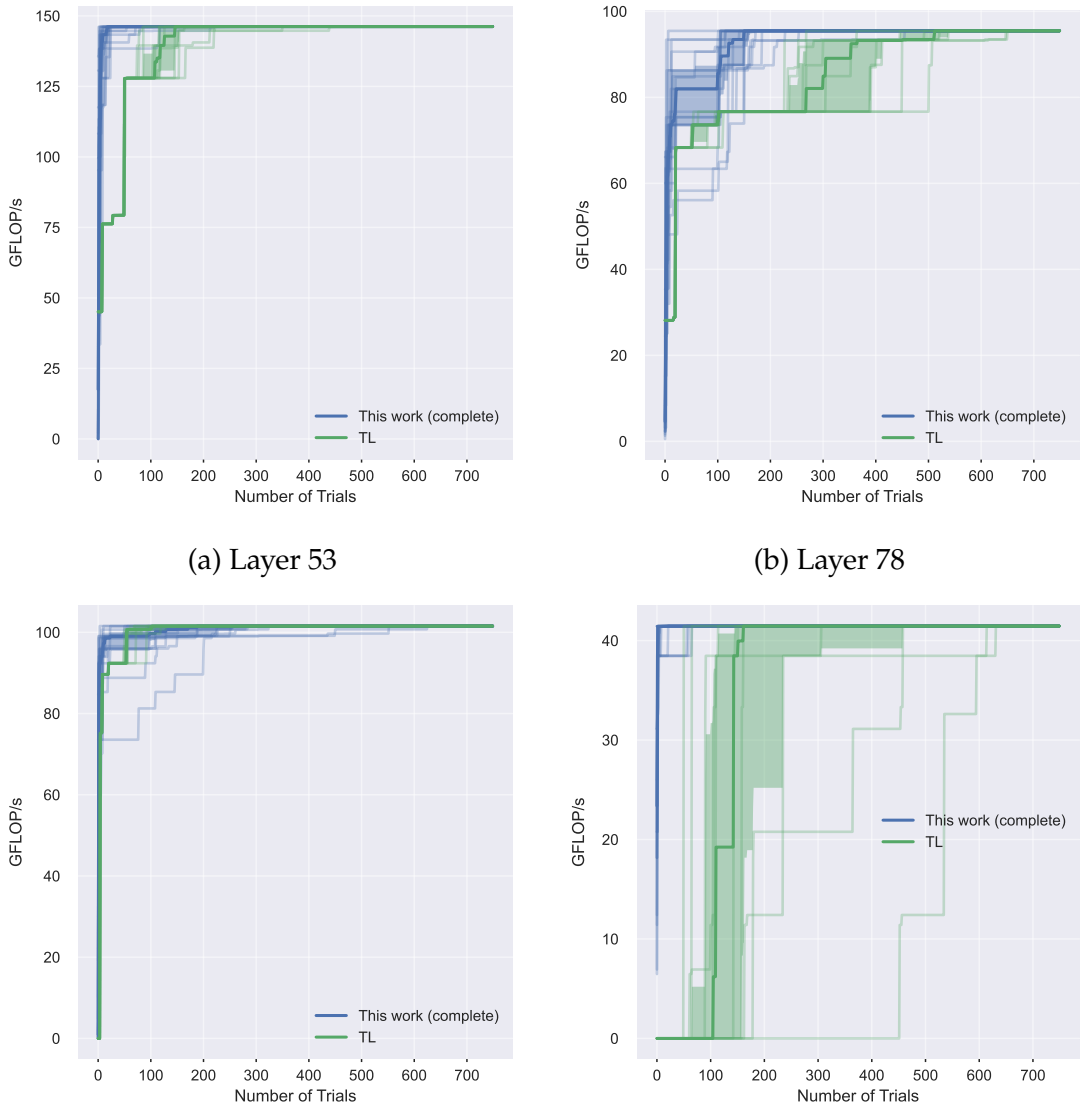


Figure 5.11 Distribution of trials to find the best configuration for each workload for the new method and pre-trained AutoTVM for 10 tuning runs.

method already provided validity information for the start of the tuning process. A third option was already presented in Section 5.1.3.2: features that offer a more workload-agnostic representation of an individual candidate to enable transfer learning in the tuning process. This way, the performance prediction model can be trained on data of other workloads, with the goal to improve the search process. In the following experiments a model trained on the same dataset as in Section 5.1.3.2 is compared against the tuning with presampling. Due to the large overheads during transfer learning, experiments were only repeated 10 times, instead of 20. The latter will still use the *knob* features, as they are cheaper to obtain (see Table 5.6).

It is important to note that during the search with transfer learning the initial model is specialized towards the given workload with every epoch, until only



(a) Layer 53

(b) Layer 78

(c) Layer 8

(d) Layer 59

Figure 5.12 Comparison of presampling against runs with pre-trained models.

data from the current workload is used to train the performance prediction model. In a sense, this is a similar philosophy to the work presented in this Chapter, as both aim to give the model a head start over random sampling.

5.3.2.1 Convergence and Robustness

Table 5.5 compares the convergence time of this work and the transfer-learned tuner against the untrained baseline. Overall, the pre-trained model is only needs 90.5% of the trials the baseline needs to find the best result, however, in almost half the experiments, tuning performance decreased. Five workloads (3,

Table 5.6 The left-hand side and middle show the overhead of knob and context relation features at runtime. This includes feature extraction, simulated annealing and model training on new measurements. The middle and two right-most columns show the initial overhead before the search.

Layer	Knob	Context-relation features	presampling	Sample selection	Pre-training
78	77.8	656.8	416.8	13.4	512.8
8	77.5	1118.7	375.5	13.0	839.9
92	88.4	1139.7	396.0	11.7	653.7

5, 78, 92, 106) with transfer learning perform worse than the baseline, layer 5 by a factor of $\times 2.085$. On the other hand, layers 8 or 42 performed significantly better than the presampling based method by factors of $\times 2.24$ and $\times 1.54$. For the robustness, a similar behaviour is observed. Transfer learning improves the baseline by $\times 1.107$, but has strong variations in both directions. Layers 8 and 42 achieve an exceptional robustness, with the IQR being only 2.7% and 6.5% of the baseline. For layer 59, learning from other models is not successful, as the IQR increases by a factor of $\times 5.386$, although the convergence is slightly better.

5.3.2.2 Individual Workload Analysis

Turning to absolute numbers in Figure 5.11 confirms these observations. Overall, the pre-trained model is consistently over 200 samples in median convergence, with the exceptions of layers 42, 53 barely and 107. Layers 8, 42, 53 and 107 show high robustness and no outliers, while outperforming the presampling method. However, workloads 5, 78, 92 or 106 are the opposite of this. They need over 300, the latter three over 500 samples to converge to the best solution. They also have a low robustness with the outer quartiles reaching up to the sampling limit.

