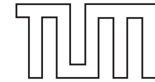
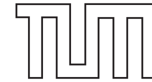


Technical University of Munich
TUM School of Computation, Information and Technology

Code Optimization and Generation of Machine Learning and Driver Software for Memory-Constrained Edge Devices

Rafael Christopher Stahl



Technical University of Munich
TUM School of Computation, Information and Technology

Code Optimization and Generation of Machine Learning and Driver Software for Memory-Constrained Edge Devices

Rafael Christopher Stahl

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz: Prof. Dr. Andreas Herkersdorf

Prüfende der Dissertation:

1. Priv.-Doz. Dr. Daniel Mueller-Gritschneder
2. Prof. Andreas Gerstlauer, Ph.D.

Die Dissertation wurde am 20.12.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 18.06.2024 angenommen.

Acknowledgment

First, I would like to thank the German Ministry of Education and Research (BMBF) and the taxpayers for funding my research.

I want to thank Prof. Schlichtmann for providing me with the opportunity to pursue my research at his chair.

A special thanks goes to my colleagues for contributing to the great environment during this time.

My deep and heartfelt gratitude goes to my supervisor, Daniel Müller-Gritschneder. Your constant support, exceptional teaching skills and deep expertise were essential for the completion of this thesis.

Finally, I am grateful to my family, partner and friends for their endless encouragement, patience and love throughout this time.

Abstract

Compact electronic devices that interact with the physical world, so-called edge devices, are essential for a wide range of applications that advance efficiency, convenience and well-being. Their cost and power consumption must be low because they are deployed in large quantities and often run on batteries. This constrains the available resources on edge devices tightly, especially for memories, which contribute significantly to cost and power consumption. This motivates the need for more research to optimize software that is deployed on microcontrollers, which are the computer chips at the heart of most edge devices.

The contributions presented in this thesis focus on the code optimization and code generation for the deployment of machine learning tasks on single and multiple cooperating devices as well as on the generation of device drivers at the interface between hardware and software. Modern compilers have largely exhausted the potential for optimization from the high-level programming language to machine code. Therefore, this work explores the translation from a domain-specific representation to the high-level programming language. Code generation automates, simplifies and generalizes this manual software development process.

In this thesis a novel method to partition neural network layers is developed. It could be used to reduce the memory usage of certain machine learning models by up to 76.2% without introducing significant run time overhead. Additionally, this method was used to improve a distributed machine learning inference flow so that it can fully scale memory usage by incorporating more cooperative devices, while reducing communication demand by up to 28.8% compared to previous methods. Automated optimization of driver code was able to reduce the number of memory accesses by 36%, the estimated run time by 52% and the code size by 22%.

Contents

1 Introduction **13**
1.1 Motivation 13
1.1.1 Machine Learning Inference on MCUs 14
1.1.2 Distributed Machine Learning Inference 14
1.1.3 The Interface Between Hardware and Software 14
1.2 Contribution of this Thesis 15
1.2.1 Fused Tiling for Optimized ML Inference 15
1.2.2 Optimized Distributed ML Inference 16
1.2.3 Automated HW/SW Interface Definition and Optimization 16
1.3 Structure of this Thesis and Previous Publications 16

2 Background **19**
2.1 Microcontrollers 19
2.2 Deep Neural Networks 21
2.2.1 Dense Layer 22
2.2.2 Convolutional Neural Networks 23
2.3 Deep Learning Frameworks 25
2.4 Summary 27

3 State of the Art **29**
3.1 Deep Learning Inference on Edge Devices 29
3.2 Fused Tiling 30
3.2.1 Memory-aware Scheduling 33
3.2.2 Memory Layout Planning 33
3.3 Distributed Inference 34
3.4 Driver Software on Edge Devices 35
3.5 Summary 36

4 Fused Tiling for Memory Optimization in DNN Inference **37**
4.1 Introduction 37
4.2 Fused Depthwise Tiling (FDT) 39
4.3 Automated Tiling Exploration 41
4.3.1 Memory-aware Scheduling 41
4.3.2 Memory Layout Planning 44
4.3.3 Block-based Path Discovery 46

4.3.4 Automated Graph Transformation . . . 51
4.3.5 Implementation . . . 52
4.4 Experimental Results . . . 55
4.4.1 Automated Tiling Exploration . . . 55
4.4.2 Fused Depthwise Tiling . . . 56
4.5 Summary . . . 57

5 Optimization of Memory and Communication in Distributed DNN Inference 59
5.1 Motivation . . . 59
5.2 Contribution . . . 61
5.3 Methods for DNN Partitioning . . . 62
5.3.1 Baseline . . . 62
5.3.2 Pipelining . . . 63
5.3.3 Feature Partitioning with FFMT . . . 64
5.3.4 Weight Partitioning with FDT . . . 64
5.4 Optimized DNN Partitioning . . . 68
5.4.1 ILP-based Memory Footprint Minimization . . . 68
5.4.2 ILP-based Communication Optimization for Weight Partitioned Layers 69
5.5 Experimental Results . . . 70
5.5.1 ILP-based Memory Footprint Minimization . . . 71
5.5.2 ILP-based Communication Optimization . . . 71
5.5.3 Evaluation on Raspberry Pi Edge Cluster . . . 72
5.6 Summary . . . 76

6 Hardware/Software Interface Generation and Optimization 77
6.1 Motivation . . . 77
6.2 Contribution . . . 79
6.3 C Language Extension . . . 79
6.3.1 Bit Field Group Definition . . . 80
6.3.2 Hardware Side Effects . . . 81
6.3.3 Behavior Description . . . 83
6.3.4 Implementation . . . 83
6.4 Heuristic Optimization . . . 85
6.4.1 Control-Data-Flow Analysis . . . 85
6.4.2 Bit Field Access Conflict Graph (BFACG) . . . 85
6.4.3 Bit Field Group Simplification . . . 87
6.4.4 Heuristic Algorithm . . . 87
6.5 Automated Code Generation . . . 89
6.6 Experimental Results . . . 91
6.7 Summary . . . 92

7 Conclusion and Outlook 93

Acronyms 95

Bibliography 97

List of Figures **107**

List of Tables **109**

Chapter 1

Introduction

Microcontrollers are ubiquitous in the world today and are expected to become even more important in the future. A microcontroller (or *MCU* for microcontroller unit) is an integrated circuit that contains all the essentials of a programmable computer: memory to store program instructions and working data, at least one processor core that executes instructions, and interfaces to the outside world through peripheral devices. The advantages of MCUs over larger computers are their low size, cost and power consumption. These properties allow to easily embed MCUs into other devices responsible for a raised standard of living, such as medical, industrial, communication, transportation, home and consumer applications [32, 17, 29, 74, 40, 2]. When added to a system that would normally not be programmable, MCUs can improve safety, efficiency, functionality and flexibility. To achieve these improvements, the system requires software, that is, instructions for the MCU to execute. Such software is running in a tightly constrained environment because the available resources in terms of computing capability and amount of memory are extremely limited. The topic of this thesis is the optimization and automated generation of software that is suitable for deployment on resource-constrained devices. Optimization is desired because MCUs are often produced in large quantities and have long lifetimes. Therefore, even small improvements can yield large total savings. While there has been massive research progress since the inception of MCUs and their software, this thesis presents advances of the state-of-the-art in the following specific areas.

- Machine Learning Inference on MCUs
- Distributed Machine Learning Inference
- The Interface between Hardware and Software

1.1 Motivation

The following presents the challenges addressed with the approaches presented in this thesis. Solutions are proposed in the subsequent section.

1.1.1 Machine Learning Inference on MCUs

Machine learning applications on self-sufficient devices offer superior possibilities over cloud computing approaches in terms of communication demand, latency and data privacy. Such devices have a wide range of computation classes, and it was shown that certain machine learning workloads can be performed even on tiny, low-power microcontroller-type devices. MCUs have become sufficiently capable to run machine learning applications such as keyword spotting, visual wake-up, anomaly detection or radar-based gesture recognition. In these applications, a machine learning model that has previously been trained on a larger machine is deployed and computes a prediction for new input, a process also called *inference*. Machine learning inference is heavily constrained by the limited resources on MCUs, which spawned a large field of research known as *TinyML* or *Extreme Edge AI* [104, 55]. This thesis will focus on reducing the memory demand of TinyML applications. There are a number of TinyML solutions that tackle this issue by sacrificing some model accuracy to reduce the memory demand, such as quantization, pruning and neural architecture search. Although there also exist methods that are able to reduce memory demand without degrading model accuracy, they may increase inference run time significantly or are limited in their applicability to different types of models.

1.1.2 Distributed Machine Learning Inference

On top of the challenge of limited working memory, the huge number of pre-trained model parameters of modern machine learning models require sufficient storage memory. One possible solution is to run the machine learning inference task in a distributed fashion, where the pre-trained model parameters are partitioned and distributed across multiple devices to reduce the amount of data on each individual device [22, 64, 106]. Furthermore, many deployments of devices already match such a system architecture when they are connected to each other via a local network, for example, a cluster of surveillance cameras. Distributed inference also lends itself to applications in which participating devices are mostly idle because new inputs arrive rarely. Thus, their unused processing power can support the devices that receive the input. The challenges addressed in this thesis are optimal partitioning of the data of the machine learning model and optimization of the communication demand imposed by device cooperation.

1.1.3 The Interface Between Hardware and Software

The peripheral devices, or short *peripherals*, of an MCU are its connection to the outside world [75, 60]. Sensors and actuators can be connected to such peripherals through pins. The control of peripherals is enabled by low-level hardware interfaces that are usually implemented as memory-mapped registers. Write accesses to special addresses may induce behavior in the peripheral, and read accesses to them can return data to the processor. The way in which the hardware registers are mapped to memory locations is commonly dictated by what is most convenient for the hardware design. It follows that the software accesses to these memory-mapped registers are not optimized with regard to software metrics, such as code size and the number of necessary accesses. However, since the amount of memory and compute time on MCUs is tightly constrained, it is desirable to optimize the interface between hardware and software. Another important resource in the life cycle of an MCU is

the cost of software development. To reduce it, parts of the development flow have to be simplified and automated whenever possible. Instead of being able to focus on just driver behavior, developers are challenged by having to additionally consider the register layout of the peripheral, performance of their accesses and the resulting memory footprint. On top of that, using low-level interfaces directly in driver and application code may pollute the source code with macros and bit manipulation operations, decreasing readability and maintainability. While this can be alleviated by using idioms like a Hardware Abstraction Layer (HAL) that hides low-level code, such layers of abstraction prevent behavior-specific optimization, causing inferior performance and memory footprint [30]. The challenges of optimization and effort of software development are addressed in this thesis by a method for optimized code generation of driver interface software.

1.2 Contribution of this Thesis

For all identified challenges, this thesis proposes to solve them using a similar approach. Each problem is first mathematically modeled so that the desired metrics can be optimized. In all cases, the developed methods are implemented within a design automation software tool to demonstrate its applicability and effectiveness in a wide range of test cases.

1.2.1 Fused Tiling for Optimized ML Inference

The inference task for a trained machine learning model can be described as a directed graph with nodes representing high-level operations and edges representing intermediate buffers. If the lifetime of two such buffers do not overlap, their memory storage may overlap, allowing to reduce the overall memory demand. A previously existing method to further reduce the memory demand is *fused tiling*, which will be described in this thesis [4, 106, 24, 13, 67, 68, 57, 25]. Fused tiling is a process in which the size of the intermediate buffers is reduced by calculating them in *tiles*, while also decoupling their lifetimes by *fusing* multiple consecutive operations. A new contribution of this thesis is *Fused Depthwise Tiling* that applies fused tiling in novel ways to enable new tiling opportunities without any run time overheads that would be induced by existing fused tiling methods. These new opportunities come from a wider applicability to more types of operations compared to existing methods that focus solely on convolutions. Combined with existing fused tiling, TinyML memory optimization could be improved significantly by expanding the available design space. To demonstrate the effectiveness of fused tiling and the improvement achieved by the new method, this thesis describes an end-to-end deployment flow that automatically determines where and how to apply fused tiling optimally on any given machine learning model. This flow also requires suitable memory-aware scheduling of operations and memory buffer layout planning. Hence, these two steps are also automated and efficiently implemented to conduct a fast exploration. Optimized tiling opportunities are found quickly through a method called path discovery, which analyzes any given machine learning model and explores possible fused tiling configurations. Expanding the fused tiling design space with Fused Depthwise Tiling improved the average memory reduction of sampled models from 32.8% to 46.3% with an unchanged run time overhead of 12.8%. When targeting performance-aware designs, the overhead could be eliminated while still achieving 28.8% average memory reduction.

1.2.2 Optimized Distributed ML Inference

In distributed machine learning inference, fused tiling also helps reduce the memory demand of each individual device, with the added benefit that tiles can be computed in parallel by all cooperating devices. Additionally, it is possible to apply fused tiling for the reduction of the storage memory required by each device, which is presented first in this work. This thesis describes a process that simultaneously optimizes for computation, memory and communication demands of a distributed deployment. The approach includes joint optimization of computation and the memory demands for both working memory and storage memory by distributing data evenly over all cooperating devices. This allows to run an application in the distributed scenario that would be too demanding for a single device. Furthermore, communication demand is minimized by finding an optimized configuration of the fused tiling. A fully distributed deployment of different machine learning models is demonstrated on a Raspberry Pi cluster to explore trade-offs between run time, memory requirements and communication overhead for different network bandwidths and device counts. For six devices on 100 Mbit/s connections, the integration of fused tiling additionally leads to a reduction of communication demands by up to 28.8%. This results in run time speed-up of the inference task by up to 1.52x compared to partitioning without fusing. Automatic optimization of the partitioning configuration could reduce the memory footprint per device by 25% over a handpicked configuration from previous work.

1.2.3 Automated HW/SW Interface Definition and Optimization

The smallest unit of a peripheral interface is called *bit field*. A bit field is a value that the processor can read or write to interact with the peripheral. Each bit field consists of one or more bits that are accessed atomically, that is, all at once. The first step toward a more optimized interface between hardware and software is the design and definition of a new specification format for the bit fields that enables new opportunities for optimization. An extension for the C programming language is described in this thesis, which allows one to define a flexible hardware/software interface, where the mapping between bit fields and memory addresses is not yet predetermined. This language extension allows developers to focus on desired software behavior using special features such as bit field array and hierarchy, while not having to consider performance implications imposed by the low-level interface. An optimized mapping of the bit fields to a register layout is determined by a heuristic method. Finally, a code analysis and generation approach that takes advantage of this optimized layout is shown. The approach is able to combine accesses to different bit fields to reduce the total number of accesses, and it inserts base pointers systematically to reduce memory usage through code reuse. In simple examples of driver code, the number of memory accesses is reduced by 36%, the estimated run time is reduced by 52% and the driver code size is reduced by 22%. This could be achieved at the cost of an 8.7x larger register map. The complexity of the source code is reduced by 39% when measured by Halstead effort.

1.3 Structure of this Thesis and Previous Publications

The remainder of this thesis is organized as follows. Additional background on MCUs and deep neural networks is given in Chapter 2. The state of the art prior to the publications

associated with this thesis is presented in Chapter 3. The description of the technical contributions is split into three chapters that are based on the following previous publications.

The use of fused tiling for memory optimization in deep learning inference is presented in Chapter 4 and is based on: Rafael Stahl, Daniel Müller-Gritschneder and Ulf Schlichtmann: "Fused Depthwise Tiling for Memory Optimization in TinyML Deep Neural Network Inference", in "TinyML Research Symposium 2023" [91]. This conference paper proposes a new tiling method that reduces memory usage without inducing any run time overhead compared to previously existing methods. It improves TinyML memory optimization significantly by reducing memory of models where this was not possible before while additionally providing alternative design points for models that show high run time overhead with existing tiling methods. Furthermore, an automated end-to-end flow with a new path discovery method is proposed that ensures all compared tiling methods are applied optimally for a fair comparison.

In Chapter 5, contributions to distributed deep learning inference on edge devices are discussed. These are based on: Rafael Stahl, Alexander Hoffman, Daniel Müller-Gritschneder, Andreas Gerstlauer and Ulf Schlichtmann: "DeeperThings: Fully Distributed CNN Inference on Resource-Constrained Edge Devices", in "International Journal of Parallel Programming", volume 49, 2021 [89]. This journal paper proposes an approach that supports a full distribution of CNN inference tasks by partitioning commonly used layer types along with a holistic optimization across layers. Memory, computation and communication demand is jointly optimized with techniques that combine both feature and weight partitioning with a communication-aware layer fusion method. The journal paper is an extension of the following conference paper: Rafael Stahl, Zhuoran Zhao, Daniel Müller-Gritschneder, Andreas Gerstlauer and Ulf Schlichtmann: "Fully Distributed Deep Learning Inference on Resource-Constrained Edge Devices", in "Embedded Computer Systems: Architectures, Modeling, and Simulation: 19th International Conference (SAMOS) 2019" [92].

Chapter 6 details the driver generation for optimizing memory-mapped register interfaces and is based on: Rafael Stahl, Daniel Müller-Gritschneder and Ulf Schlichtmann: "Driver Generation for IoT Nodes with Optimization of the Hardware/Software Interface", in "IEEE Embedded Systems Letters", volume 12, no. 2, 2019 [90]. This journal paper proposes a new method to reduce memory size, performance and development effort for device drivers. This is achieved by describing the driver behavior with a new C-like domain-specific language. The layout of the driver register interface is optimized so that register accesses can be combined. The required source code for the driver software is generated in an automated flow.

This thesis is concluded in Chapter 7.

Chapter 2

Background

BEFORE going into the details of the existing work and the contributions of this thesis, this chapter covers the foundation and introduces terminology used throughout the remainder of this thesis. Core concepts are the hardware platform of a microcontroller and deep neural networks used in machine learning.

2.1 Microcontrollers

A MicroController Unit (MCU) is a small computer on a single integrated chip [75, 60]. The common components of an MCU are shown in Figure 2.1. All communication between components is facilitated by one or more *busses*. A program and any static data are stored in *storage memory*, also called read-only memory (ROM), which is nowadays typically implemented as flash memory. The instructions of a program are read and executed by one or more *processor cores*. A core has a few internal registers for temporary working data, but an MCU also incorporates *working memory* to store larger amounts of data. Working memory is almost always implemented as random-access memory (RAM) - usually SRAM. Also on the bus, although often on a secondary lower-speed bus, are the peripheral devices. Peripherals that are present in almost every MCU are timers, interrupt controllers, input/output pin controllers (General Purpose Input/Output (GPIO)) and converters between analog and digital signals (Analog Digital Converter (ADC), Digital Analog Converter (DAC)). The interface between hardware and software is predominantly implemented through memory-mapped registers. This means that the processor core is executing regular memory load or store instructions and the system bus will redirect certain predefined address ranges to peripheral devices instead of the system memories. The smallest logical unit of a peripheral interface is a device parameter that can be read or written by the processor. In this thesis, the device parameters are called *bit fields*. The behavior of the individual bit fields within the memory-mapped registers is defined by the peripheral hardware specification. Typically, an MCU is provided to customers along with device driver code that abstracts this low-level interface to a more intuitive user-oriented one. An additional abstraction layer, called the hardware abstraction layer (HAL), may be introduced to separate driver behavior from low-level primitives [30]. Instead of accessing a raw memory address and performing shift and mask arithmetic, a bit field is accessed by its name through a HAL function. Chapter 6

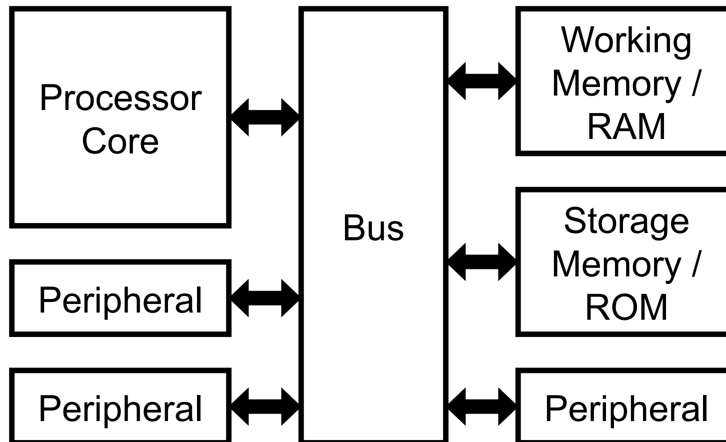


Figure 2.1: Common components of an MCU.

will present contributions toward an optimized hardware/software interface supported by automatic code generation.

The performance of a program running on an MCU is determined by the properties of all the mentioned components. First, the processor frequency limits the speed at which instructions can be executed. Its instruction set architecture dictates how many instructions are required to perform a desired computation. Bus and memory speeds and dynamic congestion activities can also have a great impact on overall performance. Many details of the microarchitecture, such as caches and branch prediction, influence program performance to a great extent as well. The focus of this thesis lies on the memories of an MCU. They limit the size of programs and data that can be stored on the device and are a major factor in the cost and energy consumption of the system because they take up a large portion of the chip area. For example, a popular microcontroller STM32F051R8T6 has been analyzed for its components [71] and the memories occupy more chip area (23%) than the core (22%) or peripherals (21%) with the remaining area dedicated to interconnects and I/O pads. Optimizing the memory usage of an application therefore either reduces cost and energy consumption or allows one to deploy a more advanced functionality that requires more memory.

The size and computation class of MCUs is at the very low end after servers, desktop computers, mobile and embedded devices. Compared to specialized hardware that has its function fixed, MCUs offer the ability to be programmed flexibly because they have a general purpose processor. Specialized hardware also requires a great deal of effort into chip design, while MCUs are available ”off-the-shelf”. These properties make MCUs a suitable platform for the Internet of Things (IoT) that connects huge amounts of tiny devices for applications such as smart cities, smart homes and industrial automation [8]. The devices in the outer layer of the IoT that interact with the physical world are called *edge devices* or *edge nodes*.