This is also visible in the individual experiments of Figure 5.12. Many runs in layer 78 plateau around 150 to 300 trials, with outliers taking even longer to break 90 GFLOp/s barrier, which explains the large IQR. In layers 53 and, 73 and 8 the initial sampling epoch $\mathcal{E}_0$ performs better than the untrained baseline, usually finding at least one well-performing candidate in the first epoch. For workload 59, the convergence is better than in the untrained baseline, as the search always finds the best candidate, even after 450 of no valids in the beginning. However, the spread over individual runs is much wider, which also leads to the larger IQR.

5.3.3 Tuning Overhead

The previous sections compared the tuning performance by the number of trials, not time-to-solution. This metric makes comparison of different approaches easier and decouples the evaluation from runtime effects. However, in reality, the time spent to find a solution is important. While hardware execution is a major contributor to the overall tuning time, other overheads can increase the absolute runtime. Section 5.1.3.2 already demonstrated that training the performance model from scratch can take up a significant amount of time. Table 5.6 lists the per-run overheads of the different methods. For the baseline with random initialization, the overhead is 0 seconds. The presampling performing the validation of individual candidates for model initialization also creates a significant overhead, while the sample selection process itself is negligible. For layers 78, 8 and 92 presampling plus sample selection is still 82.6, 415.4 and 246 seconds faster than transfer learning. However, the presampling has to be performed for every new workload, whereas the trained model can be reused for any number of workloads.

Feature extraction is faster on knob features than the context relation features by factors of $\times 8.44$, $\times 14.42$ and $\times 12.89$ for the layers in Table 5.6. Knob features just flatten the configuration vector. To use the context relation features, the configuration has to be lowered into a full intermediate code representation before the features can be extracted. This process is costly, and since it has to happen at runtime, it cannot be avoided, like model training. Overall, the presampling outperforms the transfer learning, as the steps prior to the actual tuning are faster. And even if an already trained model exists, the faster feature extraction amortizes the sampling cost.

5.4 Discussion

The experiments in this chapter confirm the hypothesis that a large number of invalid configurations influence the auto-tuning performance and that a tuning process starting with a curated set of initial candidates, balancing valid and invalid configurations finds better solutions faster and more consistently. The baseline starting with uniform sampling of a space with non-uniform solution distribution is outperformed in every experiment. Eventually, the baseline is able to find the best-performing candidate for all layers in the experiment, but not in every individual tuning run. Thus, depending on the initial samples in

$\mathcal{E}_0$, good candidates are found either quickly, or never before the trial limit. The presampling algorithm, on the other hand, always found the best candidate before reaching the trial limit and was more consistent in doing so, leading to the conclusion that the model with presampling is adapting to the search space quicker.

5.4.1 Results

The ability to discard invalid candidates was instilled into the model, without losing the ability to rank well enough to solve the problem, as experiments with larger numbers of valid candidates showed, such as in Figure 5.8c.

Additional modifications to the overall method showed minor improvements over basic presampling. Over all experiments with both distance maximization and BSA, improvements of 6.7 and 5.3 percentage points for convergence and 11.5 and 7.7 percentage points for the IQR were possible. This pales in comparison to the improvements of the curated model itself.

With the right feature representation, it is possible to train a model on the data of previous tuning runs. This process of transfer learning aims to improve the initial performance of the tuning process. Since the performance prediction model is initialized with a rich dataset, the initial random sampling is replaced with a simulated annealing search using this model. The data in Table 5.5 show an overall improvement of roughly ten percentage points over the random baseline. This mean value, however, is deceptive. Looking at individual workloads, transfer learning outperforms even presampling in some experiments, sometimes by a large margin. However, for other workloads convergence and IQR can degrade drastically.

The source of the performance discrepancy can be attributes to several factors. First, there is less control over the dataset itself. In practice, the data for the transfer learning comes from previous tuning runs. This means the data is biased towards valid samples, since ideally only the initial few epochs contain invalid configurations. Further the training set only contains information about a similar but not identical problems. The individuality of each workload with respect to valid and invalid configurations is not yet learned at this point. In contrast, the performance model using presampling only ever learned about the very workload it is used to optimize for. This issue comes down to the question, how similar the training set to the actual problem is and how well the model can generalize. The context relation feature used for transfer learning provides a

more detailed view of a candidate's execution, at the cost of significantly larger feature vectors, compared to the cheaper knob features. And while the *knobs* are a simpler representation, they seem to accurately capture both validity and performance information. The context relation features lose this specificity to gain portability and the experiments showed that for this strategy works well for a subset of the investigated workloads. But the enhanced initialization method showed a better overall generalization ability.

Lastly, the cost of features can be dominating when focussing on the absolute time to perform a tuning process. For real evaluations in this can be critical. The overhead of presampling exists for every individual workload, while the overhead for pre-training the performance model can be amortized across many workloads. However, the extraction of features that can be used to learn across workloads is expensive and cannot be skipped. Ignoring the actual tuning performance, the cost of presampling and the cost of feature extraction need to be weighed against each other.

For real world applications, the process presented in this chapter brings several advantages. The first is more confidence in the selected tuning duration. In practice, we do not know in advance if the solution has been found, thus the search usually runs longer than necessary. With the improved robustness, tuning can end sooner and give greater confidence in the result. Further, the 90th or 95th percentile of performance is often found much earlier than the true best. Often these candidates are sufficient for practical application, which further improves the confidence in the quality of the results. Second, the process is not dependent on existing training data. The presented method only requires data from its own search space, thus it is an approach without overhead for new hardware targets.

5.4.2 General Considerations on Program Optimization

From a more general perspective, auto-tuning and performance optimization in general are a trade-off between data costs, engineering costs and measurement costs. For general purpose systems with an abundance of measurement data, like GPUs in a cloud environment, fully data-driven approach can generate impressive results quickly, at the cost of millions of training examples [43]. This is not always feasible on more specialized devices, like embedded systems. Here, the system is often designed for specific use cases and the measurement campaigns to acquire the necessary data would have to be repeated for each new system. Approaches with higher engineering cost can be a hurdle in some

contexts, like hardware and system design. Fitting the model to hardware updates during development can be time intensive and prone to inaccuracies. While feedback driven approaches are the most portable, a reduction of hardware measurements is the key to bring optimization times to an acceptable level. In this context, the presampling method for search-spaces with a large share of invalid configurations demonstrated significant performance improvements over the baseline. Improving the initialization point is not a novel concept, per se. Algorithms like `kmeans++` [8] already demonstrated the benefits of starting-point optimizations

5.4.3 Related Work

Existing work in the tuning space shows a clear gap to handle invalid tuning candidates. Chameleon[4] works with a reinforcement learning based agent. To reduce hardware sampling time, a single measurement is selected to represent a neighbourhood of similar configurations. This cluster can reduce tuning times drastically. For the case studied in this chapter, the clustering can be difficult to realize. Even when using the improved initialization, the validity of a majority of candidates is not known until it is sampled. Thus, for most candidates we do not immediately know if they represent a region of valid candidates. A candidate could also be invalid himself or used to represent several invalid configurations. Both cases would detriment the tuning quality.

AutoHalide [3] and Ansor [124] create larger search spaces, by not only tuning templates based on parameters like tiling, unrolling or vectorization, but also by dynamically transforming the loop structure during the tuning process. By trying to gauge the performance of high-level loop structures before fully tuning them, they are more refined than UNIT when it comes to dealing with structurally different kernels. Results are only reported for general purpose architectures. However, such a process could also be further developed to target accelerators by describing the space of legal loop structures for an instruction embedding. The necessary information could be provided by the method presented in Chapter 4. However, as of this writing, no such solution is available.

Telamon [10] implements a completely different approach, which avoids invalid configurations by relying on a constraint based, manually crafted hardware model. The constraint program predicts the upper performance bound, while avoiding the construction of invalid configurations. Likewise, DORY [12] optimizes code with a constraint model of the target hardware's memory

hierarchy. This handcrafted approach delivers good performance, however, comes at high engineering cost and requires deep hardware knowledge. This limits the portability of this method.

5.4.4 Outlook

What remains to be explored is the generalization of this method. How well it translates to other hardware architectures is highly dependent on both the architecture itself and code generation. However, since the primary insight is that a more nuanced initialization of the prediction model yields better tuning performance, the method is agnostic to the source of the invalid configuration. If the performance prediction model can interpret validity from the presented features, a generalization is possible.

Chapter 6

Discussion and Outlook

This work set out to evaluate and improve the processes used to offload deep-learning operators, $Conv2D$ in particular but not exclusively, to loop-back style DNN accelerators. Special attention was paid to the problem of how a workload can be computed with sequences of complex instructions, like matrix-multiply. An evaluation of the existing process to achieve this was done in Chapter 3. The findings have been used to implement a novel concept called dataflow embedding, presented in Chapter 4. Section 6.1 discusses and compares these approaches. Future research topics are discussed in Section 6.2.

6.1 Top-Down and Bottom-Up Embedding

In Chapters 3 and 4, two philosophies for embedding have been presented and evaluated. The first philosophy, *Top-Down* is driven by program transformations, or rewrites, in order to find an equivalent version of the original computation in which a pattern matching process can find a suitable embedding location for a hardware intrinsic. In contrast, the dataflow embedding is *Bottom-Up*. It establishes matches between the dataflow of the operator and the intrinsic. From this match, the necessary transformations on loops and data layout can be derived. After reviewing the benefits and drawbacks of each method, this section will discuss how they relate to each other.

Top-Down Benefits: The evaluation in Section 3.4.2 showed how embeddings for variations of the *dense* operator are produced quickly and without a lot of overhead. One reason for this was the high similarity between available

instructions and the workload. Another reason is the relatively low complexity with respect to the number of loops, tensor dimensions and access functions in the workload. Also, this method has a relatively low overhead when describing instructions and workloads, which is similar to existing tools like TVM. Lastly, the method can also match multiple instructions in parallel, reducing the overhead for hardware architectures where multiple instructions are potential candidates for an embedding.

Top-Down Drawbacks: The loop-based workload representation leads to a situation where increasingly complex workloads, like *Conv2D* or *Capsule Routing*, the search space can grow to intractable sizes. Selecting and ordering transformations to manage this complexity is crucial to find embeddings with sensible effort. One issue is the potential to disassemble the workloads into multiple sequential operations. This leads to high combinatorial complexity when it comes to generating different possible loop nesting and operation fusions. The loop-based representation also forces implicit implementation decisions into the problem, like tensor dimension ordering. Resolving the ordering either requires additional transformation or a more complex pattern matching process that accounts for this issue. Adding more transformations is not beneficial, as the evaluation showed diminishing returns with respect to possible solutions. Further, the exploration is only so powerful as the set of existing transformations. This limits the ability explore new possible implementation strategies. For example, Top-Down is limited to one-on-one matching between workload and intrinsic dimensions unless a fusion is performed explicitly before the matching.

Bottom-Up Benefits: Chapter 4 presented a novel approach that enabled the embedding on scalar dataflow level, circumventing the main issues of the Top-Down approach. By embedding on a representation that is detached from loops and access functions, embedding issues related to them are avoided. Mainly, loop-ordering, input and output transposition, access strides or loop selection. Since the concept is based on constraint programming, the embedding space starts with every single possible implementation and is then reduced by applying target-specific constraints to describe a solution space that fits the capabilities of the deployment stack and hardware. The Bottom-Up approach enables exploration of the embedding space without writing transformations, but instead deriving and implementing the transformations based on the solutions provided by the solver. From searching embeddings in a setup with few constraints beyond

dataflow matching, completely new implementation strategies can potentially be found. By principle the method also enables the mapping of multiple workload dimensions to a single intrinsic dimension.

Bottom-Up Drawbacks: The main drawback is the scaling with instruction size, as it increases the number of variables for the constraint problem to solve. Further, only a single instruction can be analysed per run. Lastly, the setup for a specific hardware target is more complex and requires a deeper understanding of both hardware and software, compared to the Top-Down approach.

The choice between the Top-Down and Bottom-Up approach should be considered carefully, depending on the problem to solve. Learnings from the former were integrated into the latter's design, allowing Bottom-Up to focus on the problems not yet resolved by Top-Down matching. Improvements to better capture loop orderings or memory access issues come at the cost of a higher effort to describe a solution space and only embedding a single instruction at a time. This trade-off between usability and ability to solve problems makes Top-Down approaches viable for low complexity workloads or use cases where the transformation complexity is relatively limited. On the other hand, workloads like *Conv2D* can create too many possible choices in the Top-Down method, requiring a more powerful method to handle the details of workload variations.

Despite their differences, both approaches also share properties. As discussed in the introductory chapters, relieving application developers from knowledge about the underlying system while enabling new operators and specialized hardware targets without significant library implementation efforts was a driving motivation for this work. In this respect, both approaches offer solutions. Top-Down achieves this with a systematic creation of a solution space through transformations, in which candidates must be found, while the Bottom-Up system constrains a scalar problem space towards desirable solutions.

Navigating the space of transformation without relying on predefined dimension semantics like *NCHW* is a step forward with respect to supporting new operators and hardware combinations. Chapter 4 demonstrated how this type of information can be found and translated into a set of layout transformations without relying on $NCHW$-like dimension semantics.