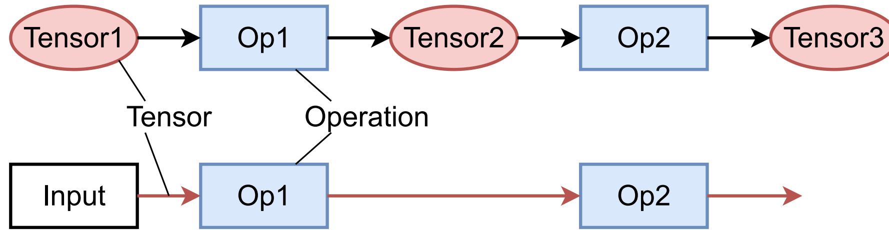


Figure 2.2: Example machine learning model architecture.

2.2 Deep Neural Networks

As MCUs became more capable, they could be established as a viable platform even for one of the most demanding application classes of machine learning [14, 95]. Although many machine learning applications run on large server farms or the *cloud*, such a deployment has various disadvantages compared to directly running an application on the device that senses new input and needs to act on it. The additionally required communication limits the bandwidth at which data can be exchanged between a device and its backend, also affecting the economics of a cloud solution. Since the two communication partners are often physically distant and have to communicate over the internet, the connection may have significant latency or be unreliable. Furthermore, the data to be processed must leave the device, raising concerns about privacy. These points motivate the deployment of machine learning applications on the IoT edge or completely off-grid. Terms for such a deployment have only recently been established as TinyML or Extreme Edge AI [104, 55].

A machine learning model is defined primarily by its *model architecture*. It describes which input arguments are accepted by the model, how these arguments are processed by the model computationally, and finally, what outputs are returned by the model. Computations are carried out on entities called *tensors*. Their primary attributes are their dimensionality and data type. While some modern model architectures include tensors of dynamic shape, they are typically not implemented dynamically in TinyML to avoid dynamic memory allocation and the increased complexity of dynamic operators. Instead, a static upper size bound is chosen and any unused data are dropped during inference. An example model architecture is shown in Figure 2.2. The top graph represents tensors as nodes of the graph, in contrast to the alternative notation on the bottom, which represents tensors as the graph edges or special properties of the operation nodes. Both notations are common and will be used in this thesis. The upper one is more clear about the fact that tensors may be reused by multiple successor nodes, while the lower notation is more concise and is more compatible with graph algorithms since there is only one type of node.

Besides the model architecture, a machine learning model also comprises trained static parameters, called weights and biases, or just *weights*. They are arguments of the operations within the model architecture and are determined by *training* the model. Figure 2.3 shows a generic end-to-end TinyML flow from the dataset and model architecture to the model deployed on a target device. Training takes a dataset and the model architecture and adjusts its weights so that the performance of the model on new unseen input is optimized. Optionally, the model can be refined with neural architecture search (NAS), a process that

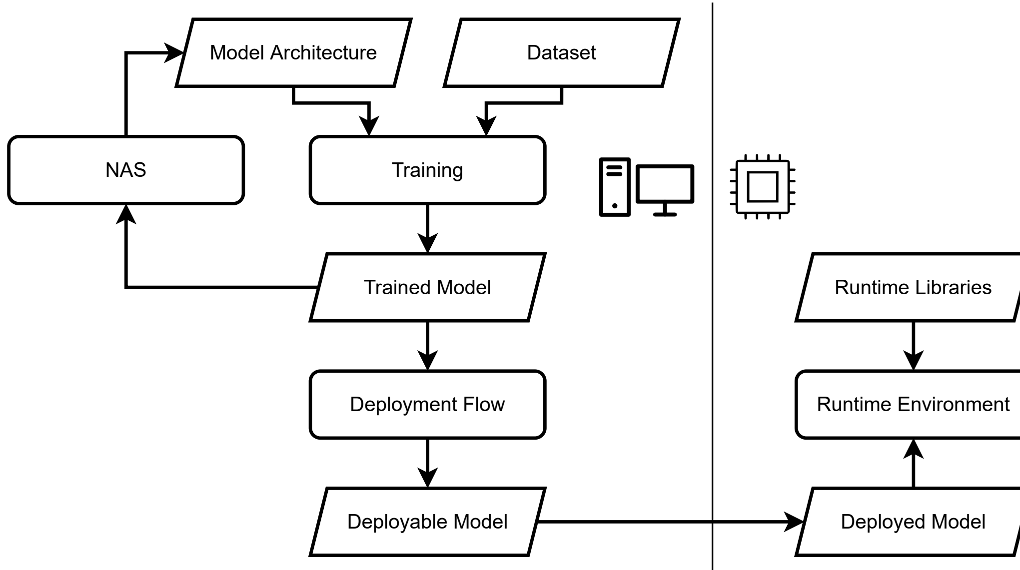


Figure 2.3: Generic TinyML flow.

automatically optimizes the model architecture. After the training stage, the model is deployed to its target device, where it can be fed with new input to predict the output values in a process called *inference*.

A model architecture with more than one operation is called a deep neural network (DNN). Since all modern practical machine learning models fulfill this criterion, the term DNN is used as an umbrella term for all neural networks in this thesis.

2.2.1 Dense Layer

Figure 2.4 shows an example DNN consisting of three *layers*. Layers are equivalent to the operations described above and operate on tensors, which are shown here as individual *neurons*. A layer that connects all input neurons with all output neurons is called *fully-connected* layer or also *dense* layer. Given the number of input neurons M_l and output neurons K_l for layer l, the computational operation of a fully-connected layer can be expressed as follows [95].

$$b_{l,k} = f\left(\left(\sum_{m=1}^{M_l} a_{l,m} \cdot w_{l,m,k}\right) + v_{l,k}\right), \quad k \in \{1, \dots, K_l\} \tag{2.1}$$

$a_{l,m}$ is the m-th element of the input neurons vector $\mathbf{a}_l \in \mathbb{R}^{M_l}$, $b_{l,k}$ is the k-th element of the output neurons vector $\mathbf{b}_l \in \mathbb{R}^{K_l}$, $w_{l,m,k}$ is the m,k-th element of the weight matrix $\mathbf{W}_l \in \mathbb{R}^{M_l \times K_l}$, $v_{l,k}$ is the k-th element of the bias vector $v_l \in \mathbb{R}^{K_l}$ and f is an *activation function*. An activation function is a nonlinear function that is essential for the operation of a DNN. If there were no activation functions, a DNN would only be able to model linear relationships between inputs and outputs. Common activation functions are sigmoid,

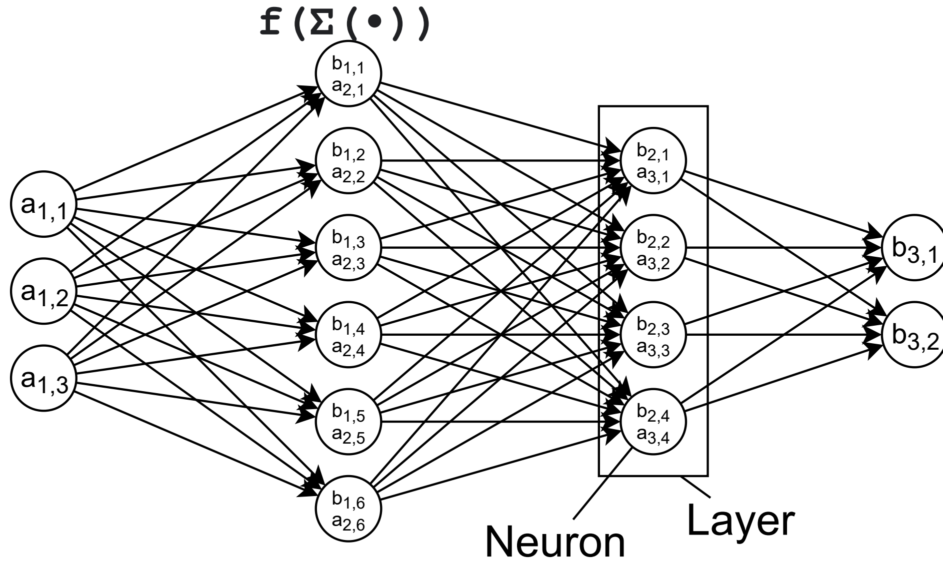


Figure 2.4: Example DNN with three layers.

hyperbolic tangent and rectified linear unit (ReLU), with the latter being most suited in the context of TinyML, because it can be computed quickly and is meaningful for quantized integer types.

Aside from dense layers, any common mathematical operation can be included in a neural network. The addition of bias values was already included in Equation 2.1, but can also be defined as an individual operation that represents element-wise addition. Different combinations of operations and how they are connected to each other define different types of DNN. Widely used ones are the standard feed-forward DNN presented here, auto-encoder networks, recursive neural networks and transformer networks. Almost all modern DNNs are convolutional neural networks, which will be described in more detail in the following section.

2.2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are of special interest for the contributions of this thesis due to their structure and because they are the most widely used model architecture in popular neural networks [84, 99, 57]. CNNs are motivated by classification tasks that take images as input. Images are represented as three-dimensional tensors for image pixel width, image pixel height and the number of color channels. The third dimension is often named *channels*, or especially for tensors other than the input tensor, *feature maps*. In later layers of a model and in use cases other than image recognition, the channels do not represent colors. Processing such large three-dimensional tensors with fully-connected layers would have extreme computational and memory requirements. This motivates a new type of layer that does not connect all input neurons with all output neurons. This new layer type called

convolutional layer takes advantage of the property of images that their data are spatially correlated. A DNN that contains any type of convolutional operation is also a CNN. A convolutional layer takes three-dimensional tensors as input, which corresponds to a set of feature maps. For example, an initial RGB input image consists of three feature maps, one for each color channel. The operation then applies a set of *filters* to this tensor, and each filter produces a single feature map of the output tensor. A filter consists of multiple two-dimensional *kernels*, one for each input feature map. The filters with their contained kernels are trainable parameters and are also called weights. As such, the computational operation of a convolutional layer can be expressed as follows [95].

$$\mathbf{B}_{l,o} = f\Big(\sum_{c=1}^{C_l} \mathrm{corr}(\mathbf{A}_{l,c}, \mathbf{W}_{l,c,o})\Big), \quad o \in \{1, \dots, O_l\} \tag{2.2}$$

The matrix $\mathbf{A}_{l,c}$ is the c-th feature map of the input tensor $\mathbf{A}_l \in \mathbb{R}^{X_l \times Y_l \times C_l}$, the matrix $\mathbf{B}_{l,o}$ is the o-th feature map of the output tensor $\mathbf{B}_l \in \mathbb{R}^{X_l \times Y_l \times O_l}$ and the matrix $\mathbf{W}_{l,c,o}$ is the kernel that connects the c-th feature map of the input with the o-th feature map of the output. The kernels are contained in the four-dimensional weight tensor $\mathbf{W}_l \in \mathbb{R}^{U_l \times V_l \times C_l \times O_l}$, where U_l and V_l are the width and height of the kernels, respectively. The activation function is again f. The function $\mathrm{corr}(\mathbf{A}, \mathbf{W})$ computes the two-dimensional cross-correlation, for which the x, y-th element is computed with the following formula.

$$\mathrm{corr}(\mathbf{A}, \mathbf{W})_{x,y} = \sum_{(u=-\lfloor \frac{U}{2} \rfloor)}^{\lfloor \frac{U}{2} \rfloor} \sum_{(v=-\lfloor \frac{V}{2} \rfloor)}^{\lfloor \frac{V}{2} \rfloor} a_{x+u,y+v} \cdot w_{u,v} \tag{2.3}$$

Note that there may be additional properties of a convolution that are not considered in these definitions. For example, a stride refers to a step size at which the feature maps are read, and a padding can be defined in different ways that expands the input feature map borders to produce larger output feature maps.

Figure 2.5 shows a CNN consisting of two layers with kernel dimensions $U_1 = V_1 = U_2 = V_2 = 3$. The input tensor has dimensions $\mathbf{A}_1 \in \mathbb{R}^{8 \times 8 \times 3}$ and is processed by the first convolution with $O_1 = 8$ output channels, resulting in the intermediate tensor $\mathbf{B}_1, \mathbf{A}_2 \in \mathbb{R}^{8 \times 8 \times 8}$. The second convolution with $O_2 = 2$ output channels produces the final output $\mathbf{B}_2 \in \mathbb{R}^{8 \times 8 \times 2}$. Note that, as highlighted with a few examples, each output neuron depends on all input neurons at the same x and y coordinates across all input channels, with additional neighboring input neurons being included for kernel dimensions larger than 1x1. A padding of one pixel was added around the input edges in the width and height dimensions of the input to obtain the same feature map dimensions in the output.

Aside from this base case of a convolutional layer, there are many variations. A depthwise convolution refers to a variant that comprises a single filter, of which each kernel produces an individual output feature map instead of summing their results. In pointwise convolution, the kernel sizes are constrained to 1×1, acting as a weighted sum over the channels of each input pixel. The combination of depthwise and pointwise convolution is called depthwise separable convolution and is a common technique to reduce the number of parameters while keeping the neuron dependencies of the standard convolution. Due to their high dimensionality, convolutional operations usually are the most computationally and memory-intensive in many CNNs. Given the focus of resource-constrained MCUs in this thesis, it is important to consider convolutions in any optimization method.

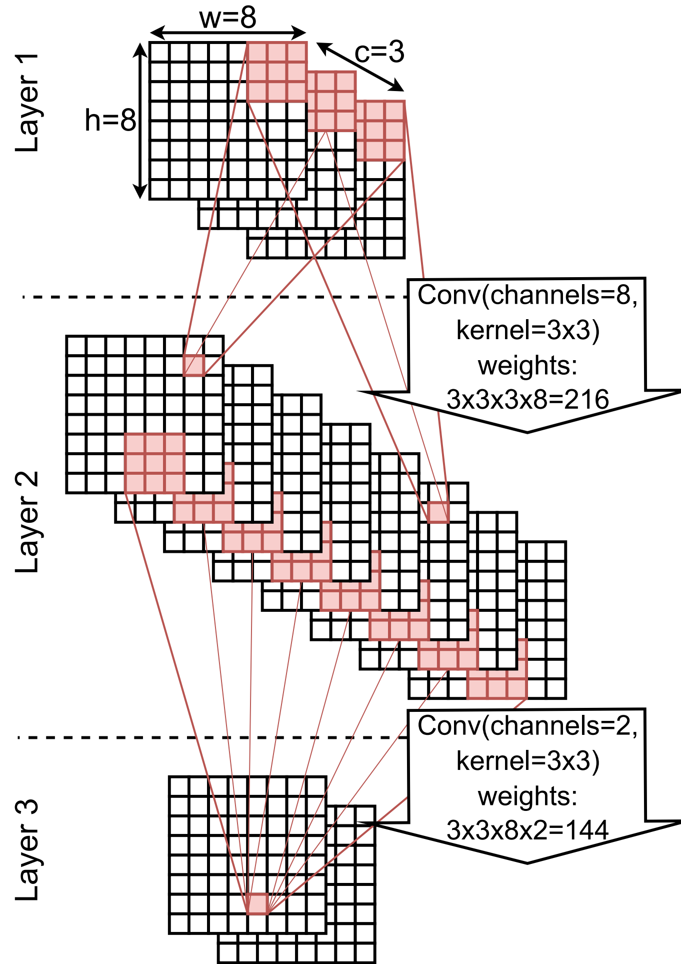


Figure 2.5: Example CNN with two layers.

2.3 Deep Learning Frameworks

As DNNs drove advances in many different fields by employing more sophisticated models and larger datasets, the need for software frameworks that support the end-to-end deployment from the source model to the target hardware has increased. Deep learning frameworks let the user define a model architecture and automatically generate optimized executable code for training or inference. The machine learning implementations for the contributions of this thesis use deep learning frameworks to achieve competitive results that are comparable to existing work. Many popular deep learning frameworks like TensorFlow [1], PyTorch [73] and MXNet [19] use an intermediate representation (IR) to implement graph-level optimizations on high-level deep learning operators. These operators are then implemented with target-specific libraries to accelerate them. TensorFlow Lite is a framework specif-

ically targeted at ”mobile and edge” devices, providing only a subset of operators and a reduced runtime environment [35]. TensorFlow Lite for Microcontrollers (TFLM) is an even smaller subset that is suitable for MCUs, reducing runtime memory overhead to values below 50 kB [28]. DarkNet is a deep learning framework that stood out because it achieved state-of-the-art results with the YOLO object detection models [78, 80]. For this reason, it was used in the existing work on which the contributions in Chapter 5 are based.

A more advanced approach to model optimization is introduced with machine learning compilers such as Apache TVM [20], Glow [81] and XLA [82]. Instead of treating each operator individually, these tools have a global view of operators that enables cross-operator optimizations like operator fusion and global memory planning. New platforms can be targeted with less engineering effort, because not every operator needs a handwritten implementation. An IR of a machine learning compiler can lower directly to operations that are available for the specific target platform, e.g. matrix/vector multiplications for Accelerators/GPUs or scalar multiplications for simple MCUs.

Apache TVM is used for the implementation of the contributions in Chapter 4 because the presented optimization method requires complex cross-operator transformations and TVM provides the appropriate tooling to implement them. TVM is a state-of-the-art machine learning compiler capable of transforming various input formats of DNN models into various deployable output formats. All the deep learning frameworks mentioned above can provide a DNN model format that is recognized by TVM as input. The compilation flow is divided into two major steps with their respective IRs that are aware of the machine learning domain. First, TVM transforms the input model graphs into the *Relay* IR where graph-level transformations can be applied. Relay has a human-readable representation that is structured as follows.

```
1  def @main(%input_1: Shape(1, 32, 32, 3)) -> Shape(1, 5) {
2    %0 = nn.conv2d(%input_1, Const[0]: Shape(3, 3, 3, 10), padding=[1, 1, 1, 1],
3      channels=10, kernel_size=[3, 3]) -> Shape(1, 32, 32, 10);
4    %1 = add(%0, Const[1]: Shape(10)) -> Shape(1, 32, 32, 10);
5    %2 = nn.relu(%1) -> Shape(1, 32, 32, 10);
6    %3 = nn.max_pool2d(%2, pool_size=[2, 2], strides=[2, 2], padding=[0, 0, 0, 0]
7      ) -> Shape(1, 16, 16, 10);
8    %4 = reshape(%3, newshape=[-1, 2560] -> Shape(1, 2560);
9    %5 = nn.dense(%4, Const[2]: Shape(5, 2560), units=5) -> Shape(1, 5);
10   nn.relu(%5) -> Shape(1, 5);
11 }
```

Functions are defined with the `def` statement and take a number of runtime parameters and return a return value. Every tensor value is annotated with a `Shape` that declares the dimensions of that tensor. Arguments named `Const` represent the weights and biases. Expressions can either be nested (`func1(func2(x))`) or aliased with a variable name, here `%0 - %5`. The example code describes a toy function `main` that takes the 32x32 3-channel image named `input_1` as input and returns a vector of five float values. It first applies a 3x3 convolution (`nn.conv2d`) with bias addition (`add`) and ReLU activation (`nn.relu`), followed by a 2x2 max pooling operation (`nn.max_pool2d`) and finally a fully-connected layer (`nn.dense`) with ReLU activation. After all optimizations have been applied on the Relay level, the model is transformed to the second IR, called *TIR*. TIR is a low-level representation that only acts on the scope of fused operations and is the direct input to various backends that are able to produce output for different deployment scenarios. For example, C code for deployment on microcontrollers or CUDA code for deployment on Nvidia GPUs. TVM was chosen as the basis for the implementation because its Relay IR

is very suitable for adding complex transformation passes. To achieve competitive results compared to widely-used frameworks like *TensorFlow Lite for Microcontrollers*, the *Ahead-of-Time (AoT) TVM backend* was chosen for the work presented in Chapter 4 because it generates static code that is able to run the DNN inference without the full TVM run-time libraries. In TVM, many DNN operations are fused to completely eliminate intermediate buffers. For example, a convolution with bias addition and activation function is carried out by adding the bias and applying the activation function while calculating each individual convolution output value. All intermediate buffers between such fused operations do not contribute to the peak memory usage of the deployed model.

2.4 Summary

Microcontrollers are integrated circuits with a focus on low price and power consumption, dictated by the chip area, of which a large part is dedicated to memory sizes. These devices are often used to perform smart functionalities, for example in IoT applications. One increasingly popular approach for processing large amounts of data collected by edge devices is machine learning, typically with deep neural networks containing some form of convolutional layer. Deep learning frameworks and compilers are software tools that support the development and deployment of DNN models.

Chapter 3

State of the Art

This chapter presents a comprehensive review of the current state of the art to establish a clear baseline for the contributions of this thesis. Existing optimization approaches for memory-constrained devices in the area of deep learning inference and device drivers are reviewed.

3.1 Deep Learning Inference on Edge Devices

As already outlined in the Introduction and Background, machine learning is valuable for many applications on constrained devices. DNNs can be fed with large, complex and noisy sensory input data and transform them into an output that is easy to interpret, e.g., a classification result. The focus of this thesis lies on the inference side, assuming that a trained model already exists. DNN inference is demanding in terms of computation, energy and memory resources.

A widely used solution to overcome the limited resources in low-power devices is to offload computation to other infrastructure, such as cloud or fog devices [47, 87]. Although this typically allows for much more powerful models, the input and output data need to be transferred over a network. Transferring raw data requires high bandwidth between the edge device and the cloud backend. The physical distance between the two introduces network latency, which can be a dealbreaker for real-time applications. The network and backend infrastructure add significant cost and reduce the overall reliability of the entire inference solution. Furthermore, the inference input and result may be sensitive data that must not leave the edge device. Due to inferior latency, bandwidth, privacy and cost, there is a strong push to move DNN inference to edge devices with the TinyML paradigm. A load-aware approach presented in [100] focuses on partitioning and distributing parts of a model between different levels of processing power. During inference, the model can stop at an intermediate layer if it has high confidence in a result. Although such a model would execute edge-only for certain inputs, it still carries all the downsides of the full cloud offloading for the others.