It is important to highlight that neither method can be seen as an out-of-the-box solution. Hardware expertise is necessary to create the transformation sequences

and constraint spaces for the respective solutions and couple them with the DNN deployment toolchain, which is by no means a trivial task. How much effort this is depends on the hardware target. For the Top-Down approach, giving a set of transformation that can extract subgraphs matching the available instructions is key. Most transformations are general enough to be used across hardware targets, while sometimes an additional set of hardware-specific transformations might be necessary for individual use cases. While this work generated exhaustive search spaces, this might not always be necessary or possible for practical applications.

In the Bottom-Up approach, the situation is very similar. Many constraints, like the graph-isomorphism or access pattern analysis are reusable, with some hardware specific constraints mixed in for better search efficiency. Section 4.5.2.2 also showed that the Bottom-Up method can generalize towards different hardware configurations without changing the constraint program design. However, a full study on a completely different hardware target was not performed, yet. Especially for significantly larger hardware intrinsics, additional measures might be necessary to keep the runtime manageable.

While both approaches still require hardware knowledge, effort put towards generalizable methods for a specific application domain can be seen as more beneficial as the recurring, manual implementation of libraries or embedding templates for new workloads and hardware.

Like most existing methods, this work has split the problem into strategy choice (data layout and algorithm) and schedule optimization for two reasons. First, the runtime of operators is not necessarily the only goal for strategy selection. As demonstrated in Section 4.5.2 memory footprint, latency, or data movement are often conflicting and need to be addressed on the system or application level. Second, it enables strategy coordination between adjacent operations to enable graph-level optimizations. This flexibility is paid for with higher auto-tuning effort. While this work presented solutions to improve this process for hardware accelerators in Chapter 5, it still is a significant effort to tune enough implementations for a global graph optimization.

6.2 Outlook

A key component of making specialized hardware accelerators usable is accessibility through software. The application side of deep learning provides a host of tools, like TensorFlow or PyTorch [2, 72] to democratize the technology. However, creating efficient inference kernels from these high-level DNN descriptions for

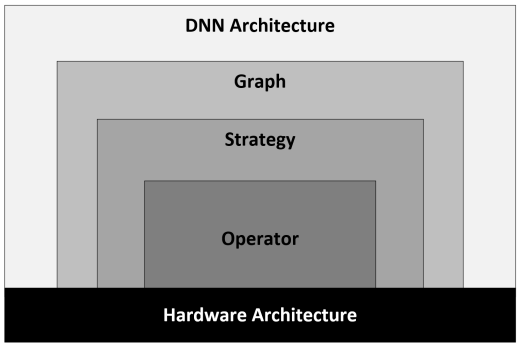


Figure 6.1 Layers of hardware-aware DNN optimization.

non-commodity hardware targets is challenging. Major contributors to solving this problem are TVM and MLIR. Both aim to provide developers with the infrastructure to target their custom devices, while controlling the full deployment stack. While TVM is fully geared towards DNN workloads, MLIR is targeting an application space beyond machine learning. The methods presented in this work can potentially be integrated into TVM. Both, the embedding and auto-tuning methods are stand-alone solutions and can be used to deploy DNN kernels to accelerators. This work mostly looked at performance aspects of deployment, but as mentioned above, system integration to achieve usability for users is an important step forward. An integration into one these tools, with a standardized interface could be a next important step.

Figure 6.1 shows the different levels of optimization possible when creating a DNN for specific task on a specific hardware target. Ideally, all decisions take the underlying hardware into account. For any level, from network architecture to individual operators, feed-back driven approaches proved to be very successful in finding the best option, except for some libraries on highly available targets like GPGPUs at the operator and strategy level. This results in multiple, nested optimization loops, starting with network architecture search applications over graph level optimization down to focus of this dissertation, the strategy and operator level. Each layer then adds to the runtime of layer above, increasing the computation effort to solve the problem at a specific layer. In turn, this decreases the general usability and availability of such solutions. Therefore, the deployment problem cannot be considered solved, yet. The methods presented in this work only looked at individual operators.

DNN Architecture Recently, hardware-aware network architecture search (HW-NAS) gained more research attention [97, 28, 49, 96, 60]. In HW-NAS, Pareto

optimal solutions between application performance, like accuracy, and hardware metrics like latency or number of parameters are searched. The training of networks is often time-intensive and the evaluation time for individual architectures should not be prolonged by the latency evaluation, which could be the case with extensive auto-tuning or embedding operations. While proxy metrics like the number of multiply-and-accumulate operations (MACs) can be used to improve runtime, they often fall short to capture specific hardware behaviour. Also, a reduced number of MACs does not guarantee to materialize a similar gain in latency [97]. A solution that allows the integration of both, HW-NAS and lower-level optimization in the context of deployment to specialized accelerators would be a valuable step forward.

Graph-Level Optimization Graph-level optimizations that fuse multiple sequential or parallel operators demonstrated significant performance benefits, especially for large DNNs [48, 29]. But, as pointed out by the authors of [48], the time to evaluate the performance such of a subgraph is crucial to make the process viable in practice. The relatively time-consuming embedding and optimization processes presented in this work would prevent tools like TASO to optimize DNN for specialized accelerators dynamically, or at least significantly increase the runtime.

One solution for this problem comes at larger timescales. When the same hardware is used long enough, solutions found once can be cached and reused on demand, which in turn makes longer, more thorough optimization runs more beneficial. In situations where hardware-software co-design is practised and the hardware changes frequently, this time scaling becomes less efficient. Improving the time to solution is therefore still a desirable property for future research on DNN deployment.

A next step could be the integration of Bottom-Up's strong embedding capabilities into a tuning framework. A possible candidate would be Ansor, as it already provides optimization processes for high-level loop structure decisions. The simplest way would be to provide Ansor with a pre-determined set of possible strategies, which are described through a constrained set of high-level loop structure decisions. This would avoid the necessity to select and fully tune a set of strategies and choosing from them afterwards, but instead have the tuning dynamically determine the best strategy for a given operator.

Strategy Selection An important aspect of automatic strategy generation are the necessary data-layout transformations. In this work, a rule-based mapping between matched patterns and transformations was used. This approach fits well with existing deep learning frameworks, where a set of general purpose transformation like `reorder`, `slice`, `reshape` or `gather` are already implemented. However, more complex rewrite operations like stencil unrolling are not yet supported natively and implementations using `gather` have a severe performance penalty. Automatically generating high-performance code for complex memory transformations, directly from the embedding results would be a logical next step to improve the overall system performance. The evaluation for VTA only considered fully software-driven rewrite operations. Hardware accelerators that allow for more complex memory access on the hardware, for example through a DMA and a scratch-pad memory, would also benefit from dedicated load and store routines.

Operator Optimization Automatic performance for computationally intensive kernels will probably remain an active topic of interest for the time to come. While a range of different techniques has been proposed, all of them are somewhere in the triangle of engineering, data, and measurement effort. And with the heterogeneous hardware and application environment, even inside deep learning field, many of them have their merit. However, creating tools for users with the same stability and usability as Tensorflow or PyTorch still requires research and engineering effort.

Hardware Architecture Support for more heterogeneous platforms can be a direction for future research, like the PULP platform [25]. In the current state, a full operator is offloaded to an accelerator. How to offload different operators that can run in parallel to different parts of a system is an open optimization problem. Cooperative processing of one operator on multiple compute resources in system would be interesting approach to support a wider range of workloads and improve utilization.

Another topic not discussed in this work are emerging technologies for accelerators. For example, in-memory compute engines are attractive for the data-intensive computations in DNNs, as they can significantly reduce the energy expenditure for data movements between memory hierarchies. They are based on non-volatile memory (NVM) and can operate on analog circuits [91, 92, 86] or digital values [7, 19]. Potentially new deployment challenges arise, since they

are fundamentally different from architectures that load values from, and store results to a global memory.

Chapter 7

Conclusion

This work set out to investigate how neural network operators, and convolutions in particular, can be offloaded to accelerators providing complex instructions like matrix multiplication. It was motivated by the usability gap between specialized hardware targets and the large number of different DNN operators. After investigating the complexity of transformation-driven solutions, a novel approach was suggested. By solving the embedding with dataflow matching between workload and intrinsic, the necessary data layout and program transformation can be derived and code generated without the need to create several different candidates which might not match the hardware. For example, the split and fusion of multiple workload dimensions can be determined by the dataflow embedding. No previous approach demonstrated the ability to find such a transformation in a deterministic way. Similarly, no speculation is necessary to determine if rewrites like image packing or stencil unrolling will enable an embedding. After detecting the necessary data layout and program transformations for the embedding, efficient code generation for accelerators was investigated. Through hardware aware initialization of the auto-tuning process, the tuning time and robustness have been improved significantly.

The main challenges in mapping operators to hardware accelerators are the necessary data layout, specific loop structures and memory access functions. On the loop-level, the difficulty of this task can grow exponentially with the operator complexity, often with diminishing returns on the solution space. Using the right combination of transformations is crucial to the success. But the evaluation also showed that *dense*-like operators can often be embedded quickly with such an approach and do not warrant more sophisticated method. This is caused by

the relatively low complexity regarding loops and access functions, as well as hardware intrinsics often being very close to these operations. However, for the different variations of convolutions, the loop-level approach does not scale well.

Through analysis of the dataflow, enabled by polyhedral program representation and constraint programming, solutions for operators of the convolution family are produced reliably. From this information, the set of transformations necessary for an offloading is derived. While this still requires explicit program transformations, instead of direct code generation, the transformations are now specifically matched to the presented workloads and hardware abilities. With respect to performance, the results on different classes of convolutions are mixed. In some cases, the raw operator speed-up is diminished by the more complex data-layout transformations, which are implemented inefficiently. And while first experiments on a modified hardware configuration were promising, a full evaluation on a different hardware architecture is an open objective. Neither the polyhedral IR, nor the constraint programming are novel in the domain of accelerator compilers, per se. But this work combined them in a novel way to solve the particular problem of embedding instructions into DNN kernels.

This is complemented by a technique to reduce the optimization time in ML-driven auto-tuning. The feedback driven algorithms used in AutoTVM and similar optimizers that rely on ML-models for guidance either start from scratch for each workload or with an already initialized model, trained on data from other tuning runs. In both cases, the initial phase of the search is decisive on how well rest of tuning process shapes up. A hardware-aware initialization of the tuning algorithm improves the robustness of the search and reduces the tuning time significantly. The improved initialization method curates a set of measurement samples from both, valid and invalid candidates. This makes the process portable to other targets, as the underlying ML-Model learns to distinguish between the classes and then operates independently in the rest of the search process. Evaluation on workloads beyond $Conv2D$ is also still open. However, in the current ML domain, convolutions are the most complex operators. While Transformer networks are also computationally intensive, they mainly consist of *dense*-like operations, where the embedding itself is not as challenging.

The separation of strategy and scheduling has benefits and drawbacks. For many convolutions, strategy generation is straight forward and the scheduling is the main challenge for the deployment. In cases where the hardware and workloads are not a direct match, the strategy generation becomes a more

important part of the deployment process. This comes at the cost of prolonged deployment times due to the search for multiple strategies and their tuning. Especially in scenarios where the workload will deploy on resource constrained devices, the memory and runtime improvements offered by different strategies is an argument for the prolonged compilation time. In other situations where deployment time is of the essence, like HW-NAS, this type of code generations can be a potential bottleneck for the rest of the application. Resolving the tension between fast and best deployment is a challenge for future research.

This work contributes a study on code generation and optimization techniques in the context of DL operators and their deployment on hardware that offer complex intrinsics. As one of the first works in this area, it looks at the issue of data layout and program structure in a unified way. By offering code generation and optimization techniques that, by principle, are portable to different system architectures, avenues for further work on the entire hardware and software stack for DL have been opened. Both, vertical integration into the deployment stack for a specific hardware and horizontal scaling towards the support for more hardware targets provide interesting challenges for future research projects.

Acknowledgements

First and foremost I would like to thank my supervisor Prof. Dr. Holger Fröning. The privilege of his guidance over last years made this dissertation possible. His guidance helped me to grow as a researcher and as a person throughout every stage of this process.

I would also like to thank my advisors at Bosch Research, Dr. Jo Pletinckx and Axel Acosta. Jo was the initiator and driving mind behind this dissertation project and without him, it would not exist. He also helped me consistently to navigate the maze or research, engineering and bureaucracy to reach my goals. Axel was the most critical, insightful, and honest sparring partner one can wish for. Our spirited discussions and whiteboard sessions contributed to this work in a fundamental way.

Of course, this gratitude extends all other Bosch colleagues and fellow PhD candidates and students. Your support and advice greatly helped me in the completion of this project.

Likewise, I have to thank the other PhD candidates at Holger's *Computing Systems Group*. Although we only met a few times, your were always welcoming and supportive. I wish you all the best.

And last but not least, I have to thank my family and friends. You were my fellowship during this journey. The moral support and patience you bless me with is endless, and it is greatly appreciated.

References

[1] Xla - tensorflow, compiled, 2017. accessed: 11.2019.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Operating Systems Design and Implementation (OSDI'16)* (2016), USENIX Association, pp. 265–283.

[3] Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG) 38*, 4 (2019), 1–12.

[4] Ahn, B. H., Pilligundla, P., Yazdanbakhsh, A., and Esmaeilzadeh, H. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *International Conference on Learning Representations (ICLR '20)* (2020).

[5] Aho, A. V., and Corasick, M. J. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18*, 6 (June 1975), 333–340.

[6] Anderson, A., Vasudevan, A., Keane, C., and Gregg, D. High-performance low-memory lowering: Gemm-based algorithms for dnn convolution. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (2020), pp. 99–106.