Of course, it is challenging to run DNN inference on constrained edge devices, but even demanding applications such as keyword spotting, visual wake-up, anomaly detection and radar gesture recognition were shown to be deployable to tiny MCUs with only a

few hundred kilobytes of working memory [12, 102]. One core challenge is the limited RAM available for intermediate storage of run-time buffers. During training of a model, all data are typically represented as 32-bit floating-point values. This level of precision is not necessary for accurate inference, and therefore, various quantization methods have been developed [33, 52, 27]. Today's machine learning frameworks support quantization of all weight and activation data to at least 8-bit integers with low effort. This reduces the model size up to 4x in both ROM and RAM and additionally makes it faster and more efficient to execute. *Post-training quantization* is applied on an already trained model and does not require retraining, while *quantization-aware training* refers to a process in which the model is quantized during training. Another related model compression technique is pruning [15, 103, 9]. Here, weights that have a negligible impact on the model result are removed completely. On its own, this would result in sparse data structures that introduce a significant performance penalty on general purpose hardware. A more advanced pruning technique is structured pruning, in which whole rows of a dimension are systematically dropped [41, 37]. Quantization and pruning are both effective methods to reduce the size of the deployed model at the cost of some model accuracy. There exist various device-local methods to optimize performance and memory usage on a single device, such as shrinking and compressing the DNN [54, 42, 15, 69]. Applying these methods can reduce the output accuracy, thus no longer making the model a viable solution. As such, there will always be models that are too complex for a single device.

Another approach to bring deep learning inference to constrained devices is the design of model architectures specifically for constrained devices [43]. More generally, NAS is another method that is able to find compromises between the core metrics of memory usage, run time (or power consumption) and accuracy. Given a dataset and a method for evaluating fitness, many different network architectures are systematically searched to find the most suitable one [98, 58, 12]. Often, this will be a multi-objective optimization on the Pareto front across the core metrics. NAS has enormous search spaces, but in the TinyML domain, these become more manageable.

All memory optimization methods mentioned so far have in common that they change DNN parameters and, therefore, the DNN's behavior and inference results. One method to reduce memory usage without changing any DNN behavior is fused tiling, which is the basis for the contributions of Chapter 4 and is discussed in the following section.

3.2 Fused Tiling

It is observed that many DNNs have an architecture where only a single or very few intermediate buffers dominate the memory requirements of the entire model. Figure 3.1 shows four popular models where this observation can clearly be seen [28, 50, 72, 84].

Fused tiling is a method that reduces the required memory of such large intermediate buffers by changing the order of computation across operators. *Tiling* by itself refers to the splitting of DNN operations into multiple *tiles*, or also called *partitions*, which can then be computed independently of each other. It is used primarily within a single operation to accelerate execution through parallel computation, for example, in processor cores or processing elements of a hardware accelerator [58, 4]. Another application of tiling is the partitioning of DNNs so that they can be run distributed over several devices [106]. This will be discussed in more detail in Section 3.3. *Fusing* by itself refers to the process of

(a) Magic Wand

(b) Text Classification

(c) PoseNet

(d) SSDLite

Figure 3.1: RAM usage of DNN intermediate buffers.

combining multiple operations into a single one that intersperses their low-level operations. For non-tiled operations, this usually refers to the fusion of all element-wise operations that are part of the tensor operations. For example, after a dense layer, it is trivial to fuse the bias addition and activation function into the computation of each output element. *Fused tiling* is the combination of both ideas, where the tiled operations are fused together. Since fusing element-wise operations is trivial, this thesis is particularly concerned with the fusion of multiple commonly used high-level operations, such as dense and convolutional layers. These notably have wider input dependence than element-wise operations, which makes their fusion non-trivial. Fused tiling, as presented in existing literature, will be referred to as Fused Feature Map Tiling (FFMT).

FFMT refers to the fusion of convolutional layers when they are partitioned along their feature maps. Figure 3.2 illustrates FFMT on two consecutive convolutional operations as part of a DNN. The three sets of feature maps are the input of the first operation, an intermediate buffer and the output of the second operation. Since the intermediate buffer is larger than the input and output, calculating it in a tiled manner could reduce the mem-

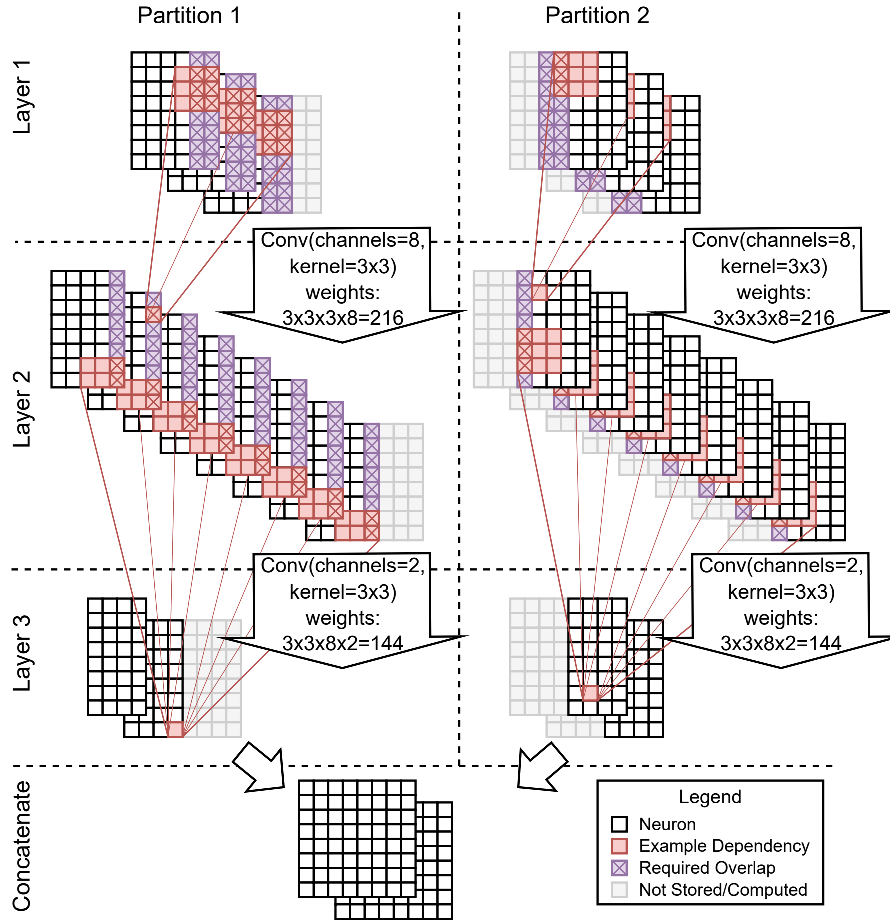


Figure 3.2: FFMT applied to two consecutive CNN layers.

ory requirements of their computation. FFMT does this by splitting all feature maps of the intermediate tensor buffer into partitions. Convolution operations have spatial locality, which allows one to produce output feature maps from split inputs mostly independently. However, convolution kernels larger than 1x1 cause an *overlap* in the input partitions that accumulates additively over all fused operations. Overlap refers to data that must be calculated by multiple partitions redundantly. More generally, overlap is introduced by an operation if its kernel size is larger than its stride size in the dimension that is partitioned. For example, a convolution with kernel size 6x2 and stride size 2x2 introduces overlap when partitioned along the first dimension, but not when partitioned along the second dimension. The example in Figure 3.2 shows how two operations are split into two partitions and their overlap caused by 3x3 convolutions is highlighted. Since the goal is a reduced working memory usage, the order of operations is important. After calculating the first partition of the large intermediate buffer, the fused second convolution must be executed directly afterward to free the memory usage of the partitioned intermediate buffer. Only then can the second

partition of the buffer be calculated. In this way, the large intermediate buffer must never be kept in memory in its entirety. In FFMT there is no inherent limit to the number of consecutive convolutions until the overlap becomes too large to achieve memory savings or the run time overhead becomes impractical. The convolutional operations tiled with FFMT may be interleaved with other operations, such as simple element-wise operations such as bias addition and activation functions. Convolutions are often interleaved with pooling operations, but even these may be part of the partitions as long as their input size does not introduce constraints that are too restrictive. Notably, the partition boundaries must not split the input of a pooling operation or overlap must be introduced.

FFMT is not a new concept introduced in this thesis and is described in more detail in the related literature [4, 106, 24, 13, 67, 68, 57, 25]. FFMT was first employed for reducing peak memory usage in [24], but their path discovery requires partially manual effort from the user. Other works that use FFMT with automated path discovery are [13, 67, 68, 57, 25].

3.2.1 Memory-aware Scheduling

For many DNNs, scheduling is trivial because their graphs do not contain any branches. The operation nodes are scheduled in the order in which they are located on the single path of the graph. In other words, there is only a single possible topological sort of the graph. However, with tiling, parallel paths are introduced in the DNN graph and different schedules become possible that determine the lifetime of the intermediate buffers and, hence, peak memory. It then becomes a challenge to find the schedule that minimizes peak memory. Scheduling for optimal run time has been widely studied and has also been applied in the context of machine learning [93]. A simple approach to memory optimization is to iterate all possible topological sorts of the DNN graph [56]. The run time of this enumeration can quickly become unmanageable for more complex DNN graphs. While optimal memory-aware scheduling has been achieved before in [3] using a dynamic programming approach with adaptive canceling, tiled graphs with large number of partitions and many split operations can still quickly cause unmanageable run times. Tiled DNNs resemble *series-parallel graphs (SP-graphs)*, that is, graphs that only consist of series and parallel compositions of other SP-graphs and the base case of a single node. Optimal memory-aware scheduling of SP-graphs has been solved with a polynomial-time algorithm by [48] based on [61].

3.2.2 Memory Layout Planning

After a schedule has been determined, all intermediate buffers of the DNN graph have to be mapped to concrete memory locations. Optimizing this mapping for minimal memory usage is a nontrivial task because buffers can overlap in memory, as long as they are not live at the same time. Thus, many buffers can be placed at overlapping memory locations to save total memory space. Determining optimal placement is an NP-complete resource allocation problem. TensorFlow Lite for Microcontrollers (TFLM) employs a greedy heuristic to approximate the optimal solution [77][86][6]. When the buffers are movable between operations, the problem becomes trivial, because they can then be packed as compactly as possible after each operation [56]. But this requires copy operations, which negatively impact performance. The Apache TVM machine learning framework implements a heuristic approach based on hill-climbing and simulated annealing that outperforms the TFLM

heuristic in many cases [20].

This thesis contains contributions to methods for scheduling and layout planning for the fused tiling approach presented in Chapter 4.

3.3 Distributed Inference

For many applications, the overall performance of the system can be greatly improved by distributing more computation to other devices [22]. An orthogonal solution is therefore the utilization of multiple cooperative devices to carry out the DNN inference task in a distributed and cooperative fashion. In many existing applications of edge devices, a large number of them are available and already connected with each other via a local network, for example, a cluster of surveillance cameras. This means that many existing installations already have the required system architecture for performing distributed inference. Another advantage of distributed inference is that, when inputs arrive, most devices are idle, given the low duty cycles of common sensor devices. Thus, a low-cost but efficient solution can be established by utilizing the idle time of other edge devices. Fully distributed inference has previously been achieved by the approach of MoDNN [64]. MoDNN distributes a DNN across multiple mobile phones connected via a wireless network. The approach distributes both the input and output data of the layers, as well as the weight data across devices. While this approach is able to partition weights, it focuses mainly on sparse fully-connected layers, i.e. fully-connected structures, where some weights are zero. The approach does not take the communication between fully-connected layers into account, and weight-intensive convolutional layers are not addressed. Furthermore, the approach proposes to process networks in a layer-by-layer fashion, requiring all devices to synchronize by exchanging data after each layer. Another way to achieve fully distributed inference is layer pipelining [65]. However, this method is unable to evenly distribute the memory demand for typical models.

Model distribution has previously been researched in the context of hardware accelerators. The work in [4] presented such a method, with the central idea of fusing the first few layers of the network to reduce the total transfer of data to and from the chip. In contrast to this thesis, the fusion method in [4] targets memory-constrained accelerators instead of similarly constrained individual edge devices. Fusing optimization for the accelerator is only investigated for the feature-intensive layers, while the fusing approach presented in this thesis additionally targets the weight-intensive layers. Other works on accelerators have focused on aggressive parallelization and do not apply to single/few core devices [7]. Another related topic is the distribution of tasks within a network of collaborative edge devices. Several methods are proposed on how this should be done [83, 18], but these works handle general tasks and focus on the network parameters. In contrast, the work presented in this thesis deals with internals of fully-connected and convolutional operations to remove dependencies between tasks, which would have had to be respected by more general approaches. The use of larger-scale edge devices to share work was explored in [49], but this has the disadvantage that a more powerful device is added to the network along with its additional power requirements.

Table 3.1 contrasts the existing work with the work presented in this thesis. Pipelining is a simple method for distributed inference, but it cannot evenly partition the data. FFMT is only capable of partitioning convolutional layers, cannot partition weights, and introduces overhead from overlapping partitions. MoDNN also uses a one-dimensional variant of FFMT

Table 3.1: Comparison of Inference Partitioning Methods

Work	Layer Types	Able to Split	Restrictions
Pipelining [65]	FC & Conv	Features & Weights	Uneven partitions
FFMT [4, 106, 24, 13, 67, 68, 57, 25]	Conv	Features	Overhead of overlap
MoDNN [64]	Sparse FC	Features & Weights	No layer fusion
Chapter 4 [91]	FC & Conv	Features	-
Chapter 5 [92, 89]	FC & Conv	Features & Weights	-

to partition feature maps of convolutional layers, but it is additionally capable of splitting the weights of sparse fully connected layers. It does not involve any layer fusion beyond trivial element-wise operations. In Chapter 4, the primary goal of fused tiling is to reduce the required working memory (RAM) for the storage of large intermediate buffers. A novel fused tiling method is introduced to achieve this without the overhead of overlaps. Chapter 5 will demonstrate another application of the novel fused tiling method for distributed inference that reduces static memory usage (ROM).

3.4 Driver Software on Edge Devices

The development of driver software for edge devices requires significant engineering effort as part of the product development cost. This has previously been addressed with improved ways of specifying driver behavior. Devil [66] and HAIL [94] are domain specific languages (DSLs) that provide mechanisms to describe the relationships between bit fields in a more granular way. Their description format consists of so-called triggers that define side effects that define how accesses to one field might affect another. HAIL allows one to define logical and sequential dependencies between bit fields and includes a mechanism to access multiple parameters together in a single block. In Devil, the defined register layout has a fixed register layout, which prevents optimization of that layout. For HAIL, the register layout is not specified in the language itself, but must be provided as configuration input. The central issue with these languages is that they are completely new languages that are unfamiliar to driver developers. Laddie [105] is an extension to Devil with the same issue. NDL [26] is a DSL that builds on top of Devil by extending it with a driver state-level function, which is beyond the scope of the work presented here.

On tightly constrained edge devices in the MCU class, driver software can occupy a significant share of the available storage memory. Optimizing the way bit fields are mapped to registers is a possible way to reduce this memory demand. Register layout optimization was investigated in [59]. The work defines costs for the different configurations of bit fields in registers, while also considering combined accesses. The objective function includes the total code size and a performance metric defined as instruction costs weighted by the number of occurrences during profiling. The user needs to choose whether the optimization should focus on code size or performance. The authors define a hardware cost as the total number of registers that are occupied after register allocation. They present an integer linear programming (ILP) formulation which can optimize for either software or hardware cost, but admit that such an ILP cannot be solved in a reasonable amount of time. As a practical solution, they provide two heuristic approaches.

Chapter 6 presents an approach that combines optimization of the register layout with

a novel way to specify driver behavior.

3.5 Summary

This chapter provided an overview of the existing work that the contributions of this thesis build upon and extend. Deep learning inference on edge devices has been achieved with cloud offloading, model compression techniques and complex neural architecture search. Fused tiling is a method to reduce peaks in memory usage to reduce the overall memory requirements of DNN inference. Distributed inference is yet another way to reduce memory requirements by sharing the inference work across multiple cooperating devices. Finally, optimized driver software is vital to reduce memory overhead from other sources than the DNN itself.

Chapter 4

Fused Tiling for Memory Optimization in DNN Inference

TinyML is a field that has recently been enabled by advances in both hardware and software [104]. General hardware improvements and the introduction of specialized hardware accelerators for machine learning operations bridged the gap on one side [46] and on the other, compression techniques, as introduced in the background chapter, closed that gap [95, 99, 27]. Although even training on MCUs is possible [36], this chapter focuses on inference and how it can be optimized effectively, especially in terms of memory demand.

4.1 Introduction

Section 3.2 introduced fused tiling and the existing Fused Feature Map Tiling (FFMT) method. Quantization and pruning techniques are orthogonal to fused tiling because tiling can always be applied additionally to such compression techniques. Neural architecture search is also an orthogonal optimization method. It can be applied along with the methods described in this chapter and might even interact symbiotically, because architectures that would have been rejected for their large memory usage could also be considered as candidates. Fused tiling is especially effective for models that have a single or very few intermediate buffers that dominate the memory usage. This chapter explores how fused tiling is applied effectively to reduce memory usage.

The main contribution of this chapter is the introduction of a so-far unexplored fused tiling method for the memory optimization of DNNs. This method, called Fused Depthwise Tiling (FDT), enables new tiling opportunities that reduce peak memory usage without any run time overhead that would be introduced by existing methods. Additionally, FDT can be applied to a wider variety of layer types than existing methods that focus solely on convolutions. A model can be tiled with existing methods and FDT in conjunction so that the design space for TinyML memory optimization is overall expanded. To explore this expanded tiling design space, an end-to-end deployment flow is described that automatically determines where and how to apply fused tiling optimally on any given DNN. Exploiting tiled graphs for memory reduction additionally requires a suitable memory-aware scheduling

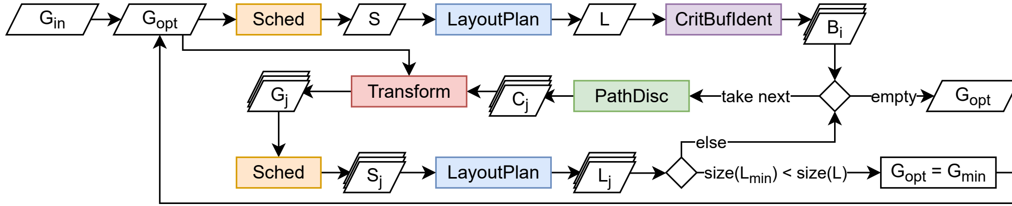


Figure 4.1: Automated tiling exploration flow.

of operations and an optimized mapping from memory buffers to a memory layout. Hence, these two steps are also automated and efficiently implemented to conduct a fast exploration. To quickly find optimized tiling opportunities, a process called path discovery is run that analyzes the DNN graph and explores possible tiling configurations. This thesis is the first work to present FDT and this chapter details its application to memory optimization of DNN inference.

In summary, the contributions are as follows.

1. The tiling method FDT applied for the memory optimization of DNNs to expand the design space by reducing memory further or eliminating run time overheads.

2. An automated exploration with a new block-based path discovery to find suitable tiling configurations, a memory-aware scheduling and optimal memory layout planning.

This chapter will describe an optimization flow using both mentioned fused tiling methods to reduce memory usage in DNNs. Figure 4.1 gives an overview of the steps involved in this flow. Firstly, the operations of the given DNN graph G_{in} are scheduled in a memory-optimized order S. After the schedule has been fixed, all required intermediate buffers are placed into a linear memory space such that the total required peak memory is minimal. The resulting memory layout L is analyzed to extract a list of intermediate buffer candidates B_i that may reduce total memory usage if they were to be tiled. These buffer candidates are passed to the *path discovery* in descending order sorted by their size. The path discovery step identifies tiling configuration candidates C_j for the first buffer candidate. If no configuration could be found that reduces the memory usage, the next buffer candidate is tested. All configuration candidates are passed to the actual graph transformation pass that applies tiling on the DNN graph to produce graph candidates G_j. These are again evaluated by scheduling and memory layout planning. If the memory size of the smallest found layout L_{min} is smaller than the current layout L, the corresponding tiling configuration improved memory usage and the currently best graph candidate G_{opt} is updated. The optimization flow works iteratively. The newly generated tiled DNN graph G_{opt} is evaluated again as new input beginning with scheduling. The flow terminates when no buffer candidate B_i produces a tiling configuration that reduces the layout size further. Each step of this flow is described in detail in this chapter.

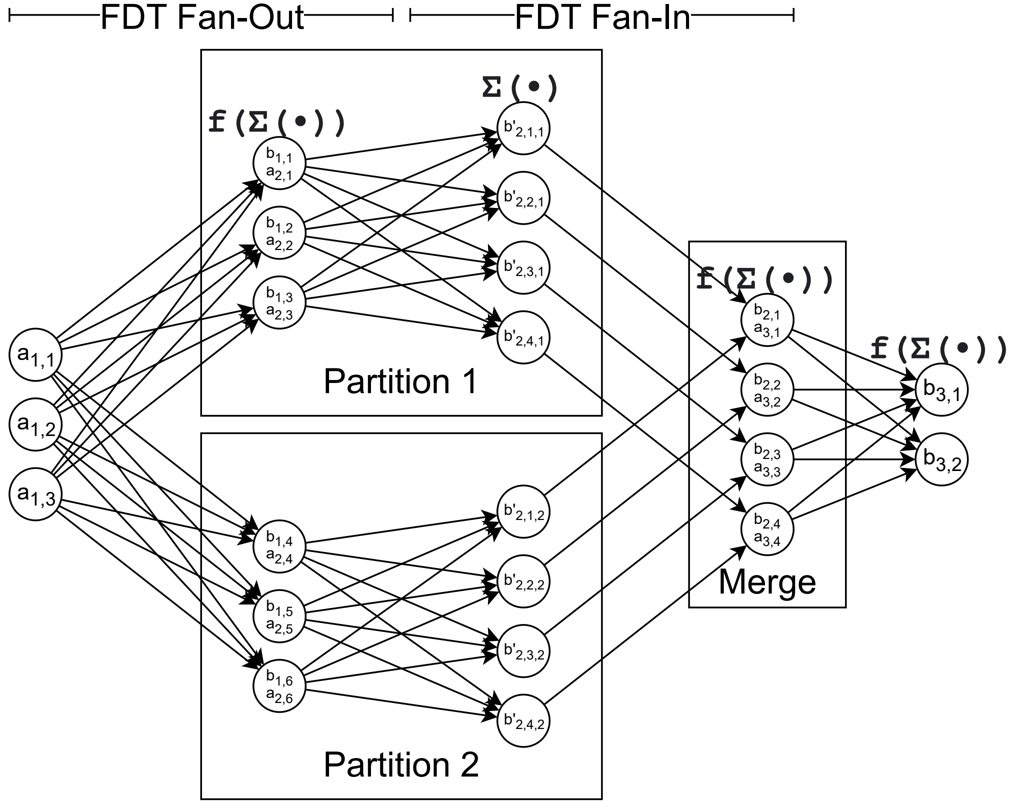


Figure 4.2: FDT applied to two consecutive dense layers.