[7] Ando, K., Ueyoshi, K., Orimo, K., Yonekawa, H., Sato, S., Nakahara, H., Takamaeda-Yamazaki, S., Ikebe, M., Asai, T., Kuroda, T., and Motomura, M. Brein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 tops at 0.6 w. *IEEE Journal of Solid-State Circuits 53*, 4 (2018), 983–994.

[8] Arthur, D., and Vassilvitskii, S. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (USA, 2007), SODA '07, Society for Industrial and Applied Mathematics, p. 1027–1035.

[9] Bastoul, C. Improving data locality in static control programs, 2004.

[10] Beaugnon, U., Pouille, A., Pouzet, M., Pienaar, J., and Cohen, A. Optimization Space Pruning without Regrets. In *International Conference on Compiler Construction (CC '17)* (Feb. 2017), ACM Press, pp. 34–44.

[11] Bielecki, W., and Palkowski, M. Tiling of arbitrarily nested loops by means of the transitive closure of dependence graphs. *International Journal of Applied Mathematics and Computer Science (AMCS) Vol. 26*, 4 (December 2016).

[12] Burrello, A., Garofalo, A., Bruschi, N., Tagliavini, G., Rossi, D., and Conti, F. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers* (2021).

[13] Chaudhuri, S., and Hetzel, A. Sat-based compilation to a non-vonneumann processor. In *2017 IEEE/ACM International Conference on Computer-Aided Design, (ICCAD'17)* (2017), pp. 675–682.

[14] Chellapilla, K., Puri, S., and Simard, P. High Performance Convolutional Neural Networks for Document Processing. In *10th International Workshop on Frontiers in Handwriting Recognition* (Oct. 2006).

[15] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architectural Support for Programming Languages and Operating Systems, (ASPLOS '14)* (2014), pp. 269–284.

[16] Chen, T., and Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2016), KDD '16, Association for Computing Machinery, p. 785–794.

[17] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *Conference on Operating Systems Design and Implementation (OSDI'18)* (2018), OSDI '18, USENIX Association. *ArXiv* 1802.04799 (2018).

[18] Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In *32nd International Conference on Neural Information Processing Systems* (2018), NIPS'18, Curran Associates Inc., pp. 3393–3404.

[19] Chen, X., Yin, X., Niemier, M., and Hu, X. S. Design and optimization of fefet-based crossbars for binary convolution neural networks. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)* (2018), pp. 1205–1210.

[20] Chen, Y.-H., Emer, J., and Sze, V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *43rd International Symposium on Computer Architecture (ISCA '16)* (2016), IEEE Press, pp. 367–379.

[21] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[22] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *ArXiv 1410.0759* (2014).

[23] Chollet, F. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (July 2017).

[24] Cong, J., and Wang, J. Polysa: Polyhedral-based systolic array auto-compilation. In *International Conference on Computer-Aided Design* (2018), ICCAD '18, ACM.

[25] Conti, F., Rossi, D., Pullini, A., Loi, I., and Benini, L. Pulp: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision. *Journal of Signal Processing Systems* (2016).

[26] Dave, S., Kim, Y., Avancha, S., Lee, K., and Shrivastava, A. Dmazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Trans. Embed. Comput. Syst. 18*, 5s (Oct. 2019).

[27] Dukhan, M. The indirect convolution algorithm, 2019.

[28] Elsken, T., Metzen, J. H., and Hutter, F. Efficient multi-objective neural architecture search via lamarckian evolution. In *International Conference on Learning Representations* (2019).

[29] Fang, J., Shen, Y., Wang, Y., and Chen, L. Optimizing dnn computation graph using graph substitutions. *Proceedings of the VLDB Endowment 13*, 12 (2020), 2734–2746.

[30] Feautrier, P. Some efficient solutions to the affine scheduling problem: Part i. one-dimensional time. *International Journal of Parallel Programming 21*, 5 (October 1992), 313–348.

[31] Feautrier, P. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming 21* (1992), 389–420. 10.1007/BF01379404.

[32] Feld, D., Soddemann, T., Jünger, M., and Mallach, S. Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques* (Berlin, Germany, January 2013), A. Größlinger and L.-N. Pouchet, Eds., pp. 45–54.

[33] Friedman, J. H. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[34] Garofalo, A., Rusci, M., Conti, F., Rossi, D., and Benini, L. Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 378*, 2164 (Dec 2019), 20190155.

[35] Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D., Henry, G., Pabst, H., and Heinecke, A. Anatomy of high-performance deep learning convolutions on simd architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (2018), IEEE Press.

[36] Georganas, E., Banerjee, K., Kalamkar, D., Avancha, S., Venkat, A., Anderson, M., Henry, G., Pabst, H., and Heinecke, A. Harnessing deep learning via a single building block. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2020), pp. 222–233.

[37] Georganas, E., Kalamkar, D., Avancha, S., Adelman, M., Anderson, C., Breuer, A., Bruestle, J., Chaudhary, N., Kundu, A., Kutnick, D., Laub, F., Md, V., Misra, S., Mohanty, R., Pabst, H., Ziv, B., and Heinecke, A. Tensor processing primitives: A programming abstraction for efficiency and portability in deep learning workloads. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2021), SC '21, Association for Computing Machinery.

[38] Ghodrat, M. A., Givargis, T., and Nicolau, A. Equivalence checking of arithmetic expressions using fast evaluation. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (2005), CASES '05, Association for Computing Machinery.

[39] Gomes, C. P., and Selman, B. Algorithm portfolios. In *Artificial Intelligence* (2001), vol. 126, pp. 43–62.

[40] Grosser, T., Cohen, A., Kelly, P. H., Ramanujam, J., Sadayappan, P., and Verdoolaege, S. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *GPGPU-6* (2013), ACM, pp. 24–31.

[41] Hagedorn, B., Lenfers, J., Kundefinedhler, T., Qin, X., Gorlatch, S., and Steuwer, M. Achieving high-performance the functional way: A functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang. 4*, ICFP (aug 2020).

[42] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 770–778.

[43] Hegde, K., Tsai, P.-A., Huang, S., Chandra, V., Parashar, A., and Fletcher, C. W. Mind mappings: Enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New

York, NY, USA, 2021), ASPLOS 2021, Association for Computing Machinery, p. 943–958.

[44] Heinecke, A., Henry, G., Hutchinson, M., and Pabst, H. Libxsmm: Accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), SC '16, IEEE Press.

[45] Hoffmann, C. M., and O'Donnell, M. J. Pattern matching in trees. *J. ACM 29*, 1 (Jan. 1982), 68–95.

[46] Hurkat, S., and Martínez, J. F. Vip: A versatile inference processor. In *IEEE International Symposium on High Performance Computer Architecture (HPCA '19)* (2019), pp. 345–358.