4.2 Fused Depthwise Tiling (FDT)

Fused Depthwise Tiling (FDT) is a novel fused tiling method proposed by this thesis. In Chapter 5 it will be applied as a means of partitioning DNN weights of fully-connected layers and convolutional layers that have a large number of weights. This chapter will discuss the application of FDT for the optimization of working memory, i.e. RAM, whereas the work presented later targets the static parameters, i.e. ROM.

The primary goal of fused tiling for memory optimization is the splitting of large intermediate tensor buffers so that their partitions can be computed independently with reduced memory demand. As shown in Figure 4.3, FDT does this in the depthwise dimension instead of along the feature maps as with FFMT (compare Figure 3.2). Switching to the depthwise dimension avoids any overlap in the intermediate buffer. However, it requires that the input and output buffers are fully available to every partition, because every single output feature map is the result of summing all input feature maps after applying a convolutional filter. Figure 4.2 helps explain this concept with two consecutive dense layers tiled into two partitions. Half of the original six output neurons of the first layer (*FDT Fan-Out*) are computed in each partition using all input neurons. For the second layer (*FDT Fan-In*), the

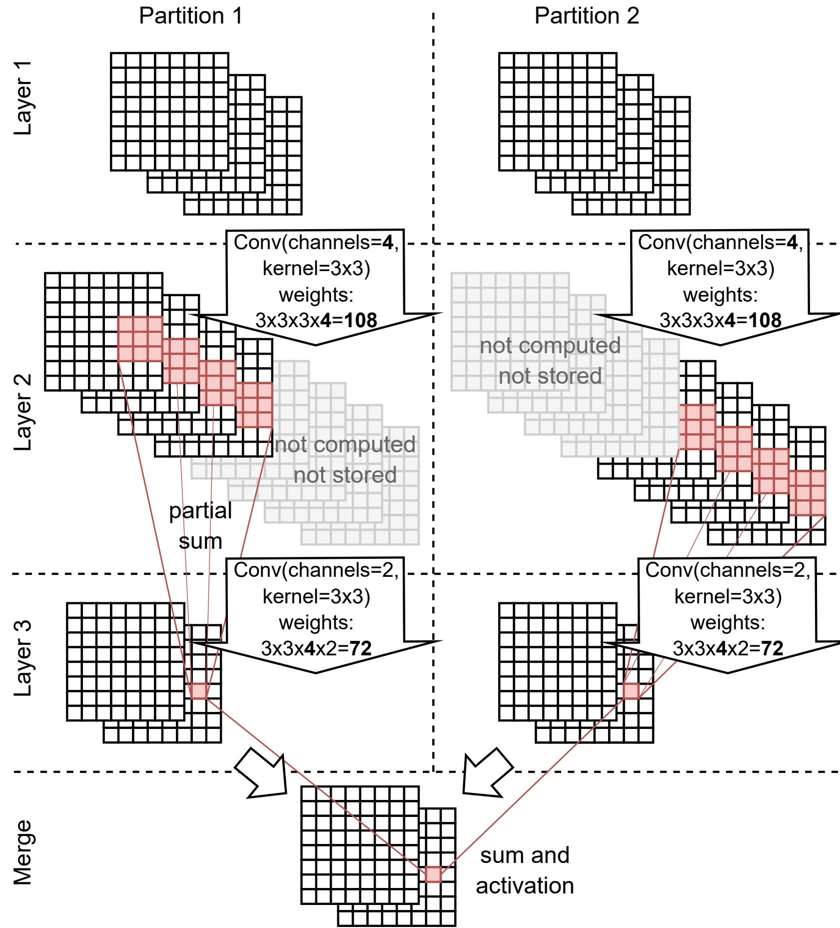


Figure 4.3: FDT applied to two consecutive CNN layers.

original four output neurons can only be computed partially, because not all input neurons are available to every partition. However, since a dense operation is a sum of products, all partial values of all partitions can be recombined by summing them element-wise and applying the activation function afterwards in a new appended *Merge* operation. Since activation functions are nonlinear, this imposes a limit of two FDT-partitioned operations for each tiled sequence.

Whereas FFMT requires spatial locality of all operations, FDT can be applied to a wider range of operations where all output elements depend on all input elements as long as there is no interdependence between the output elements. Examples of operations that can only be tiled by FDT are dense operations and pairs of embedding lookup (e.g. TensorFlow `gather` function) and axis reduction (e.g. by taking the mean).

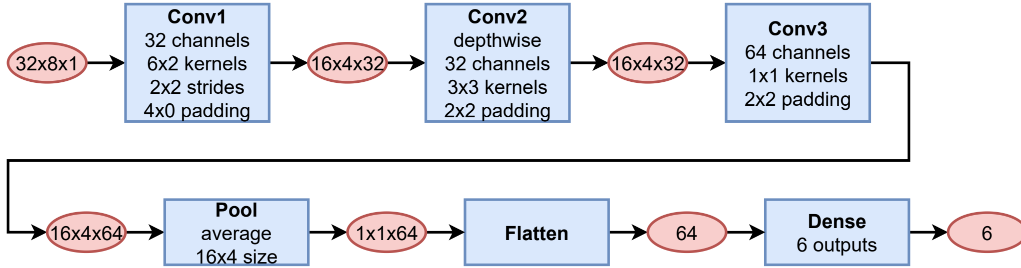


Figure 4.4: Architecture of the running example.

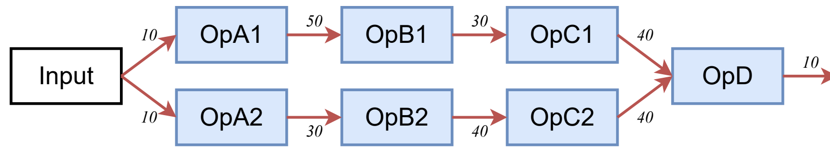


Figure 4.5: Example DNN graph for scheduling.

4.3 Automated Tiling Exploration

It is not meaningful to demonstrate the theoretical memory usage of fused tiling methods in isolation, because the practical memory usage is heavily affected by the entire end-to-end deployment flow with the interdependent problems of tiling configuration, operation scheduling and layout planning. Each of these problems will be addressed in this section. The flow in which these steps are embedded is already shown in Figure 4.1.

RUNNING EXAMPLE:

The entire automated tiling exploration flow will be demonstrated on a running example of a simple DNN. Its architecture is a simplified version of the *Key Word Spotting* DNN of MLPerf Tiny [11] and is shown in Figure 4.4. The box-shaped nodes of the graph represent layer operations and the nodes with rounded corners represent the intermediate tensor buffers that store the tensor data. The input tensor represents a two-dimensional spectrogram with length 32 and eight frequency bins. The architecture consists of three convolutional layers with widely varying characteristics, followed by an average pooling layer and a fully-connected layer.

4.3.1 Memory-aware Scheduling

Figure 4.5 introduces an example DNN graph with two parallel paths, as might be produced by fused tiling. Although the goal of tiling is to produce evenly sized partitions, this cannot always be achieved due to various constraints of the involved DNN operations. For example, convolutions partitioned by 2x2 tiles using FFMT cause very uneven partitions when their feature map size is small and even more so if they are fused with pooling operations. The example demonstrates that the optimal schedule is not trivial. The optimal schedule first

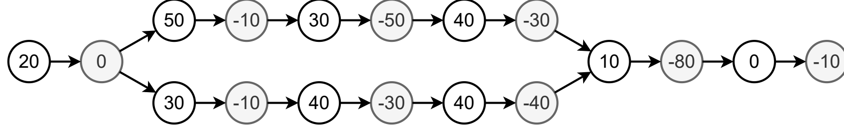


Figure 4.6: Example DNN graph transformed as task graph.

walks via OpA1 to OpB1 but then continues on the lower branch with OpA2, OpB2 and OpC2 before scheduling OpC1 and finally OpD. This optimal schedule has a peak memory of 110: keeping the output of OpB1 (30) alive while computing OpC2 $(40 + 40)$. Simple depth-first scheduling has a peak memory of 120, because it needs to keep the output of OpC1 or OpC2 (40) alive while computing OpC2 $(40 + 40)$ or OpB1 $(50 + 30)$.

Section 3.2.1 introduced existing methods for memory-aware scheduling of DNN graphs. The algorithm of [48] is implemented in the proposed flow because it finds optimal solutions for SP-graphs. However, in contrast to typical task models in distributed computing, the output of a DNN operation can be used by all subsequent operations without distinct buffers for each edge. The DNN graph needs to be adjusted to the model compatible with the algorithm. This adjustment is achieved by duplicating each node and transforming all weights into the *cumulative weight model* (see Section 4 of [48]). The example in Figure 4.5 is transformed into a task graph as shown in Figure 4.6. After duplicating each operation node, the weight of each edge of the original graph is added positively onto the edge source node and negatively onto the duplicated edge sink node. If there are nested parallel graphs, this transformation does not accurately represent the original graph, because each forking node has all accumulated weights of its outgoing edges. An alternative would be to divide all outgoing edge weights of the forking nodes by the number of successors. The underlying issue is that the cumulative weight model cannot represent the fact that the cumulative weight can only be reduced once the last successor edge is scheduled. In these cases, the algorithm may not find the optimal scheduling.

For non-SP-graphs, a mixed integer linear programming (MILP) formulation is given, because it was deemed easier than the method by [3]. The MILP is given as follows.

$$\min_{\mathbf{m},\mathbf{t},\mathbf{H}} \quad max_x(m_x) \tag{4.1}$$

$$s.t. \quad \forall x = 1...N \quad m_x = \sum_{i \in N} H_{x,i} W_i \tag{4.2}$$

$$\forall i = 1...N \quad 1 \leq t_i \leq N \tag{4.3}$$

$$t_i \neq t_j \quad \forall j = 1...i-1 \tag{4.4}$$

$$t_i > t_p \quad \forall p \in pred(i) \tag{4.5}$$

$$t_i < t_s \quad \forall s \in succ(i) \tag{4.6}$$

$$H_{x,i} \in \{0, 1\} \tag{4.7}$$

$$t_i = x \implies H_{x,i} = 1 \tag{4.8}$$

$$t_i < x \wedge \left(\bigvee_{s \in succ(i)} t_s \geq x \right) \implies H_{x,i} = 1 \tag{4.9}$$

The objective function (Eq. 4.1) minimizes the largest sum of buffer sizes W_i that are live (Eq. 4.2) which is equal to the peak memory usage of the schedule. The index x represents the position of execution during the schedule and i is an index for all nodes to be scheduled. To represent when each buffer is executed, the variable vector $\mathbf{t}$ is introduced, with each node i assigned an integer corresponding to its position in the schedule (Eq. 4.3). Ensuring that all indices are different from each other (Eq. 4.4) and respecting their nodes' topological order (Eq. 4.5)(Eq. 4.6), enforces a valid execution order. $\mathbf{H}$ is a Boolean matrix that is forced to true if the buffer i is live during the execution step x. This is the case if a node is currently executed (Eq. 4.8) or if it was already executed and not all of its successors have been executed, yet (Eq. 4.9).

Converting the problem into inequalities and disjunctions yields the following.

$$\begin{aligned}
&min_{m_{max},\mathbf{m},\mathbf{t},\mathbf{H},\mathbf{Z}} && m_{max} && (4.10)\\
&s.t. \quad \forall x = 1...N && m_{max} \geq m_x, \quad m_x = \sum_{i \in N} H_{x,i} W_i && (4.11)\\
&\forall i = 1...N && 1 \leq t_i \leq N && (4.12)\\
&\forall j = 1...i-1 && t_i < t_j \vee t_i > t_j && (4.13)\\
&\forall p \in pred(i), \quad \forall s \in succ(i) && t_p < t_i < t_s && (4.14)\\
& && t_i \leq x \wedge t_i \geq x \rightarrow H_{x,i} = 1 && (4.15)\\
&\forall s \in succ(i) && t_s \geq x \rightarrow Z_{x,i} = 0 && (4.16)\\
& && t_i < x \wedge (1 - Z_{x,i}) \rightarrow H_{x,i} = 1 && (4.17)
\end{aligned}$$

The implications are resolved with $a \rightarrow b \Leftrightarrow \neg a \vee b$ and all comparisons converted to less-than inequalities. The matrix $\mathbf{Z}$ was introduced as Boolean helper variables for converting the conjunction into a disjunction in the next step. The last three equations can then be written as the following.

$$\begin{aligned}
t_i \leq x - 1 \vee -t_i \leq -x - 1 \quad &\vee \quad -H_{x,i} \leq -1 && (4.18)\\
t_s \leq x - 1 \quad &\vee \quad Z_{x,i} \leq 0 && (4.19)\\
-t_i \leq -x \vee -Z_{x,i} \leq -1 \quad &\vee \quad -H_{x,i} \leq -1 && (4.20)
\end{aligned}$$

The disjunctions can be modeled through the *Big M Method* as given in the following [10].

$$\begin{aligned}
\bigvee_{i \in N} A_i x_i \leq U_i \qquad & \forall i \in 1...N && (4.21)\\
\sum_{i \in N} h_i = 1 \qquad & h_i \in \{0, 1\} && (4.22)\\
M h_i + A_i x_i \leq U_i + M \qquad & M \in \mathbb{N} \gg A_i x_i && (4.23)
\end{aligned}$$

A_i are the constraint coefficients, x_i the variables and U_i their upper bound. h_i are Boolean helper variables. This final MILP formulation is now suitable to be plugged into a solver.

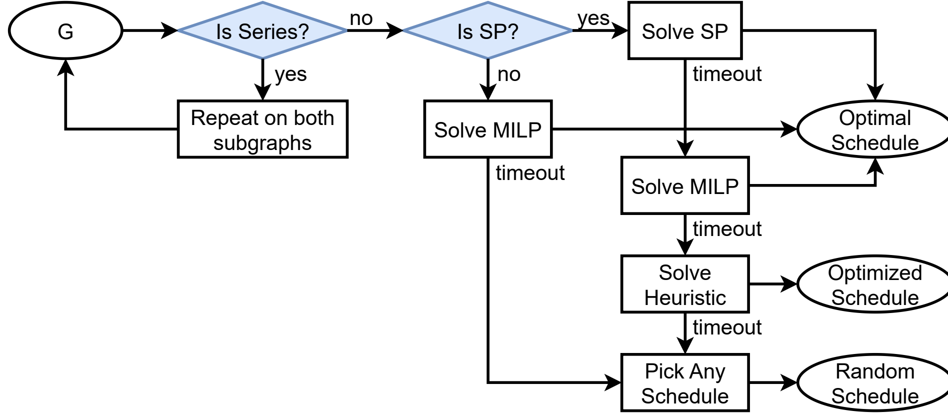


Figure 4.7: Flow of the memory-aware scheduling.

The entire flow of the memory-aware scheduling is shown in Figure 4.7. If the graph can be split into two connected components by removing a single edge, it is a series graph and both split graphs can be scheduled independently to reduce computational complexity. If the graph is series-parallel it is solved with the algorithm presented in [48] and an adjusted task model. Whenever timeouts are hit, the flow progressively falls back to the MILP, a heuristic approach and finally a random schedule. Non-SP-graphs are attempted to be solved with the MILP and fall back to a random schedule on timeout.

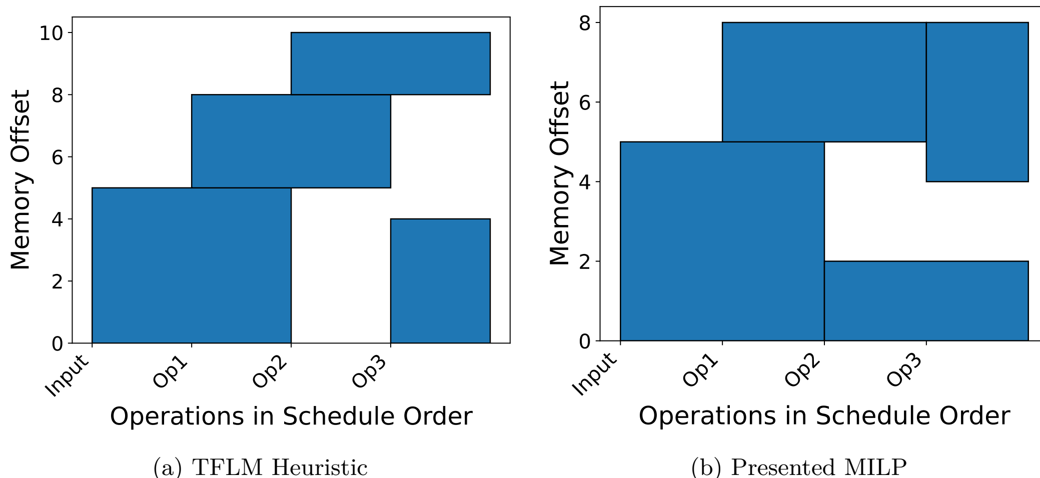
The heuristic approach is based on *hill-valley segments* introduced in [61], but compromising optimality for trivial run time complexity. For each parallel path, the heuristic determines the node $N_{i,max}$ with the maximum memory usage and the node $N_{i,min}$ with the minimum memory usage, which is also a descendant of $N_{i,max}$. The paths are now scheduled in their descending order of $N_{i,diff} = N_{i,max} - N_{i,min}$ and used as is, instead of merging them as in the optimal algorithm.

Running Example:

The running example is a trivial linear graph, so its only possible schedule is the only topological sort of the graph: *Conv1, Conv2, Conv3, Pool, Flatten, Dense.*

4.3.2 Memory Layout Planning

The next step of the proposed automated tiling exploration after scheduling is memory layout planning. Section 3.2.2 introduced existing methods for memory layout planning of DNN graphs. To avoid any compromise of existing methods, a new MILP to optimize memory layout planning is introduced. The DNN graph describes the dependencies between buffers and operations, and the schedule indicates in what order these operations are executed. Together, these two determine the exact lifetime and, therefore, conflicts that exist between any buffers. The following MILP is formulated to provide optimal memory layout planning.



(a) TFLM Heuristic

(b) Presented MILP

Figure 4.8: Example of memory buffer planning.

$$min_{\mathbf{e}} \quad max_i(e_i) \tag{4.24}$$

$$s.t. \quad e_i \geq s_i \quad e_i \in \mathbb{N} \quad \forall_{i=1\ldots N} \tag{4.25}$$

$$e_u - s_u \geq e_v \vee e_v - s_v \geq e_u$$
$$(u, v) \in c_j \quad \forall_{j=1\ldots C} \tag{4.26}$$

The i-th of a total of N buffers has the ending offset e_i and the size s_i. The j-th of a total of C conflicts is described by c_j and contains the indices u and v that refer to the buffer list. The objective function (Eq. 4.24) minimizes the largest ending offset of all buffers, which is equal to the peak memory usage of all mapped buffers. The constraint (Eq. 4.25) ensures that all buffers can only start after the address zero. Finally, the constraint (Eq. 4.26) ensures that there are no address overlaps in the list of conflicting buffers. The nonlinear disjunctions are modeled with the *Big M Method* as already shown for scheduling in the previous section. The final offsets of each buffer are obtained trivially by $e_i - s_i$. For very large DNNs, the run time can become prohibitively high, so that a fallback mechanism is required if a timeout is encountered. The selected fallback mechanism uses a method based on hill-climbing and simulated annealing as implemented in Apache TVM [20].

Figure 4.8 shows a simple example in which the presented MILP provides a better solution than the TFLM heuristic [77]. The x-axis contains all operations in their scheduling order and spans their entire required lifetime. The location of a buffer in memory is described on the y-axis by a starting offset and spanning the buffer size upward. The shown example is a linear graph operating on buffers of sizes 5, 3, 2 and 4 units. The only possible schedule produces trivial conflicts between each neighboring buffer. By greedily placing the two largest buffers of the *Input* and the output buffer of *Op3* at the same offset, the TFLM heuristic is forced to place the two remaining conflicting buffers without any further opportunity of sharing memory locations with other buffers, while the MILP finds such a solution. This reduces the peak memory usage from 10 to 8 units.

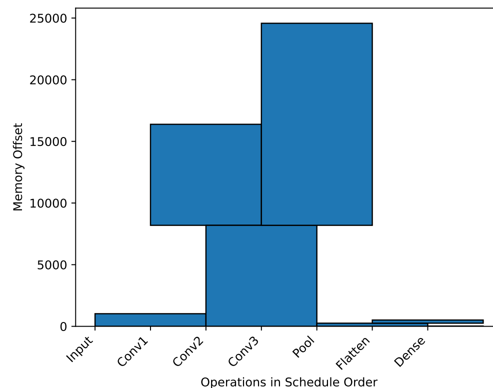


Figure 4.9: Memory layout of the running example.

Running Example:

The running example has a memory layout as shown in Figure 4.9. The total memory usage is 24576 bytes, which is, in the case of a trivial linear schedule, the largest sum of two adjacent buffers: $(16 \cdot 4 \cdot 32 + 16 \cdot 4 \cdot 64) \cdot 4$ bytes. By inspection, the layout indicates that tiling the operations *Conv2* or *Conv3* could reduce memory usage. In the following, a systematic approach to identify such tiling opportunities for complex DNN graphs is presented as next step of the automated tiling exploration.

4.3.3 Block-based Path Discovery

Path discovery has the goal of proposing optimized fused tiling configurations that dictate where and how DNN operations are tiled. The process starts at a *critical* buffer and walks the DNN graph up and down to find suitable split and merge points to discover a tiled path where the critical buffer is split into multiple partitions. After memory-aware scheduling and memory layout planning, the critical buffers are identified by selecting buffers from the memory layout that would reduce the total layout size if their size were to be reduced. This is achieved by checking whether a buffer is actively contributing to the final layout size. In the approach presented in this chapter, the input or output buffers of the model cannot be tiled because they are assumed to be written and read as a contiguous unit by the application. The method can be adapted easily if this requirement would be lifted. All critical buffers are considered for path discovery, but the largest ones are checked first.