[47] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093* (2014).

[48] Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. 47–62.

[49] Jiang, J., Han, F., Ling, Q., Wang, J., Li, T., and Han, H. Efficient network architecture search via multiobjective particle swarm optimization based on decomposition. *Neural Networks 123* (2020), 305–316.

[50] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), pp. 1–12.

[51] Karp, R. M., Miller, R. E., and Winograd, S. The organization of computations for uniform recurrence equations. *Journal of the ACM 14*, 3 (1967), 563–590.

[52] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. *Science 220*, 4598 (1983), 671–680.

[53] Kong, M., Veras, R., Stock, K., Franchetti, F., Pouchet, L.-N., and Sadayappan, P. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation* (2013), ACM, pp. 127–138.

[54] Krause, E. F. *Taxicab geometry: An adventure in non-Euclidean geometry*. Courier Corporation, 1986.

[55] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Red Hook, NY, USA, 2012), NIPS'12, Curran Associates Inc., p. 1097–1105.

[56] Larossa, J., and Valiente, G. Constraint satisfaction algorithms for graph pattern matching. In *Mathematical Structures in Computer Science* (2002), vol. 12, Cambridge University Press, pp. 403–422.

[57] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. Mlir: A compiler infrastructure for the end of moore's law. *ArXiv 2002.11054* (2020).

[58] Lavin, A., and Gray, S. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 4013–4021.

[59] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw. 5*, 3 (Sept. 1979), 308–323.

[60] Li, C., Yu, Z., Fu, Y., Zhang, Y., Zhao, Y., You, H., Yu, Q., Wang, Y., and Lin, Y. Hw-nas-bench:hardware-aware neural architecture search benchmark, 2021.

[61] Liu, S., Du, Z., Tao, J., Han, D., Luo, T., Xie, Y., Chen, Y., and Chen, T. Cambricon: An instruction set architecture for neural networks. In *43rd International Symposium on Computer Architecture* (2016), ISCA '16, IEEE Press, p. 393–405.

[62] Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., and Wang, Y. Optimizing CNN model inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 1025–1040.

[63] Liu, Y., Wang, Y., Yu, R., Li, M., Sharma, V., and Wang, Y. Optimizing CNN model inference on cpus. In *USENIX Annual Technical Conference* (July 2019), USENIX Association, pp. 1025–1040.

[64] Lücke, M., Steuwer, M., and Smith, A. Integrating a functional pattern-based ir into mlir. In *Proceedings of the 30th ACM SIGPLAN International*

Conference on Compiler Construction (New York, NY, USA, 2021), CC 2021, Association for Computing Machinery, p. 12–22.

[65] Mogers, N., Smith, A., Vytiniotis, D., Steuwer, M., Dubach, C., and Tomioka, R. Towards mapping lift to deep neural network accelerators. In *Workshop on Emerging Deep Learning Accelerators (EDLA' 19)* (2019).

[66] Moore, R. E. *Interval analysis*. Prentice-Hall, 1966.

[67] Moreau, T., Chen, T., Vega, L., Roesch, J., Yan, E., Zheng, L., Fromm, J., Jiang, Z., Ceze, L., Guestrin, C., and Krishnamurthy, A. A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro 39* (2019).

[68] Mullapudi, R. T., and Bondhugula, U. Tiling for dynamic scheduling. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques* (Vienna, Austria, January 2014), S. Rajopadhye and S. Verdoolaege, Eds.

[69] Nelson, G., and Oppen, D. C. Fast decision procedures based on congruence closure. *J. ACM 27*, 2 (apr 1980), 356–364.

[70] Nowatzki, T., Sartin-Tarm, M., De Carli, L., Sankaralingam, K., Estan, C., and Robatmili, B. A general constraint-centric scheduling framework for spatial architectures. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013), PLDI '13, ACM, pp. 495–506.

[71] Parashar, A., Raina, P., Shao, Y. S., Chen, Y., Ying, V. A., Mukkara, A., Venkatesan, R., Khailany, B., Keckler, S. W., and Emer, J. Timeloop: A systematic approach to dnn accelerator evaluation. In *International Symposium on Performance Analysis of Systems and Software (ISPASS '19)* (2019), pp. 304–315.

[72] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32* (2019), Curran Associates, Inc., pp. 8024–8035.

[73] Pop, S., Cohen, A., and Silber, G.-A. Induction variable analysis with delayed abstractions. In *High Performance Embedded Architectures and Compilers* (2005), T. Conte, N. Navarro, W.-m. W. Hwu, M. Valero, and T. Ungerer, Eds., Springer Berlin Heidelberg, pp. 218–232.

[74] Ragan-Kelley, J., Adams, A., Sharlet, D., Barnes, C., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM 61*, 1 (Dec. 2017), 106–115.

[75] Rieber, D., Acosta, A., and Fröning, H. Joint program and layout transformations to enable convolutional operators on specialized hardware based on constraint programming. *ACM Transactions Architecture Code Optimization 19*, 1 (Dec 2021).

[76] Rieber, D., and Fröning, H. Search space complexity of iteration domain based instruction embedding for deep learning accelerators. In *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning* (2020), J. Gama, S. Pashami, A. Bifet, M. Sayed-Mouchawe, H. Fröning, F. Pernkopf, G. Schiele, and M. Blott, Eds., Springer International Publishing.

[77] Rieber, D., Reiber, M., Bringmann, O., and Fröning, H. Hw-aware initialization of dnn auto-tuning to improve exploration time and robustness. In *2022 Workshop on Accelerated Machine Learning (AccML) @ HiPEAC2022*.

[78] Robbins, H., and Monro, S. A Stochastic Approximation Method. *The Annals of Mathematical Statistics 22*, 3 (1951), 400 – 407.

[79] Roesch, J., Lyubomirsky, S., Weber, L., Pollock, J., Kirisame, M., Chen, T., and Tatlock, Z. Relay: A new ir for machine learning frameworks. In *International Workshop on Machine Learning and Programming Languages* (2018), MAPL '18, ACM, pp. 58–68.

[80] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1988), POPL '88, Association for Computing Machinery, p. 12–27.

[81] Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., Montgomery, J., Maher, B., Nadathur, S., Olesen, J., Park, J., Rakhov, A., Smelyanskiy, M., and Wang, M. Glow: Graph lowering compiler techniques for neural networks, 2019.

[82] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature 323*, 6088 (Oct. 1986), 533–536.

[83] Sabour, S., Frosst, N., and Hinton, G. E. Dynamic routing between capsules. In *Advances in Neural Information Processing Systems (NIPS '19)* (2017), I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc., pp. 3856–3866.

[84] Schindler, G., Mücke, M., and Fröning, H. Linking application description with efficient simd code generation for low-precision signed-integer gemm. In *Euro-Par 2017: Parallel Processing Workshops* (2018), Springer International Publishing, pp. 688–699.

[85] Schreiber, R., and Dongarra, J. J. Automatic blocking of nested loops. Tech. rep., 1990.

[86] Shafiee, A., Nag, A., Muralimanohar, N., Balasubramonian, R., Strachan, J. P., Hu, M., Williams, R. S., and Srikumar, V. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture* (2016), ISCA '16, IEEE Press, p. 14–26.

[87] Shen, J., Huang, Y., Wang, Z., Qiao, Y., Wen, M., and Zhang, C. Towards a uniform template-based architecture for accelerating 2d and 3d cnns on fpga. In *International Symposium on Field-Programmable Gate Arrays* (2018), FPGA '18, ACM, pp. 97–106.

[88] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition, 2014.

[89] Smith, G. H., Liu, A., Lyubomirsky, S., Davidson, S., McMahan, J., Taylor, M., Ceze, L., and Tatlock, Z. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (New York, NY, USA, 2021), MAPS 2021, Association for Computing Machinery, p. 21–31.

[90] Smith, J. E. Decoupled access/execute computer architectures. *SIGARCH Compututer Architecture News* (1982).

[91] Soliman, T., Laleni, N., Kirchner, T., Müller, F., Shrivastava, A., Kämpfe, T., Guntoro, A., and Wehn, N. Felix: A ferroelectric fet based low power mixed-signal in-memory architecture for dnn acceleration. *ACM Trans. Embed. Comput. Syst.* (mar 2022). Just Accepted.

[92] Song, L., Qian, X., Li, H., and Chen, Y. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), pp. 541–552.

[93] Sotoudeh, M., Venkat, A., Anderson, M., Georganas, E., Heinecke, A., and Knight, J. Isa mapper: A compute and hardware agnostic deep learning compiler. In *Conference on Computing Frontiers* (2019), CF '19, ACM, pp. 164–173.

[94] Steuwer, M., Remmelg, T., and Dubach, C. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Code Generation and Optimization* (2017), CGO '17, IEEE Press, pp. 74–85.

[95] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions, 2014.

[96] Turner, J., Crowley, E. J., and O'Boyle, M. F. P. Neural architecture search as program transformation exploration. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS 2021, Association for Computing Machinery, p. 915–927.

[97] Łukasz Dudziak, Chau, T., Abdelfattah, M. S., Lee, R., Kim, H., and Lane, N. D. Brp-nas: Prediction-based nas using gcns, 2021.

[98] Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[99] Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y. Fast convolutional nets with fbfft: A gpu performance evaluation, 2015.

[100] Vasilache, N., Meister, B., Baskaran, M., and Lethin, R. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT* (Paris, France, January 2012).

[101] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *ArXiv 1802.04730* (2018).

[102] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *ArXiv 1802.04730* (2018).

[103] Vasudevan, A., Anderson, A., and Gregg, D. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2017), pp. 19–24.

[104] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2017.

[105] Verdoolaege, S. Presburger formulas and polyhedral compilation, 2016.

[106] Verdoolaege, S., and Grosser, T. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)* (Paris, France, January 2012).

[107] Verdoolaege, S., Guelton, S., Grosser, T., and Cohen, A. Schedule trees. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques* (Vienna, Austria, January 2014), S. Rajopadhye and S. Verdoolaege, Eds.

[108] Verdoolaege, S., Juega, J. C., Cohen, A., Gómez, J. I., Tenllado, C., and Catthoor, F. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization 9*, 4 (January 2013), 54:1–54:23.

[109] Vogel, S., Raghunath, R. B., Guntoro, A., Van Laerhoven, K., and Ascheid, G. Bit-shift-based accelerator for cnns with selectable accuracy and throughput. In *2019 22nd Euromicro Conference on Digital System Design (DSD)* (2019), pp. 663–667.

[110] Wang, J., Guo, L., and Cong, J. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. 93–104.

[111] Weng, J., Jain, A., Wang, J., Wang, L., Wang, Y., and Nowatzki, T. Unit: Unifying tensorized instruction compilation. In *Code Generation and Optimization (CGO'21)* (2021), Association for Computing Machinery.

[112] Willsey, M., Nandi, C., Wang, Y. R., Flatt, O., Tatlock, Z., and Panchekha, P. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang. 5*, POPL (jan 2021).

[113] Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 5987–5995.

[114] Xie, S., Kirillov, A., Girshick, R., and He, K. Exploring randomly wired neural networks for image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (October 2019).

[115] Yang, X., Gao, M., Liu, Q., Setter, J. O., Pu, J., Nayak, A., Bell, S. E., Cao, K., Ha, H., Raina, P., Kozyrakis, C., and Horowitz, M. Interstellar: Using halide's scheduling language to analyze dnn accelerators, 2020.

[116] Yang, Y., Phothilimthana, P., Wang, Y., Willsey, M., Roy, S., and Pienaar, J. Equality saturation for tensor graph superoptimization. In *Proceedings of Machine Learning and Systems* (2021), A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, pp. 255–268.

[117] Yu, F., and Koltun, V. Multi-scale context aggregation by dilated convolutions. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings* (2016), Y. Bengio and Y. LeCun, Eds.

[118] Zadrozny, B. Learning and evaluating classifiers under sample selection bias. In *Proceedings of the twenty-first international conference on Machine learning* (2004), p. 114.

[119] Zampelli, S., Deville, Y., and Solnon, C. Solving subgraph isomorphism problems with constraint programming. In *Constraints* (July 2010), vol. 15 of *3*, Springer Verlag, pp. 327–353.

[120] Zampelli, S., Deville, Y., Solnon, C., Sorlin, S., and Dupont, P. Filtering for subgraph isomorphism. In *Principles and Practice of Constraint Programming (CP '07)* (2007), pp. 728–742.

[121] Zerrell, T., and Bruestle, J. Stripe: Tensor compilation via the nested polyhedral model. *CoRR abs/1903.06498* (2019).

[122] Zhang, X., Zhou, X., Lin, M., and Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices, 2017.

[123] Zhao, J., Li, B., Nie, W., Geng, Z., Zhang, R., Gao, X., Cheng, B., Wu, C., Cheng, Y., Li, Z., Di, P., Zhang, K., and Jin, X. Akg: Automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2021), PLDI 2021, Association for Computing Machinery, p. 1233–1248.

[124] Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. Ansor: Generating high-performance tensor programs for deep learning. In *Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 863–879.

[125] Zlateski, A., Jia, Z., Li, K., and Durand, F. The anatomy of efficient fft and winograd convolutions on modern cpus. In *Proceedings of the ACM International Conference on Supercomputing* (New York, NY, USA, 2019), ICS '19, Association for Computing Machinery, p. 414–424.

[126] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), pp. 8697–8710.