Figure 4.10 shows an example memory layout that will be used to describe the algorithm used in the automated tiling exploration flow. First, all buffers that end at the total layout size are the starting point of this search. Here, the layout ends at offset 100 with the buffers `B1`, `B5` and `B8`. For each of these starting points, all buffers are collected that end at the exact offset that they start at and are associated to the starting point as a chain. If there are no further buffers at the starting offset, the entire chain up to the starting point is discarded. This step is repeated until there are no chains left or the offset zero is reached. `B1` starts at offset 40, but there is no buffer that ends at this offset, so this chain is discarded. `B8` starts

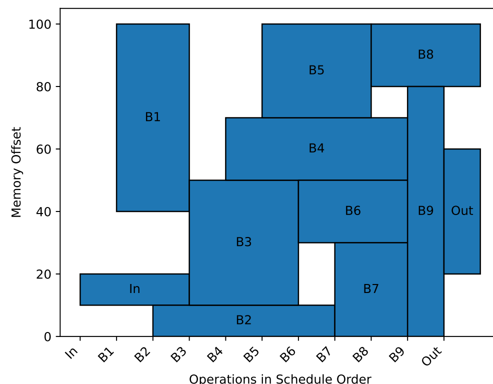


Figure 4.10: Example layout for critical buffer selection.

at offset 80, where B9 ends. Since B9 starts at offset zero, this chain is complete and its buffers defined as critical buffers. B5 starts at offset 70, where B4 ends. But since B4 does start at offset 50, the search must continue. At offset 50, two different buffers B3 and B6 end, so the chain is forked into B5-B4-B3 and B5-B4-B6. B3 starts at offset 10, where only B2 ends and B6 starts at offset 30, where only B7 ends. Since both B2 and B7 start at offset zero, two additional chains are completed at all their buffers are added as critical buffers. Finally, all buffers collected in all chains that reached the offset zero are collected. Here, these are B8, B9, B5, B4, B3, B2, B6 and B7. These are the critical buffers and are reported in descending sorted order by their size. In this case, the path discovery would start by considering B9 first because it is the largest critical buffer.

It is not detrimental if too many buffers are identified during this step because the only consequence is an increase in run time of the exploration flow. In fact, it is practical to implement a cutoff size (e.g., 5% of the largest buffer) to prevent testing tiny buffers that are extremely unlikely to reduce peak memory if they were tiled.

Running Example:

For the running example layout from Figure 4.9, the critical buffers are the output buffers of *Conv2* and *Conv3*. Reducing the size of any other buffer in isolation would not affect the total memory usage. Since the output of *Conv3* is the largest of the two buffers, it is selected as the first critical buffer. Buffers that are inputs or outputs to the model are assumed to be user-provided buffers and, therefore, must be excluded because they would not be able to be split.

The next step of path discovery is the formation of paths by associating DNN graph operations to different types of blocks and tracking all possible configurations. Figure 4.11 shows all blocks of the presented block-based path discovery along with their supported operations. Each block has two terminals representing the way their input and output tensors are split. The terminal types are defined as follows.

- Path start (I): This marks the start of any path where a buffer is split into multiple

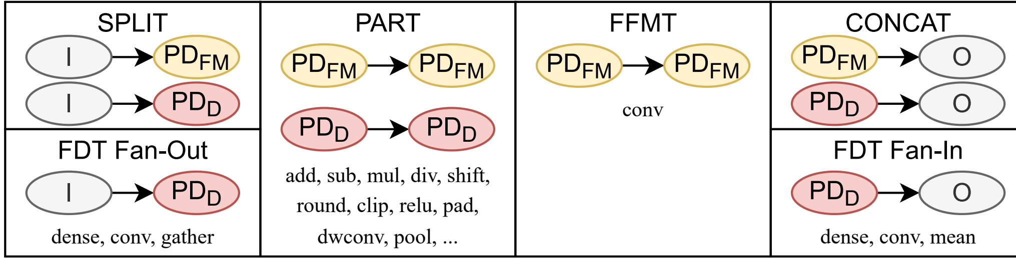


Figure 4.11: Path discovery building blocks and supported operations.

partitions that can be executed independently.

- Path end (O): This marks the end of any path where a buffer is recombined.
- Buffer partitioned depthwise (PD_D): These buffers are split along their depth dimension. For three-dimensional tensors that would be the axis representing the channels. However, this type may also be applied to tensors of other dimensions.
- Buffer partitioned by feature maps (PD_{FM}): These buffers are split along their feature maps for three-dimensional tensors. The two-dimensional feature maps may be split along one or both of their axes.

The blocks themselves are defined as follows.

- Explicit split ($SPLIT$): A trivial operation that divides its input into the chosen number of partitions along a given axis. This operation is not required if an implicit split is realized through FDT. This block may produce depthwise partitioned values ($\mathrm{PD_D}$) or feature map partitioned values ($\mathrm{PD_{FM}}$).
- FDT Fan-Out ($FDTO$): The operation is split by only computing some of the output values for each partition. For convolutions, only the depthwise (or channel) dimension is split to support the operator fusion of FDT. Serves as an implicit split operation.
- Partitioned operations ($PART$): Some output values are computed by using their respective input values. Applicable to any element-wise operation, because all values can be computed independently.
- Concatenation operation ($CONCAT$): This operation concatenates multiple inputs from partitioned operations back into the original non-partitioned buffer O.
- FDT Fan-In ($FDTI$): The operation is split by using only some of the input values to compute all of the output values. This results in output values that are only a part of the total sum of the actual output and which must be combined into the original non-partitioned buffer O again. Therefore, this operation also includes the final merge operation that performs an element-wise summation as discussed in Section 4.2.

- Fused Feature Map Tiling (*FFMT*): Only applicable to convolutional operations. The partitions form tiles that have the same size on every feature map. Each such tile can be computed independently of the others. If the kernel size is larger than 1x1, the input tiles need to be larger and overlap with each other, because the convolution has data dependencies into neighboring tiles. Recomputing these values also introduces a computational overhead.

FDT Fan-Out, *PART*, *FDT Fan-In* and *FFMT* replace their original operation with the tiled variant, while *SPLIT* and *CONCAT* are additionally inserted operations to build a valid path.

At the critical buffer, multiple candidate paths are proposed for type PD_D or PD_{FM} if possible. One proposal is created for each number of partitions $N \in \{2, ..., 25\}$ with the upper limit chosen to reduce overheads while observing that higher limits rarely provide additional memory savings. For *FFMT*, quadratic two-dimensional tiling configurations are added as $N \in \{2x2, 3x3, 4x4, 5x5\}$. Next, the path is discovered starting from the critical buffer in both directions, where any compatible block can be chosen. Whenever the *FDT Fan-In* method is used, one version of the path without *FDT Fan-In* is kept, because a *CONCAT* could require less memory than continuing with partial values. Whenever an *FFMT*-partitioned operation that has overlap is encountered, one version that stops before that operation is kept and finalized with *SPLIT* or *CONCAT*. This is done because overlaps that become too large may cause inferior paths compared to shorter ones. The discovery has to stop at any operation that is incompatible with fused tiling (e.g. softmax, slice, concat). For each of the proposed path candidates, the operation before the critical buffer with the lowest input buffer size is selected as start of the path and the operation after the critical buffer with the lowest output buffer size is selected as end of the path. If no such operation could be determined before and after the critical buffer, the path is discarded and if no valid paths are left, the discovery fails. In the final step, path discovery determines the path that is expected to cause the lowest memory usage. As mentioned in the overview, this is done by evaluating the memory size with memory-aware scheduling and memory layout planning. The best configuration is the one with the lowest memory size.

Running Example:

The steps of path discovery will be exemplified with the model in Figure 4.4. Since the output buffer of *Conv3* was selected as the first critical buffer, this is the starting point of the path discovery. The following candidate paths will be created from this 16x4x64 buffer.

- PD_D along the third axis with N=2 to N=25
- PD_{FM} along the first axis with N=2 to N=16
- PD_{FM} along the second axis with N=2 to N=4
- PD_{FM} along the first and second axis with N=2x2 to N=4x4

Moving upward in the dataflow, the *Conv3* operation must now be mapped to a compatible block type. Since this is a convolutional operation, the only possible

choice for all PD_D candidates is FDT Fan-Out which has an input terminal of type I, signifying the start of the path. For all PD_{FM} candidates, *FFMT* is the only choice for the *Conv3* operation, and this continues up until the 32x8x1 input buffer of the model. However, as already mentioned, every time an *FFMT* operation with overlap is encountered, another path candidate is created that terminates the path with *SPLIT*. Overlap is caused by operations whose kernel size is larger than the stride size in the dimensions that are partitioned. This is the case for *Conv2* and, if split along the first axis, *Conv1*. At this point, the following path candidates have been created.

- [FDTO] along the third axis with N=2 to N=25
- [SPLIT,FFMT] along the first axis with N=2 to N=16, along the second axis with N=2 to N=4 and along both axes with N=2x2 to N=4x4
- [SPLIT,FFMT,FFMT] along the first axis with N=2 to N=16 and along both axes with N=2x2 to N=4x4
- [SPLIT,FFMT,FFMT,FFMT] along the same axes as in the second bullet

Note that, for example, for the path candidates of the last bullet, the input is not simply split by the number of partitions, but has accumulated overlap. For example, with N=2 along the second axis, the first partition ranges from row 1 to 6 and the second partition ranges from row 3 to 8. Moving downward in the dataflow, for the FDTO paths, the *Pool* and trivial *Flatten* operations are matched with the *PART* block and the final *Dense* operation is matched to the *FDT Fan-In* block that finalizes the paths. All FFMT paths encounter an issue when processing the *Pool* operation, because its output is of size 1x1. Feature maps of size 1x1 cannot be split by *FFMT* and all FFMT paths are discarded because they are unable to tile the critical buffer. The final remaining path candidates are therefore the following.

- [FDTO,PART,PART] along the third axis with N=2 to N=25
- [FDTO,PART,PART,FDTI] along the third axis with N=2 to N=25

Path pruning will reduce the paths from the first bullet to [FDTO,PART] because *Pool* and *Flatten* have the same output size. Next, the memory layout of all paths is evaluated, and the shortest path with the least number of partitions is selected as the best path. In this case, once N reaches 3, the critical buffer is no longer relevant for the maximum memory usage and all layout sizes are the same. Therefore, the path [FDTO,PART] with N=3 is selected as the best path. Figure 4.12 shows the final DNN architecture of the running example after applying the transformation given by this path. Figure 4.13 shows the final memory layout that is obtained after the transformation. The total layout size is given as the sum of the output buffers of *Conv1* and *Conv2* as $(16 \cdot 4 \cdot 32 + 16 \cdot 4 \cdot 32) \cdot 4 = 16384$ bytes. As can be seen, due to the split critical buffer, the memory demand could be significantly reduced.

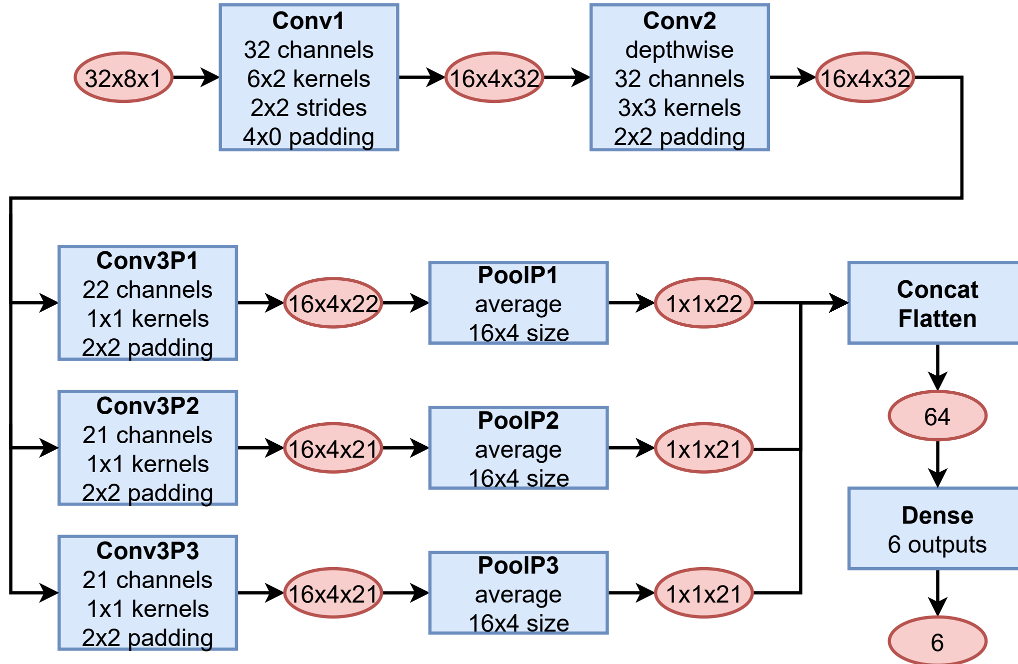


Figure 4.12: Architecture of the running example after transformation.

4.3.4 Automated Graph Transformation

Once the best path configuration has been determined, it is applied by transforming the DNN graph with the given parameters. At the start of the split path, either an explicit or implicit split has to be realized. For an explicit split, a new operation has to be inserted that slices the input into partitions according to the tiling configuration. An implicit split is implemented by replicating the convolutional or dense layer by the number of partitions and splitting their weight dimension that is responsible for producing outputs. Any following operations are also replicated on each partition and need their parameters changed to match their new input dimensions. For example, a bias addition no longer adds its original constants, but only the ones corresponding to the respective partition. Another example are padding operations where their padding needs to be eliminated at split boundaries to preserve the original DNN behavior. Depthwise convolutions can be split trivially along the channel dimension as tiling method *PART*, since every output channel only depends on its respective input channel. The associated filter weights must still be split accordingly. The exact splitting logic for every operator has to be determined on a case-by-case basis. However, it is possible to define categories with similar splitting logic. *FDT Fan-In* operations are split equivalently to *FDT Fan-Out* ones, just that the input channel dimension of the weight tensor is split. Care has to be taken to prohibit automatic fusing of the last operations on the split paths with the *CONCAT* or *FDT Fan-In* operation, because that would lead to keeping their inputs alive on multiple split paths. After all transformations have been applied to the graph, the flow goes back to scheduling it as shown in Figure 4.1.

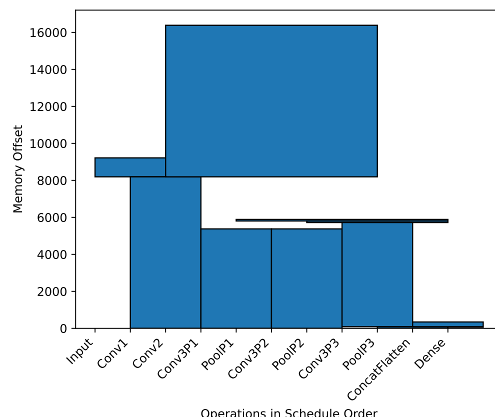


Figure 4.13: Memory layout of the running example after transformation.

Running Example:

Figure 4.12 shows multiple aspects of the automated graph transformation. The *Conv3* operation is replicated three times to realize an implicit split. All weights and biases are split along the depthwise axis to process 22, 21 and 21 channels, respectively. The *Pool* operations can be replicated as is because they do not have any parameters relevant to the split dimension. Their only difference from the original operation is that they operate on a reduced number of channels. The automated tiling exploration would now continue by analyzing this new transformed graph again for critical buffers. However, in the running example, the flow stops because the output buffers of *Conv1* and *Conv2* cannot be tiled further since they do not have a straight-line successor buffer with smaller size.

4.3.5 Implementation

The complete end-to-end flow to compare FDT to FFMT has been implemented in *Apache TVM* [20] which was introduced in Section 2.3. As mentioned, TVM fuses many DNN operations to completely eliminate intermediate buffers. Therefore, when analyzing a DNN for critical buffers, only the buffers of non-fused operations are taken into consideration. However, during path discovery, all fused operations are transformed into fine-grained operations because they may contain operations that are suitable as terminals of the split path. Otherwise, a single operation that would not be supported or beneficial in the split path, would cause the entire operation to not be eligible for the path, even when large parts of it could have been valid. For example, there might be an operation that is tiled with FFMT that is preceded by a fused operation that would not improve the FFMT partitioned section. However, when that preceding fused operation has a widening cast fused as the last individual operation, that widening cast could become a beneficial inclusion to the FFMT partitioned section. By including the cast in the split path, it possibly contributes to a configuration with a smaller memory layout than without its inclusion. Another consider-

ation with fused operations should be respected while deciding the optimal path split and merge positions. In case there are any ties in operation sizes, it may be beneficial to respect the original boundaries between fused operations to eliminate the need for an additional intermediate buffer. After the tiling transformation is applied, the model representation is fused again for the next round of analysis, starting with scheduling. Operations that could be carried out as simple in-place operations without any intermediate buffer do not need special treatment, because they will always be fused with neighboring operations with more complex dataflow.

Some models that were imported into TVM contain *reshape* operations that do not have any computational behavior beyond an exact copy. They are either inserted for reasons of type system integrity or leftovers from optimization passes. Since they introduce unwanted and unnecessary intermediate buffers, the implementation removes or ignores them.

When concatenating tensors in TVM, the arguments of the concatenation operation need to be given as a data type called tuple. After applying the final operator fusion pass again, this resulted in an issue with prolonged lifetimes of intermediate buffers. By making the entire tuple object a parameter of the fused function, the entire tuple was kept alive until all partitions had been merged. The remedy was a special transformation pass that hoists the extraction of tuple elements out of the fused function, so that the parameter of the fused function can instead be just the tuple element. In this way, each finished partition reduces the amount of tuple data that needs to be kept alive.

Another consideration for the implementation is where in the machine learning compilation flow to apply the analysis and transformations. Early in the flow, the Relay IR is very abstract, high-level and not optimized. Although this simplifies the analysis and transformation, it is also farthest from the actual low-level machine code deployed. Any optimizations made at this stage could be less relevant or effective for the final memory usage. Late in the flow, the TIR IR is very target-specific, low-level and highly optimized. Therefore, the memory optimizations will be more accurately mapped to the final memory usage, but the analysis and transformation become exceedingly complex and intricate. The implementation of the ideas in this chapter tries to find a good trade-off between these two extremes to produce accurate results with manageable effort. That is, all analysis and transformations are applied after high-level graph optimizations, but before target-specific optimizations. In TVM terms, this is just before the lowering from the Relay IR to the TIR IR.

Running Example:

The following code shows the Relay IR of the running example.

```
1  def @main(%input_1: Shape(1, 32, 8, 1)) -> Shape(1, 6) {
2    %0 = nn.conv2d(%input_1, Const[0]: Shape(6, 2, 1, 32),
3      strides=[2, 2], padding=[2, 0, 2, 0], channels=32,
4      kernel_size=[6, 2]) -> Shape(1, 16, 4, 32);
5    %1 = add(%0, Const[1]: Shape(32)) -> Shape(1, 16, 4, 32);
6    %2 = nn.relu(%1) -> Shape(1, 16, 4, 32);
7    %3 = nn.conv2d(%2, Const[2]: Shape(3, 3, 32, 1), padding=[1, 1, 1, 1],
8      groups=32, channels=32, kernel_size=[3, 3]) -> Shape(1, 16, 4, 32);
9    %4 = add(%3, Const[3]: Shape(32)) -> Shape(1, 16, 4, 32);
10   %5 = nn.relu(%4) -> Shape(1, 16, 4, 32);
11   %6 = nn.conv2d(%5, Const[4]: Shape(1, 1, 32, 64), padding=[0, 0, 0, 0],
12     channels=64, kernel_size=[1, 1]) -> Shape(1, 16, 4, 64);
13   %7 = add(%6, Const[5]: Shape(64)) -> Shape(1, 16, 4, 64);
```

```
14    %8 = nn.relu(%7) -> Shape(1, 16, 4, 64);
15    %9 = nn.avg_pool2d(%8, pool_size=[16, 4], strides=[16, 4],
16      padding=[0, 0, 0, 0]) -> Shape(1, 1, 1, 64);
17    %10 = reshape(%9, newshape=[-1, 64]) -> Shape(1, 64);
18    %11 = nn.dense(%10, Const[6]: Shape(6, 64), units=6) -> Shape(1, 6);
19    nn.relu(%11) -> Shape(1, 6);
20  }
```

The DNN model is defined as a function `main` that takes the input `input_1` with shape 1x32x8x1 and produces an output with shape 1x6. It is not shown for simplicity, but all sequences of `nn.conv2d` (convolutional layer), `add` (bias addition) and `nn.relu` (ReLU activation function) are fused. As already shown in Figure 4.12, the desired split occurs at the third convolution (lines 11-12) and merges just after the pooling operation (lines 15-16). The following code shows the example model after the optimization transformations have been applied.

```
1   def @main(%input_1: Shape(1, 32, 8, 1)) -> Shape(1, 6) {
2     %0 = nn.conv2d(%input_1, Const[0]: Shape(6, 2, 1, 32),
3       strides=[2, 2], padding=[2, 0, 2, 0], channels=32,
4       kernel_size=[6, 2]) -> Shape(1, 16, 4, 32);
5     %1 = add(%0, Const[1]: Shape(32)) -> Shape(1, 16, 4, 32);
6     %2 = nn.relu(%1) -> Shape(1, 16, 4, 32);
7     %3 = nn.conv2d(%2, Const[2]: Shape(3, 3, 32, 1), padding=[1, 1, 1, 1],
8       groups=32, channels=32, kernel_size=[3, 3]) -> Shape(1, 16, 4, 32);
9     %4 = add(%3, Const[3]: Shape(32)) -> Shape(1, 16, 4, 32);
10    %5 = nn.relu(%4) -> Shape(1, 16, 4, 32);
11    %6 = nn.conv2d(%5, Const[4]: Shape(1, 1, 32, 22), padding=[0, 0, 0, 0],
12       channels=22, kernel_size=[1, 1]) -> Shape(1, 16, 4, 22);
13    %7 = add(%6, Const[5]: Shape(22)) -> Shape(1, 16, 4, 22);
14    %8 = nn.relu(%7) -> Shape(1, 16, 4, 22);
15    %9 = nn.avg_pool2d(%8, pool_size=[16, 4], strides=[16, 4],
16      padding=[0, 0, 0, 0]) -> Shape(1, 1, 1, 22);
17    %10 = nn.conv2d(%5, Const[6]: Shape(1, 1, 32, 21), padding=[0, 0, 0, 0],
18       channels=21, kernel_size=[1, 1]) -> Shape(1, 16, 4, 21);
19    %11 = add(%10, Const[7]: Shape(21)) -> Shape(1, 16, 4, 21);
20    %12 = nn.relu(%11) -> Shape(1, 16, 4, 21);
21    %13 = nn.avg_pool2d(%12, pool_size=[16, 4], strides=[16, 4],
22      padding=[0, 0, 0, 0]) -> Shape(1, 1, 1, 21);
23    %14 = nn.conv2d(%5, Const[8]: Shape(1, 1, 32, 21), padding=[0, 0, 0, 0],
24       channels=21, kernel_size=[1, 1]) -> Shape(1, 16, 4, 21);
25    %15 = add(%14, Const[9]: Shape(21)) -> Shape(1, 16, 4, 21);
26    %16 = nn.relu(%15) -> Shape(1, 16, 4, 21);
27    %17 = nn.avg_pool2d(%16, pool_size=[16, 4], strides=[16, 4],
28      padding=[0, 0, 0, 0]) -> Shape(1, 1, 1, 21);
29    %18 = annotation.stop_fusion(%9) -> Shape(1, 1, 1, 22);
30    %19 = annotation.stop_fusion(%13) -> Shape(1, 1, 1, 21);
31    %20 = annotation.stop_fusion(%17) -> Shape(1, 1, 1, 21);
32    %21 = (%18, %19, %20) -> Tuple(
33      Shape(1, 1, 1, 22), Shape(1, 1, 1, 21), Shape(1, 1, 1, 21));
34    %22 = concatenate(%21, axis=3) -> Shape(1, 1, 1, 64);
35    %23 = reshape(%22, newshape=[-1, 64]) -> Shape(1, 64);
36    %24 = nn.dense(%23, Const[10]: Shape(6, 64), units=6) -> Shape(1, 6);
37    nn.relu(%24) -> Shape(1, 6);
38  }
```

The convolution is replicated in lines 11-12, 17-18 and 23-24. Note that the weight tensor argument is split along the output channel axis into sizes 22, 21 and 21. The `channels` argument is also changed accordingly. The fused `add` and `nn.relu`, and the final `nn.avg_pool2d` (average pooling layer) are also replicated in lines 13-16, 19-22 and 25-28. Note that the bias vector arguments of the broadcasting additions are also split into three parts. Lines 29-31 are annotations that prevent fusion of its

input and output operations. Finally, lines 32-34 concatenate all partitioned values back together before the unmodified remainder of the model finishes execution.

4.4 Experimental Results

From a wide range of models, the following subset was identified that benefits from fused tiling.

1. *Audio Wake Words (AWW)*: Detection of keywords from audio. Part of the MLPerf Tiny benchmark [11].
2. *Text Sentiment Analysis (TXT)*: [34, 63].
3. *Magic Wand (MW)*: TinyML gesture recognition with an accelerometer [28].
4. *PoseNet (POS)*: Pose estimation [72].
5. *MobileNet V2 SSDLite (SSD)*: COCO classifier [84].
6. *Cifar10 classifier (CIF)*: Own CNN [50].
7. *Radar Gesture Recognition (RAD)*: Own TinyML CNN for gesture recognition with a radar sensor [45].

The target architecture for all experiments was RISC-V in the *RV32GC* configuration. The GNU toolchain at version 11.1.0 was used with the optimization flag set to `-Os` and options to prune all unused code and data. RAM and ROM usage is determined from the section sizes in the compiled binary. The run time is estimated by counting the number of MAC operations required in their final optimized DNN graph. This is not equivalent to the run time after deployment, but is sufficient for a comparison. The MILPs were implemented in OR-Tools 9.3 [76] using the Gurobi 9.1.2 solver [38].

4.4.1 Automated Tiling Exploration

The presented optimal memory layout planning algorithm using an MILP was compared to the best-performing heuristic approach in TVM that uses hill-climbing and simulated annealing. The heuristic finds the optimum for most models, but in one case (the TXT model), the presented MILP approach achieved a memory reduction of 16.8%.

The presented MILP memory-aware scheduling solution is optimal, as defined by its cost function. The work in [3] reports a run time of 37.9 seconds for the SwiftNet model [21]. When running the presented MILP scheduling with the same SwiftNet model, a run time of 37 seconds was measured on an *AMD Ryzen 9 3900X* processor. Although these results are not directly comparable because different machines were used, they show comparable performance.

The presented path discovery is able to traverse a large variety of models and selects the optimal solution within its search space. This search space ranges from zero to hundreds

Table 4.1: Memory reduction of FDT compared to FFMT

	Mem [kB]			[%]		MACs [1 million]			[%]	
Model	Untiled	FFMT	FDT	FFMT Savings	FDT Savings	Untiled	FFMT	FDT	FFMT Overhead	FDT Overhead
AWW	65.6	65.6	53.7	0.0	18.1	2.66	2.66	2.66	0.0	0.0
TXT	18.6	18.6	4.4	0.0	76.2	0.00	0.00	0.00	0.0	0.0
MW	17.6	6.8	11.3	61.3	35.5	0.06	0.08	0.06	32.5	0.0
POS	9.35k	5.11k	8.94k	45.3	4.4	837	1215	837	45.1	0.0
SSD	14.3k	8.66k	12.2k	39.4	14.6	313	314	313	0.2	0.0
CIF	157	60	148	61.8	5.7	4.55	5.09	4.55	12.0	0.0
RAD	35	27	28	22.0	19.6	0.09	0.09	0.09	0.0	0.0
Avg.				32.8	24.9				12.8	0.0

depending on the critical buffer dimensions and operations used to create a path. Further factors are variants with early path stops and the iterative application of tiling. The innermost operations of graph transformation, scheduling and layout planning have to be executed that number of times. For the evaluated models, the entire flow has a run time of 3 minutes for the RAD model (38 tiling configurations) up to an hour for the POS model (172 tiling configurations). [13, 67, 68, 57, 25] do not provide run times of their flow. The work of [24] reports 82 to 375 seconds to search nine configurations, while still having to manually select the number of partitions and their axes. This shows that the implemented flow runs efficiently and, in contrast to existing work, does not require a manual choice for the tiling configuration.

4.4.2 Fused Depthwise Tiling

The results in Table 4.1 show the working memory (RAM) usage and the estimated MAC operations for each untiled network and the improvements by applying FFMT or FDT individually. The first two models can only be tiled by FDT. In the case of AWW, the critical buffer is involved in a sequence of convolutions that reduce the size of the feature map to 1x1, which cannot be split by FFMT. The critical buffer of the TXT model exists within an embedding lookup followed by a mean axis reduction that can only be tiled by FDT. The remaining models are all CNNs with sufficient feature map sizes such that either method is applicable. FDT eliminates run time overheads at the cost of lower memory reduction compared to FFMT. The average memory savings are 32.8% for FFMT and 24.9% for FDT, with the highest savings achieved for the TXT model with FDT at 76.2%. The average run time overhead is 12.8% for FFMT when including the models where it did not achieve any memory savings, whereas FDT requires no overhead as expected.

Enhancing an FFMT-only TinyML deployment flow with FDT expands the tiling design space for memory and performance goals, which is shown in Table 4.2. In the case of a memory-optimized design, the fused tiling method with the highest memory savings was selected. This selection results in an improvement of average memory savings from 32.8% to 46.3% with an unchanged run time overhead of 12.8% compared to an FFMT-only flow. In the case of a performance-optimized design, the highest memory savings were selected with the constraint that the run time overhead may not exceed 1%. This still resulted in an

Table 4.2: Tiling Design Space Exploration with FFMT and FDT

	Memory Optimized			Performance Optimized		
Model	Method	Mem Savings	Perf Overhead	Method	Mem Savings	Perf Overhead
AWW	FDT	18.1	0.0	FDT	18.1	0.0
TXT	FDT	76.2	0.0	FDT	76.2	0.0
MW	FFMT	61.3	32.5	FDT	35.5	0.0
POS	FFMT	45.3	45.1	FDT	4.4	0.0
SSD	FFMT	39.4	0.2	FFMT	39.4	0.2
CIF	FFMT	61.8	12.0	FDT	5.7	0.0
RAD	FFMT	22.0	0.0	FFMT	22.0	0.0
Avg.		46.3	12.8		28.8	0.0

Table 4.3: ROM usage of FDT compared to FFMT.

	ROM [kB]			Overhead [%]	
Model	Untiled	FFMT	FDT	FFMT	FDT
AWW	126	126	124	0	-1.8
TXT	698	698	699	0	0.1
MW	73.7	79.0	75.6	7.3	2.7
POS	13.6k	13.4k	13.4k	-1.2	-1.1
SSD	6.44k	6.38k	6.38k	-0.8	-0.8
CIF	546	548	548	0.4	0.4
RAD	226	234	226	3.2	-0.3

average memory savings of 28.8% and FDT is selected for five of seven models. The exploration also found tiling configurations, in which FFMT and FDT are applied in conjunction. However, in the best case the results were as good as the best configuration with a single tiling method. Still, for possible new models, the combination could also yield benefits.

As can be seen in Table 4.3, the ROM overhead of FDT is negligible with the highest increase of 2.7%. The increase is caused by an increase in code size due to the increased number of operations in the DNN graph, but often the ROM usage of FDT is also lower because TVM may choose simpler schedules than for the untiled variant.

4.5 Summary

This chapter presented the novel fused tiling method FDT for memory optimization in DNN inference. FDT can be applied to more layer types than FFMT and does not introduce overhead from overlapping partitions. The method is evaluated in a complete end-to-end DNN deployment flow for an accurate evaluation. The flow combines memory-aware scheduling, memory layout planning and block-based path discovery to a fully automated deployment. New MILP formulations have been presented for scheduling and layout planning that can effectively find optimal solutions.

Chapter 5

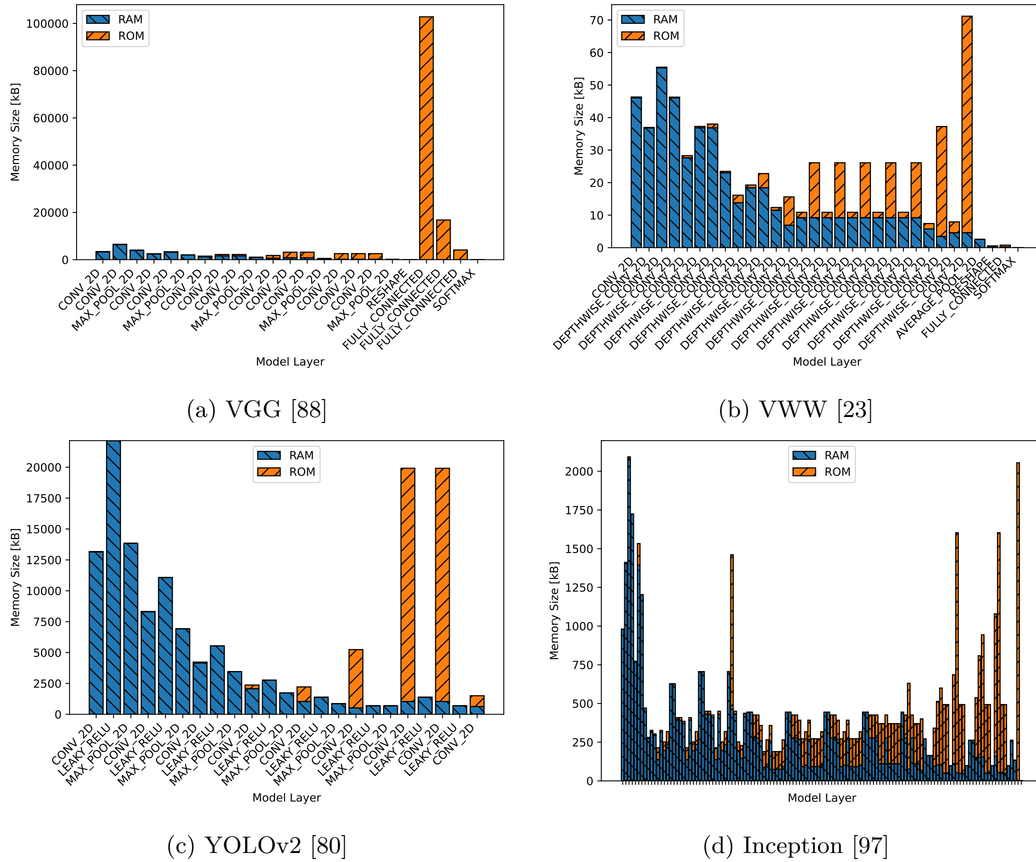
Optimization of Memory and Communication in Distributed DNN Inference

ANOTHER field that can greatly benefit from fused tiling is distributed inference. Distributed inference refers to the splitting of the inference task across distinct compute devices with separate processor cores and memories. This chapter presents contributions that enable fully distributed inference on constrained devices with a focus on memory limitations.

5.1 Motivation

As already outlined in the previous chapters, DNN inference is heavily constrained by memory sizes because it must store a large amount of input, parameter and intermediate data. Fused tiling, as presented in the previous chapter, is able to reduce the amount of intermediate data, but can not help with the parameters if executed on a single device. To perform the entire inference task, a single device must have all of the model parameter data available to it. Therefore, given an application that requires a model with a sufficiently large amount of parameter data, that application cannot be run on a single device. While this could be solved trivially by deploying the application on a more powerful device, the cost of such a solution is prohibitive and may not be appropriate for the target application. If the system is battery-powered, the increase in energy consumption could exceed the specified energy budget. Another issue with this approach is the cost of upgrading as soon as existing hardware falls below the required memory threshold for an updated application.

Section 3.3 already introduced existing work on distributed inference. The most closely related work is MoDNN [64] which partitions both feature and weight data. Their method for partitioning weights is, however, limited to sparse fully-connected layers. It would be possible to combine their method with the layer fusion and optimization methods presented in this chapter. With additional constraints, the communication overhead could be further reduced by the approaches presented in this chapter.



(a) VGG [88]

(b) VWW [23]

(c) YOLOv2 [80]

(d) Inception [97]

Figure 5.1: Memory requirements for the computation of individual model layers.

The work presented in this chapter builds on the previous DeepThings approach [106], which addressed adaptive distributed deep learning inference for systems with dynamic availability of edge nodes. A fusing approach for multiple layers, which focuses on data partitioning in feature-intensive layers, was the main contribution of that work. However, it did not consider weight partitioning. This is an important consideration because, given a sufficiently deep DNN, it is no longer possible for the presented approach to store a large volume of weight data on a single resource-constrained device after a certain layer depth. DeepThings requires that weight-intensive network layers are evaluated on a central powerful gateway edge device that has sufficient memory. This constraint is removed by the approach presented in this chapter, enabling fully distributed inference on a set of memory-constrained edge devices.

5.2 Contribution

A typical DNN structure starts with input data that has large width and height dimensions and few channels, for example, an image. The inputs are processed by a number of convolutional layers that apply multiple filters to produce a set of feature maps. These filters are typically trained to extract certain features or characteristics that are then represented in the feature maps. With a high number of filters in each layer, the number of feature maps grows with each convolutional layer. At the same time, interspersed pooling layers shrink the width and height of feature maps to keep the total size of the intermediate tensors manageable. Figure 5.1 shows the memory requirements of typical DNNs and outlines the share of memory that is working memory (RAM) and storage memory (ROM) for each individual layer. The layers on the x-axis are sorted in topological order. Working memory is occupied by input/output data and intermediate buffers, collectively also called feature data. The storage memory holds the weights and biases associated with each layer. The typical model architecture starts out with large feature data that is progressively reduced in size by consecutive convolutional and pooling layers. The latter layers either consist of fully-connected layers or convolutional layers with a large number of weights. Because of this structure, the feature data dominate the memory usage for the first layers of a DNN, while in the latter layers, the weights dominate.

The previous DeepThings work introduced a method for memory- and communication-aware partitioning and fusing of feature-dominated convolutional layers [106]. DeepThings is extended with methods to partition and fuse convolutional and fully-connected layers whose weight data size dominates their feature data size. This extended comprehensive approach for the distributed inference of complete DNNs considers all layer types while simultaneously optimizing for computation, memory and communication demands. The computation and memory footprint of processing and storing feature and weight data is evenly distributed across all devices, so that the DNN inference task can be scaled down for resource-constrained edge devices. The full distribution is achieved by combining the existing DeepThings approach for partitioning the feature-dominated layers with a new partitioning method for partitioning the weight-dominated layers, no matter whether they are convolutional or fully-connected layers. Both partitioning methods make use of fused tiling in the form of FFMT and FDT that were described in the previous chapter. In the context of distributed inference, the fusing aspect is used to reduce the communication demand between the cooperating devices. In addition, an approach that is capable of minimizing the memory footprint of a full DNN model by finding the optimal point at which to switch from feature partitioning to weight partitioning will be presented. Furthermore, an approach is presented that minimizes the communication overhead by finding the optimal configuration for partitioning the weight-partitioned layers. These methods are deployed and evaluated on a real-world edge device setup that performs DNN inference. Four different DNNs are explored in a case study on a Raspberry Pi cluster with regard to the trade-offs between run time, memory requirements and communication overhead for different network bandwidths and device counts.

5.3 Methods for DNN Partitioning

To formulate optimal methods to partition a DNN, the following metrics are first formulated on top of the notation introduced in Chapter 2. The number of weights Q_l denotes the number of parameter data values required by a layer l. The number of multiplication operations of a layer l is denoted as R_l and serves as an approximate metric for the computation time. For fully-connected layers with input size M_l and output size K_l, these two metrics are simply defined as follows.

$$Q_l = R_l = M_l \cdot K_l \tag{5.1}$$

For convolutional layers with input size $X_l \times Y_l \times C_l$, output size $X_l \times Y_l \times O_l$ and kernel size $U_l \times V_l$, the metrics can be calculated as follows.

$$\begin{aligned} M_l &= X_l \cdot Y_l \cdot C_l, & K_l &= X_l \cdot Y_l \cdot O_l \\ Q_l &= U_l \cdot V_l \cdot C_l \cdot O_l, & R_l &= X_l \cdot Y_l \cdot U_l \cdot V_l \cdot C_l \cdot O_l \end{aligned} \tag{5.2}$$

The sum of all weights and the total computational load are trivially defined as follows.

$$\sum_l Q_l \quad \sum_l R_l \tag{5.3}$$

Given resource-constrained edge devices that are not able to handle either of these sums, it is required that one or more fully-connected or convolutional layers are distributed to enable execution of such a DNN. As was already pointed out, for fully-connected layers and latter convolutional layers, the number of weights dominates the total memory requirements of those layers. To explore different distributed inference schemes, the metric F_n is introduced, which denotes the number of weights to be stored on the n-th device. The computational load in distributed inference results predominantly from the multiplications R_l performed on these weights. However, distributing the inference across multiple devices introduces potential for parallel execution. The longest path of execution, given by the number of sequential multiplications on the slowest path across all devices, is denoted T.

Distributing the work over multiple nodes requires some form of coordination that incurs a communication load. This load, denoted C with the unit *number of feature data values*, is the result of the exchange of feature data between devices. The value of C is only an estimate of the exact communication impact on a real DNN inference implementation because it does not take into account any protocol overhead from the application layer distributed inference protocol down to the physical layers. Nonetheless, the experimental results show that C significantly contributes to the run time of the inference task. As such, the communication overhead and the parallelization factor impact the overall run time in opposite ways, as most edge devices are bandwidth-constrained. Therefore, the use of a larger number of devices generally leads to a lower memory footprint per device F_n in exchange for higher communication overheads C.

5.3.1 Baseline

The trivial baseline of distributed inference is the execution of the entire DNN on a single device without any cooperation. Considering a DNN with L layers mapped to one device,

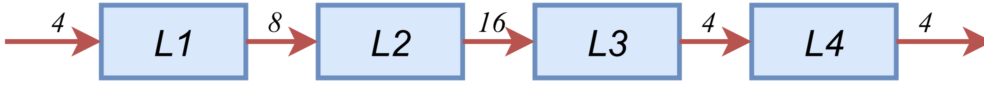


Figure 5.2: 4-Layer example on a single device.

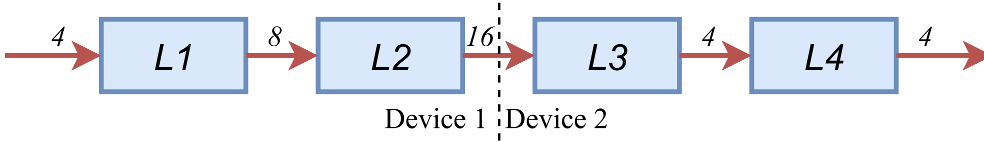


Figure 5.3: 4-Layer example with sequential layer mapping.

the following metrics can be calculated.

$$F_n^{(N)} = \sum_{l=1}^{L} Q_l, \quad T^{(N)} = \sum_{l=1}^{L} R_l, \quad C^{(N)} = 0 \tag{5.4}$$

Running Example:

To illustrate the partitioning schemes more clearly, a simple example DNN is introduced. Figure 5.2 shows this example DNN consisting of four fully-connected layers. Each layer is represented by a box labeled L1 to L4, and each edge represents the feature data between layers and is labeled with the feature data size. In this simple example, the relevant metrics can be easily read from the figure: $F_1^{(N)} = 4 \cdot 8 + 8 \cdot 16 + 16 \cdot 4 + 4 \cdot 4 = 240$, $T^{(N)} = 240$, $C^{(N)} = 0$.

5.3.2 Pipelining

Layer pipelining as in [65] can be applied to the same example to distribute layers between two devices, reducing F_n. Layers 1 and 2 can be mapped to Device 1 and Layers 3 and 4 to Device 2, as shown in Figure 5.3. Intuitively, the new metrics are $F_1^{(DL)} = 4 \cdot 8 + 8 \cdot 16 = 160$, $F_2^{(DL)} = 16 \cdot 4 + 4 \cdot 4 = 80$, $T^{(DL)} = 160 + 80 = 240$ and $C^{(DL)} = 16$. The two devices have different memory footprints because there is no way to evenly distribute the memory between the two devices. When determining the required memory for each device, the maximum partition size must be used, in this case $F_1^{(DL)}$. Moreover, this method does not utilize any parallelism for a single input, leading to the same high run time as for running on a single device, now with additional communication overhead. This mapping can be improved by using layer partitioning, which will be the focus of the remainder of this section.

5.3.3 Feature Partitioning with FFMT

Depending on the size of the features and weights, the layer data should be partitioned by either features or weights to achieve a reduced memory footprint per device. Partitioning the features of DNN layers is useful when their size dominates a layer's memory usage. In typical DNN model structures, this is the case for early layers, because their feature resolution is large and there are fewer convolutional filters. To support the holistic partitioning of DNNs, feature partitioning must also be supported. For this purpose, the state-of-the-art method presented in [106] is included in the solution presented in this thesis. In that work, an approach called *Fused Tile Partitioning (FTP)* is presented that first partitions the input and output feature maps of multiple layers into sets of tiles in an NxN grid. Then, the corresponding tiles are fused across layers to exploit the inherent locality of convolutions. This results in partitions with a set of NxN fused tile stacks that can be executed independently. Since the computation of each consecutive layer depends only on the respective tile of the previous layer, this connection (or fusion) does not require synchronization between devices. The method is equivalent to FFMT, which was detailed in Section 3.2. The method achieves a reduced memory footprint from divided feature maps and reduced communication overhead because synchronization is not required. FFMT is only applicable to convolutional layers and not fully-connected ones, because it exploits the structure of the convolution operation. Due to the nature of fully-connected layers, they are, however, always weight-dominated.

5.3.4 Weight Partitioning with FDT

The core mechanism of the newly presented approach is the application of partitioning to weight-dominated layers of a DNN such that weight data and computational load are evenly distributed across all available devices, whilst minimizing the inference run time. In the following, the presented partitioning schemes are first discussed with respect to fully-connected layers. The translation to convolutional layers will be presented afterward. A weight partitioning scheme can be achieved by partitioning weights so that either the inputs or outputs of a layer are split. Each partition is then mapped to a respective device that only needs to store the weights required to compute its share of input or output data. Weight partitioning by splitting input and output data is simple and very effective when distributing weight data whilst minimizing communication overhead. Finally, a core idea presented in this thesis is the fusion of two consecutive weight-partitioned layers. By first splitting the outputs and then the inputs in the following layer, the need for communication between these two layers can be eliminated entirely. This method is equivalent to FDT that was presented in Section 4.2.

The first layer of FDT, also referred to as the FDT Fan-Out, uses all inputs $\mathbf{a}_l$ and a partition of $\mathbf{W}_l$ to calculate a subset of output neurons $\mathbf{b}_l$. The output value sums are fully complete and can be finalized by applying the activation function. The partitioned output values must then only be concatenated to obtain the full output vector $\mathbf{b}_l$, thus synchronizing the output of the layer across all devices. All partitions of the FDT Fan-Out can be executed in parallel by different devices.

Assuming that FDT Fan-Out is used to all L layers, given N devices, the memory footprint $F_n^{(FDTO)}$, the execution time $T^{(FDTO)}$ and the communication demand $C^{(FDTO)}$

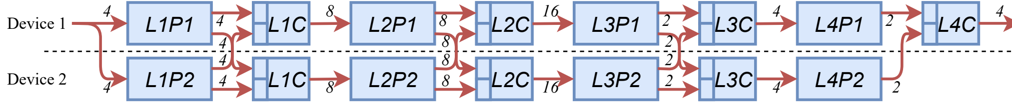


Figure 5.4: 4-Layer example partitioned with FDT Fan-Out.

are given as follows.

$$F_n^{(FDTO)} = \sum_{l=1}^{L} Q_l \cdot \frac{1}{N} \tag{5.5}$$

$$T^{(FDTO)} = \sum_{l=1}^{L} R_l \cdot \frac{1}{N} \tag{5.6}$$

$$C^{(FDTO)} = \sum_{l=1}^{L} C_l^{(FDTO)} \tag{5.7}$$

Independent of the remaining network structure, for any layer using FDT Fan-Out, its communication demand can be calculated as follows.

$$C_l^{(FDTO)} = (1 - r_{l-1}) M_l (N-1) + r_l K_l (N-1) + (1 - r_l) K_l \tfrac{N-1}{N} \tag{5.8}$$

The Boolean variable r_l is 1 if data reuse between layers l and $l+1$ is possible. This is the case when an FDT Fan-Out layer is followed by another FDT Fan-Out layer. This partial communication intuitively means that part of the output is already ready in memory for the next layer. Thus, part of the output data can be reused without the need for additional synchronization. The first summand in Eq. (5.8) counts the input distribution to other $(N-1)$ devices, only if there is data reuse between the current and previous layer. The second and third summands in Eq. (5.8) describe the output distribution for the concatenation operations. If there is no reuse between the current and next layer, the partial data only need to be sent back to the initiating device for concatenation. Otherwise, the partial data need to be shared with all other devices so that they can be concatenated locally.

Running Example:

Figure 5.4 shows the running example DNN, but now partitioned with FDT Fan-Out at every layer. The $LXPY$ blocks represent the partition Y of the layer X and LXC the concatenation operation for the layer X. The first summand in Eq. (5.8) only appears as the leftmost arrow that crosses to Device 2 with the label 4. At the input of all other fully-connected blocks, the data are already available from the previous concatenation and do not need to be synchronized. The second summand in Eq. (5.8) appears as the crossing arrows after every $LXPY$ block but the last. Finally, the third summand appears as the rightmost arrow that crosses to Device 1 with the label 2 and represents the gathering of data back at the initiating device. At this point, the other devices no longer need an own copy of the data. In this given partitioning

configuration, the metrics are obtained as: $F_1^{(FDTO)} = F_2^{(FDTO)} = T^{(FDTO)} = \frac{240}{2} = 120$ and $C^{(FDTO)} = 4{+}4{+}4{+}8{+}8{+}2{+}2{+}2 = 34$. The communication demand can be obtained by adding all arrows that cross the device boundary. The figure also illustrates how the data of previous operations is reused with this method. The memory footprint is evenly balanced across all devices, allowing for full utilization of parallelism, whilst requiring some communication between devices.

Another way of applying FDT is to use its second component, the Fan-In and merge operation. FDT Fan-In takes a partitioned subset of $\mathbf{a}_l$ to calculate the respective partial output of $\mathbf{b}_l$. Although the subset of weights is now different from FDT Fan-Out, the number of weights required per device remains unchanged. As the output values are only an incomplete part of the complete output nodes, they must be summed before being passed to the activation function. As such, the activation function cannot be executed within the partition, and the merge operation is required that sums the output values before applying the activation function of the layer.

If FDT Fan-In were used on all layers for all N devices, the same memory footprint, $F_n^{(FDTI)} = F_n^{(FDTO)}$, would be obtained. The same execution time, $T^{(FDTI)} = T^{(FDTO)}$, is obtained as when applying FDT Fan-Out. The respective communication demand using FDT Fan-In is:

$$C^{(FDTI)} = \sum_{l=1}^{L} C_l^{(FDTI)} \qquad C_l^{(FDTI)} = M_l \cdot \frac{N-1}{N} + K_l \cdot (N-1) \tag{5.9}$$

The first summand refers to the input data for every partitioned operation, and the second summand refers to the output data that must be transferred for the merge operation. A data reuse scheme would not be useful in this case, because the amount of reused data is the same as the data that needs to be exchanged for each redundant merge operation. This results in large communication demand when only FDT Fan-In is used.

Running Example:

Applying only FDT Fan-In to the running example would result in a communication overhead of $C^{(FDTI)} = 48$. However, when the number of inputs to a layer is significantly larger than the number of outputs, applying FDT Fan-In to that particular layer selectively can still reduce the overall communication demand over other layer partitioning types, such as FDT Fan-Out.

As the concatenation and merge operations require input from all partitioned data subsets, they are considered synchronization operations. Synchronization operations carry a large communication overhead. As such, optimizations to the overall execution time can be made by performing more optimal synchronization placement. When combining both presented FDT components, the layers become fused in such a way that one such synchronization point is eliminated. The sequence is then equivalent to the FDT partitioning presented in Section 4.2. The intermediate tensor data between both layers remain local to the devices and do not contribute to any communication demands. It is also important to note that both partial and full FDT operations are applicable to convolutional layers as

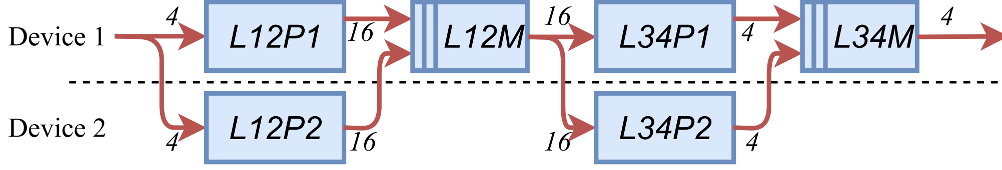


Figure 5.5: 4-Layer example with FDT.

well. This was also detailed in the previous chapter in Section 4.2. The methods are also compatible with sparse weight data, since for those, only the non-zero weight data have to be distributed evenly.

By applying FDT Fan-Out to the first operation of the two fused operations, the interim data now have the appropriate shape to be passed as the input to an FDT Fan-In operation applied to the following DNN layer. The resulting output of the combined operation is equivalent to the output of the FDT Fan-In operation in isolation. Therefore, synchronization is again inevitable at this point because the output only represents partial values of the output. These partial output values must also be merged for summation and activation. This implies that only two consecutive layers can be fused using this method before synchronization is required. For operations that have been fused with FDT, the memory footprint and execution time are the same as for the non-fused FDT components.

$$F_n^{(FDT)} = F_n^{(FDTO)} = F_n^{(FDTI)}, \quad T^{(FDT)} = T^{(FDTO)} \tag{5.10}$$

Although the execution time does not increase, the overall run time of using fused operations, similarly to using the individual FDT components, increases due to the incurred communication overhead. Even though a fusion on two layers could be described as a single new layer, they will still be described in terms of the two original layer indices, to more easily compare different partitioning configurations. The communication demands C for the first and second parts of the fused layers are described below.

$$C_l^{(FDT1)} = (1 - r_{l-1})M_l(N-1), \quad C_l^{(FDT2)} = K_l(N-1) \tag{5.11}$$

The communication demand of the first fused layer is equivalent to that of the FDT Fan-Out without the terms for the outputs. The communication demand of the second fused layer is equivalent to that of the FDT Fan-In without the term for the inputs. These are the intermediate tensor values that no longer need to be communicated because the operations are fused. When multiple devices are executing their fused partition, they work in parallel on their share of the first and second part of the fused layers. Note that if data reuse is possible between the first fused layer and its preceding layer, there is no communication required for the first fused layer at all. Given these benefits of FDT, it might be tempting to apply it to every layer of a DNN. However, this might not be possible, for example, on DNNs that have odd layer counts or when intermittent layers are present that do not qualify for fusing. In this case, the remaining layers can still be partitioned with FDT Fan-Out or Fan-In to achieve fully distributed inference.

RUNNING EXAMPLE:

When applying FDT twice to the running example, as shown in Figure 5.5, the communication demand can be observed as $C = 4 + 16 + 16 + 4 = 40$. The memory demand F and execution time T remain unchanged compared to the individual FDT components. The observed result is worse than applying FDT Fan-Out to every layer. This is because two fusions are not a good solution for the given example. This demonstrates that fusion decisions need to be taken judiciously in order to achieve improvements in distributed inference, as will be further discussed in the following section.

5.4 Optimized DNN Partitioning

There exist several degrees of freedom to partition each DNN for fully distributed inference. In this work, these degrees of freedom will be explored with ILP. A two-step process is proposed involving two ILPs to find optimized solutions. The first ILP optimizes the memory footprint per device by deciding when to use feature partitioning versus weight partitioning. The second ILP optimizes the communication demand in the weight-partitioned DNN layers by selecting the method of weight partitioning to use from the selection of methods presented in the previous section. The solutions to these two optimization problems are detailed in the following.

5.4.1 ILP-based Memory Footprint Minimization

For standard DNNs, the earlier layers are feature-dominated and should run on FFMT as presented in [106], while the latter layers are usually weight-dominated layers, which should use weight partitioning. To reach an optimal memory footprint per device, the following ILP optimization is proposed to identify the point at which to switch from FFMT to weight partitioning.

$$
\begin{aligned}
min_{\mathbf{a},\mathbf{b},c} \quad & F_n^{(FULL)} = c + \sum_{l=1}^{L} (b_l \frac{Q_l}{N} + (1 - b_l) Q_l) && (5.12) \\
s.t. \quad & \forall_{l=1...L} \quad a_l = b_l(M_l + K_l) + (1 - b_l)\frac{M_l + K_l}{N} && (5.13) \\
& \forall_{l=2...L} \quad b_l \geq b_{l-1} \quad b_l \in \{0, 1\} && (5.14) \\
& \forall_{l=1...L} \quad c \geq a_l \quad c, a_l \in \mathbb{N}, && (5.15)
\end{aligned}
$$

Where a_l, described by Eq. (5.13), are integer variables that hold the memory footprint attributed to the inputs and outputs of layer l. b_l are Boolean variables that are true when layer l uses weight partitioning and false if it uses feature partitioning. In weight-partitioned layers ($b_l = 1$), all input/output data must be duplicated on each device, while in feature-partitioned layers ($b_l = 0$) it is partitioned by the number of devices. This formulation includes some small simplifications, since FTP has a "slightly larger (3%)" [106] memory footprint due to overlapped data. Moreover, LOP or LIP layers do not require

the full input or output. The numbers presented in the experimental results section will represent the actual methods without these simplifications, which are only present in this ILP formulation. Eq. (5.14) ensures that the layer partitioning can only be changed once from feature partitioning to weight partitioning. c is an integer variable that represents the maximum of a_l, because only the highest input/output pair counts toward the total memory footprint. Since the objective function is minimized, it is sufficient to specify the constraint in Eq. (5.15) to achieve the desired equivalency $c = max_l(a_l)$. The total memory footprint per device is described in Eq. (5.12) as the the sum of the maximum input/output data footprint and the sum of the weight data footprint of all layers. In weight-partitioned layers ($b_l = 1$), the weight data is divided by the number of devices, while in feature-partitioned layers ($b_l = 0$) all weights must be duplicated on each device. By solving this ILP, the point at which to switch from feature-partitioning to weight-partitioning can be extracted as the first b_l that is 1.

5.4.2 ILP-based Communication Optimization for Weight Partitioned Layers

In the latter weight-dominated layers, fusing at every possible opportunity, as done in Figure 5.5, may possibly be an inferior solution to the careful selection between FDT Fan-Out, Fan-In and full FDT fusing two layers. This is because communication depends on the output and input layer sizes, which can vary greatly between layers, and since layers can only be fused pairwise. Fusion should be performed in such a way that communication sizes between fused layer pairs are minimized. If a layer's output or input is fused, its input or output can, in turn, no longer be fused and must be communicated. Additionally, a non-fused layer may favor either FDT Fan-Out or FDT Fan-In partitioning schemes. The solution to this optimization problem is called Optimized Weight Partitioning (OWP). Given a network of L layers, $o_l = 1$ is defined if layer l uses FDT Fan-Out, similarly $i_l = 1$ if it uses FDT Fan-In, $f_l = 1$ for the first part of a fused layer and $s_l = 1$ for the second part of a fused layer. Otherwise, all variables are zero. With this we can formulate the communication demand C for the OWP as follows.

$$C^{(OWP)} = \sum_{l=1}^{L} \left(o_l C_l^{(FDTO)} + i_l C_l^{(FDTI)} + f_l C_l^{(FDT1)} + s_l C_l^{(FDT2)} \right) \tag{5.16}$$

The proposition is that partitioning and fusion decisions should be made so that the communication demand C is minimized. This is achieved with the following ILP optimization formulation.

$$\begin{aligned}
min_{\mathbf{o},\mathbf{i},\mathbf{f},\mathbf{s},\mathbf{r}} \quad & C^{(OWP)} && (5.17)\\
s.t. \quad & \forall_{l=1\ldots L} \quad o_l + i_l + f_l + s_l = 1 \quad o_l, i_l, f_l, s_l \in \{0,1\} && (5.18)\\
& \forall_{l=1\ldots L-1} \quad s_{l+1} = f_l && (5.19)\\
& \forall_{l=1\ldots L-1} \quad r_l = o_l(o_{l+1} + f_{l+1}) \quad r_l \in \{0,1\} && (5.20)\\
& f_L = 0 \qquad s_1 = 0 \qquad r_L = 0 && (5.21)
\end{aligned}$$

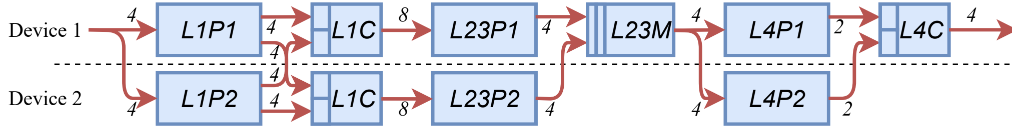


Figure 5.6: 4-Layer example with Optimized Weight Partitioning.

Here, Eq. (5.18) assures that only one partitioning scheme is chosen per layer, Eq. (5.19) assures that the second fused layer is performed directly after the first fused layer and Eqs. (5.21) prohibit illegal fusing pairs at the boundaries and disallow data reuse from the last layer. Eq. (5.20) defines that data reuse occurs when the current layer is FDT Fan-Out and the subsequent layer is either FDT Fan-Out or a first fused layer. Eq. (5.20) and some summands of Eq. (5.16) contain products of variables, but since they are binary variables, they can be linearized with a helper variable and additional constraints as follows [16].

$$c = ab \quad a, b, c \in \{0, 1\} \tag{5.22}$$

$$c \leq a \qquad c \leq b \qquad c \geq a + b - 1 \tag{5.23}$$

Running Example:

Optimizing partitioning and fusion decisions for our running example leads to the optimized solution shown in Figure 5.6. Layers 1 and 4 both employ FDT Fan-Out, while layers 2 and 3 are fused with FDT. The calculated communication overhead of $C^{(OWP)} = 22$ shows a significant reduction given the partitioning and fusion optimizations. The previously most significant communication demand was present between layers 2 and 3, as seen in Figure 5.5. This demand could be eliminated through the fusion decisions seen in Figure 5.6. In summary, the presented ILP optimization considers the input and output sizes of all layers to decide on the best partitioning scheme of each layer.

5.5 Experimental Results

The presented methods are evaluated using the widely known CNN models YOLOv2 [80], AlexNet [51], VGG-16 [88] and a GoogLeNet derivative (called "Extraction") [96]. Pre-trained models were taken from [79], available under the names `YOLOv2 608x608`, `AlexNet`, `VGG-16` and `Extraction`, respectively. YOLOv2 consists of only convolutional layers, making it a fully convolutional network [62]. The optimization methods presented in this chapter as Eq. (5.12) and Eq. (5.17) are applied to these models to find their optimal partitioning configuration. Then, the models are deployed on a Raspberry Pi 4 edge cluster for measurements of run time and memory usage. The ILPs were implemented using the "Coin-or branch and cut" ILP solver included in OR-Tools [31, 76]. On an Intel Core i5-7500 machine running at 3.4 GHz, all ILP evaluations completed in less than a second, posing no limitation to the practicality of the distributed inference approach.

Model	L	First OWP layer	$F_n^{(SINGLE)}$ [MB]	$F_n^{(FULL)}$ [MB]	F_n reduction	$F_n^{(SEQ)}$ [MB]
YOLOv2	32	13	256	28.4	9.0x	51.8
AlexNet	14	3	16.8	2.65	6.3x	5.82
VGG-16	25	8	84.5	13.2	6.4x	25.8
GoogLeNet	27	5	97.5	12.2	8.0x	20.1

Table 5.1: Memory footprint reduction for ten devices.

Model	$C^{(FDTO)}$ [MB]	$C^{(OWP)}$ [MB]	$C^{(OWP)}$ Saving [%]
YOLOv2	84.0	59.8	28.8
AlexNet	9.69	9.11	5.96
VGG-16	202	182	10.1
Extraction GoogLeNet	47.0	41.8	11.1

Table 5.2: Communication savings of OWP over LOP for six devices.

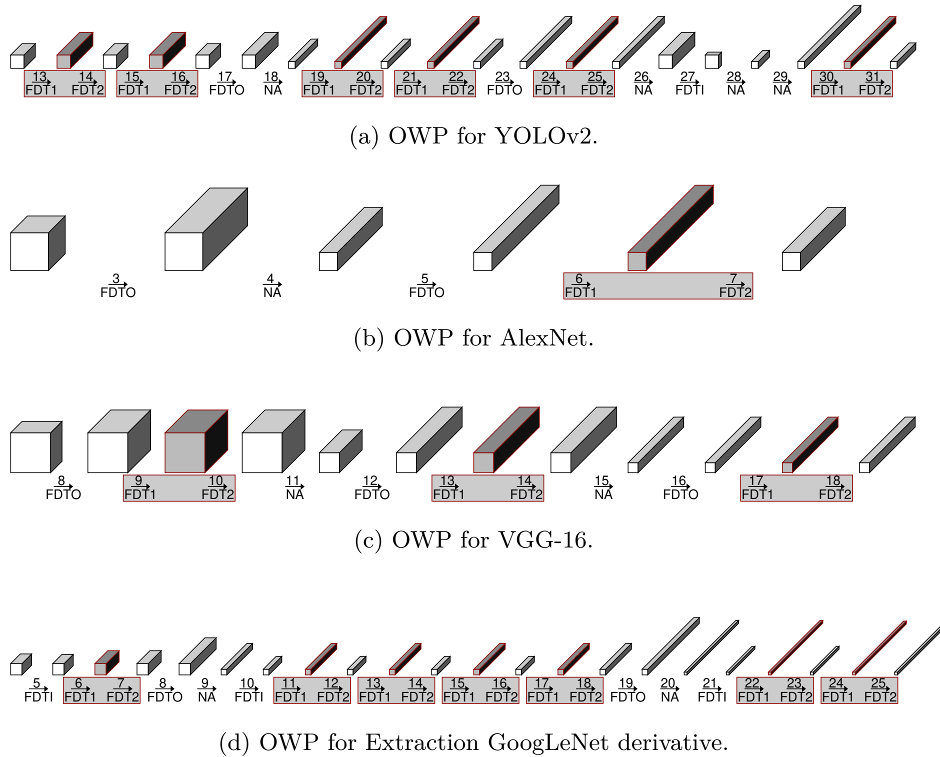
5.5.1 ILP-based Memory Footprint Minimization

Table 5.1 shows the different models with their number of layers L and the optimized layer at which to switch from feature to weight partitioning. It lists the memory footprint per device $F_n^{(FULL)}$ using a cluster of ten devices compared to a single device that performs the inference with footprint $F_n^{(SINGLE)}$. The layer that was manually selected in [106] to switch from feature-intensive partitioning to weight-intensive partitioning was layer 17 for the YOLOv2 model. The choice was made manually on the basis of analysis of the memory usage per layer. When evaluating Eq. (5.12), this results in $F_n^{(FULL)} = 37.9$ MB which is 33% higher (equivalent to a 25% reduction) than the automatically optimized solution that switches at layer 13 to achieve a memory footprint of 28.4 MB. Using sequential layer mapping, the memory footprint per device can only be reduced to the largest memory requirement of a single layer, which is the best-case scenario. Additional layers may have to be assigned to the device with the largest layer to limit communication demand. For the evaluated models, this memory demand is shown in the last column as $F_n^{(SEQ)}$. Compared to sequential layer mapping, with the proposed OWP method the number of devices to reduce the memory footprint per device can be increased arbitrarily.

5.5.2 ILP-based Communication Optimization

The ILP for finding the Optimized Weight Partitioning (OWP) is evaluated on the four introduced models in a system architecture consisting of six devices. The solutions for evaluating the ILP on the different models are shown in Figure 5.7. The figures represent the weight-partitioned part of the DNNs using the decisions from the previous section. The volumes represent the feature tensors at each layer, and their sizes are proportional to the tensor sizes. The arrows between the volumes represent the layer operations with their index indicated above the arrow and the ILP solution for the weight partitioning type indicated below the arrow. If a layer is not convolutional, it is labeled `NA`. Whenever a volume is located between the `FDT1` and `FDT2` types, it does not contribute to the total communication demand because neurons do not have to be communicated between the fused layers.

Table 5.2 contains the different models and compares the communication demand resulting from the application of FDT Fan-Out at every layer ($C^{(FDTO)}$) with the demand of



(a) OWP for YOLOv2.

(b) OWP for AlexNet.

(c) OWP for VGG-16.

(d) OWP for Extraction GoogLeNet derivative.

Figure 5.7: Optimized Weight Partitioning for different CNNs.

OWP ($C^{(OWP)}$). Savings of approximately 6% to 29% indicate that the ILP can provide good decisions for the selection of weight partitions for different DNN models. Note that these solutions are guaranteed to be optimal in terms of the communication demand as described by the ILP.

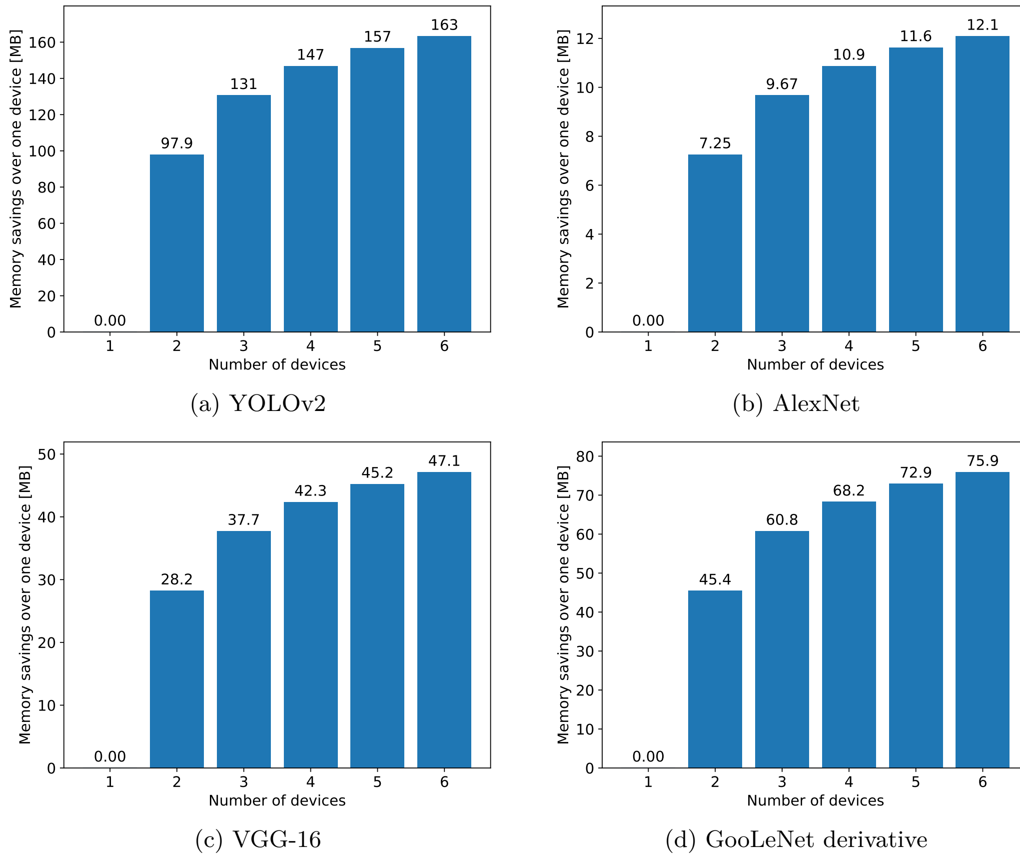
5.5.3 Evaluation on Raspberry Pi Edge Cluster

The complete approach is validated on a physical edge cluster consisting of six Raspberry Pi 4 devices. They have a quad-core ARM Cortex-A72 processor, 2 GB of RAM and a gigabit Ethernet interface. Their operating system is *Raspberry Pi OS (32-bit) 2020-02-13* that includes the Linux kernel *4.19.97*. The software framework for fully distributed inference was implemented on top of the framework from [106], which incorporates the FFMT partitioning for the earlier layers. The major addition was the extension of the presented OWP method that employs FDT partitioning for the latter layers. Both the original and the presented framework are based on the DarkNet deep learning framework [78] with an added patch for use of the NNPACK acceleration library tuned for the ARM Neon SIMD instruction set extension. Along with the compiler optimization flag `-Ofast` this ensures competitive

inference performance. Edge devices fulfill two different roles during the inference process. Either they provide the input data as the source device or they assist with the inference as worker devices. This differs from the work in [106], where all processing of the weight-dominated latter layers of the DNN is delegated to a single powerful central gateway device. The framework was adapted so that the source device collects the intermediate layer results for synchronization before distributing them to the worker devices. The described data reuse mechanism is also implemented and does not require full synchronization via the source device. The central gateway device is kept in the implemented framework for device discovery and coordination, but there is no fundamental technical limitation that would prevent such a network from running purely on peer-to-peer technology. The prior existing communication pattern did not require long-running connections between devices for back-and-forth communication. However, with fusion of weight-partitioned layers, output data has to be synchronized at least every two layers between worker devices. As a result, a large number of messages must be exchanged between the source and worker devices. For the overall run time, it is therefore essential that the connection between the source device and all worker devices remain open. This has been enabled in the employed extended framework. The two partial convolution operations that implement FDT Fan-Out and Fan-In were implemented with the same linear algebra function as the baseline convolution. This ensures performance equivalent to untouched convolutions. To implement layer fusion, these operations were combined into a single FDT operation, for which all communication and memory copies were stripped. These steps extended the existing framework to enable fully distributed deep learning inference with support for layer fusion.

The run time is measured from the start of the inference until the final inference result has been calculated. The cluster consisted of six devices connected by a gigabit network switch. A seventh Raspberry Pi device acts as the gateway device, responsible for the coordination of the network. For each configuration, ten measurements were taken and the results were averaged to take run time variability into account. Memory measurements had a negligible variation of one or two pages (4-8 kB), so only one measurement is reported. The Linux tool `tc` was used to impose software-based bandwidth throttles on the device's Ethernet interfaces, thus allowing simulation of bandwidth reductions. By being able to control the bandwidth linking the devices, it was possible to magnify the communication overhead effects on inference partitioning, as will be seen when dealing with low bandwidths, such as 10 Mbit/s. Peak memory usage was measured with the Linux `/proc/pid/status` file, which includes a value `VmPeak` to measure "Peak virtual memory size".

Figure 5.8 shows measurements of the peak memory usage savings when compared to a single device. The measurements were performed for both an OWP partitioning and a partitioning that only uses FDT Fan-Out. Since the underlying DarkNet library is not optimized for memory usage, absolute memory usage is not reported. An approximate memory baseline can be taken from $F_n^{(SINGLE)}$ in Table 5.1. As mentioned in Section 2.3, current state-of-the-art edge inference frameworks such as TFLM achieve memory overheads below 50 kB which are negligible compared to the reported memory sizes. Peak memory values are measured just before inference because inefficient use of runtime queues skewed peak usage after inference. Weights are first loaded in their entirety and only pruned from memory in a second step. This would falsify the measurement because the peak would reflect the entire weights without the pruning. Since the absolute memory values are not of interest, this is simply fixed by loading a large dummy buffer after pruning. When the



(a) YOLOv2

(b) AlexNet

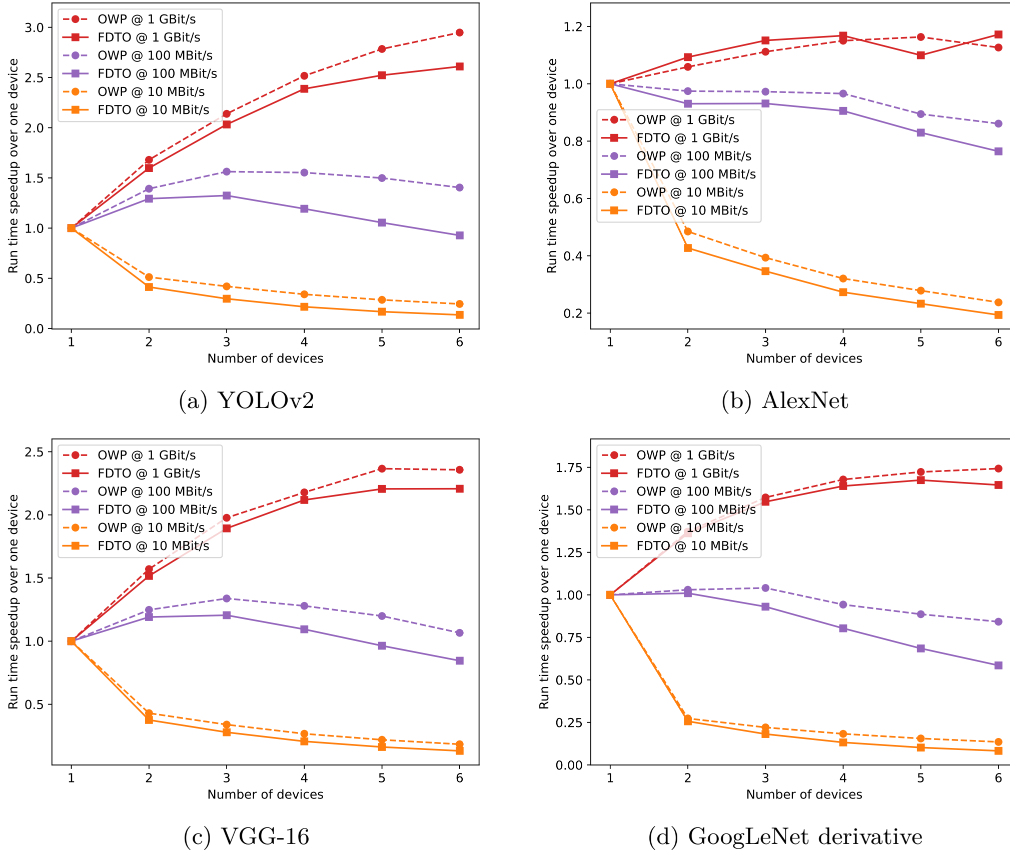
(c) VGG-16

(d) GooLeNet derivative

Figure 5.8: Memory savings results.

number of devices increases, an expected proportional decrease in the memory footprint per device according to Eq. (5.12) is observed. This confirms that balanced inference scaling is achieved across the available memory resources of the edge devices. Furthermore, OWP does not show any additional overhead when compared to the simple FDT Fan-Out partitioning.

Figure 5.9 shows the total inference run time speedup over a single device for different edge device and network parameters on the Raspberry Pi cluster. The OWP presented in this chapter is compared to a simple partitioning that uses FDT Fan-Out (FDTO) for every layer. The absolute baseline run time values for one device are given in Table 5.3. The results from the previous section were applied to decide at which point to switch from feature to weight partitioning and for finding the OWP. Note that the switching point was optimized for ten devices and kept the same for one to six devices, because this will keep the share of feature- and weight-partitioned layers constant to focus on the scaling of multiple devices. These results have previously been collected during the course of this thesis in a simulation setup [92]. Moving the experiment to real hardware reduced the variance of the results. This is likely because in the simulation setup, all applications compete for the



(a) YOLOv2

(b) AlexNet

(c) VGG-16

(d) GoogLeNet derivative

Figure 5.9: Run time speedup of FDTO vs. OWP.

same Linux scheduler and virtualized network, causing more congestion and interference with each other. The simulation setup also used a software mechanism to throttle the CPU utilization per simulated device, adding more variance to the results. Since existing work does not partition weight-dominated convolutional layers, the only meaningful comparison is to a baseline that executes on a single device. Assuming that the worker nodes are idle, a run time speed-up was observed for the inference task. Note that for AlexNet, the 1 Gbit/s results show an overall slowdown (except at 5 devices) when using OWP over FDTO. This could be explained by measurement inaccuracies because AlexNet has the lowest theoretical saving, as shown in 5.2, and the high bandwidth makes the communication savings even less significant. There are several factors that influence the final run time of the inference task compared to the theoretical values for T and C given varying numbers of edge devices. First, the effects of communication latency must be considered for each message between devices. Second, communication and parallelism have opposing impacts on run time. As such, when increasing the cluster size from three devices to six devices, no clear run time trend was observed. When comparing OWP layer fusion with standalone application of

Run time [s] / Model	1 Gbit/s		100 Mbit/s		10 Mbit/s	
	FDTO	OWP	FDTO	OWP	FDTO	OWP
YOLOv2	13.1 ± 0.23	13.0 ± 0.08	13.1 ± 0.06	13.1 ± 0.05	13.9 ± 0.15	13.9 ± 0.09
AlexNet	1.00 ± 0.018	0.98 ± 0.021	0.98 ± 0.006	0.99 ± 0.004	0.96 ± 0.045	0.99 ± 0.046
VGG-16	6.59 ± 0.022	6.64 ± 0.021	6.68 ± 0.035	6.63 ± 0.024	7.14 ± 0.052	7.11 ± 0.055
GoogLeNet	2.21 ± 0.010	2.21 ± 0.009	2.23 ± 0.015	2.22 ± 0.022	2.23 ± 0.028	2.21 ± 0.035

Table 5.3: Baseline run time values and standard deviation for one device.

FDT Fan-Out, a clear benefit can be seen in terms of run time speed-up. The benefit is less significant with very high available bandwidth (1 Gbit/s) as the fusing only improves communication demand, which does not act as a bottleneck given such network speeds. The speed-up of OWP is generally higher as more devices are involved, because there is more communication happening that can be positively affected by layer fusion. However, as the number of devices increases, the total run time may also increase again, depending on bandwidth. Nevertheless, a significant speed-up was observed when layer fusion was used with FDT and OWP, which does not impose additional run time or memory costs.

5.6 Summary

This chapter presented contributions for the optimization of memory and communication demand in distributed DNN inference. The typical structure of DNNs demands a careful selection between the fused tiling methods FFMT and FDT. Feature-dominated operations benefit from FFMT, while weight-dominated operations can only be distributed with FDT. An ILP formulation has been presented to determine the optimal operation at which to switch from FFMT to FDT. Another ILP was presented that optimizes the communication demand by choosing where and how to fuse weight partitioned layers.

Chapter 6

Hardware/Software Interface Generation and Optimization

Significant shares of the MCU product cost is spent on software development that needs to consider the limited memory as well as computational resources. This chapter investigates the interface between hardware and software in detail and provides solutions for its definition, optimization and automatic code generation.

6.1 Motivation

Application developers face the challenge to implement some functionality on top of device drivers with highly limited resources in terms of design effort, available on-chip memory and computation power. This application functionality should be able to make use of most of these system resources while keeping the resources used by the drivers as low as possible. Yet, this driver code can make up a significant portion of an MCUs limited memory. For example, the RISC-V PULPino MCU has 32 KiB of instruction memory, of which all its driver code already occupies 6% [101]. In a minimal application, it occupies the majority of the entire code size, as shown in Figure 6.1.

As introduced in Section 2.1, the smallest logical unit of a peripheral interface is a *bit field*. They can range from 1 bit size up to spanning multiple registers. Hardware design typically dictates how all bit fields are mapped to a concrete register layout. This prevents potential for optimization on the software side in terms of code size, run time and the number of necessary accesses. Due to the tightly constrained resources of MCUs, such an optimization of the interface is desirable. A second, closely related aspect is the cost of soft-

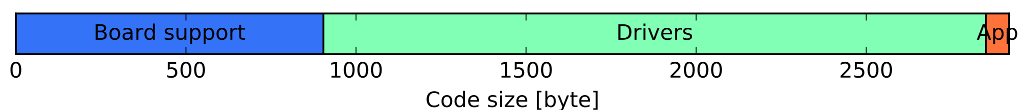


Figure 6.1: PULPino code size distribution.

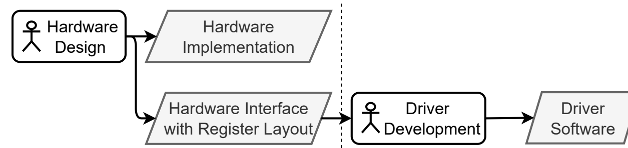


Figure 6.2: Traditional development flow.

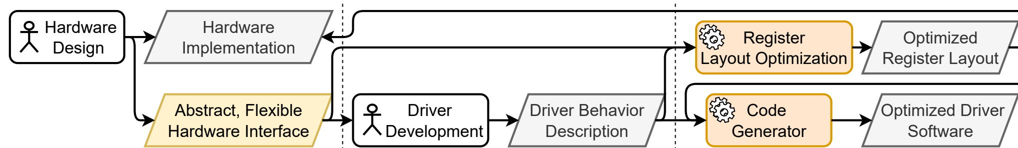


Figure 6.3: Proposed development flow.

ware development, of which large parts can be attributed to optimization and maintenance of low-level code. Of course, this also involves the driver code that implements the interface between hardware and software. The challenge is to write driver behavior while simultaneously considering the predetermined register layout to produce well-performing code with low memory footprint. Low-level primitives, such as macros and bit manipulation operations, pollute the source code, decreasing readability and maintainability. Although HAL functions can be introduced to separate driver behavior from such low-level bit field accesses, they have the disadvantage that more high-level programming concepts are incompatible with their fixed interface. Most importantly, this affects the concept of type hierarchies. When defining an interface for sub-parts of a peripheral, it is not possible to reuse this partial interface in other places with a traditional HAL. This becomes even more difficult with the concept of arrays in the interface. Driver software is unable to access arrays dynamically if all array elements have different HAL function names. Together, these restrictions can lead to inferior performance and memory footprint.

Driver code for MCUs is predominantly written in the C programming language. C specifies a programming model where all statements defined in the language may be changed by a compiler, as long as the final result is equivalent to an abstract machine calculating the statements. A convoluted source code function with many statements that for some reason always returns the integer 42 can just be replaced by a single machine instruction that places 42 in the result register. While this permits most of the powerful optimization methods implemented by modern compilers, it is an issue when interacting with entities outside of the abstract machine. This is the case for memory-mapped registers, because they may exhibit behavior other than simply storing and retrieving memory values. The C language defines the `volatile` type qualifier to let the compiler know that certain memory accesses must not take part in optimization. However, this is a very coarse way of specifying accesses since any and all optimization is prevented.

6.2 Contribution

The traditional flow from hardware design to driver software is shown in Figure 6.2. A fixed register is provided along with the hardware interface to which the driver software must adhere. The flow proposed in this chapter is shown in Figure 6.3. Instead of the fixed interface, an abstract and flexible one is provided. This interface is used to create a driver behavior description on the software side. Finally, an automated flow creates an optimized register layout and generates the optimized driver software. Of course, the optimized register layout must also be reflected in the hardware implementation. Either the actual hardware can be modified by automatically generating hardware description language, or the system bus could be reconfigured to implement different interfaces. How this is done is not discussed in this thesis.

The major enabler for a more optimized interface between hardware and software is the specification format of the bit fields. Compared to most existing approaches with new DSLs, this thesis approaches the same issue at a higher level. The concept of a flat collection of bit fields is abandoned in favor of hierarchies and multiplicity to allow higher-level behavior code to be more generic and reusable. Behavior code is simplified and therefore also uses less memory. The proposed solution is an extension for the C programming language, where a flexible hardware/software interface can be defined. By being a non-invasive extension, it is easy to adopt for driver developers familiar with C. This allows developers to focus on implementing the desired behavior without having to consider the performance implications imposed by the low-level interface.

The register layout optimization has the goal of producing a fixed register layout that assists in the generation of optimized driver software. To achieve this, it must first find all side effects and interactions between bit field accesses in the behavior code. Explicit side effect descriptions can be given in the language extension, but some implicit control-flow and data-flow dependencies must be taken into consideration as well. Implicit dependencies are extracted by source code analysis. The optimized mapping from bit fields to register layout is then obtained by a heuristic method that greedily searches for bit field accesses which can be combined into single memory accesses. Combined accesses reduce code size, run time and the number of bus accesses.

Finally, a code generation approach is presented that takes advantage of this optimized register layout. The approach is able to combine accesses to different bit fields to reduce the total number of accesses and systematically makes use of base pointers and array indices to reduce memory usage through code reuse. The final output is optimized C driver code using the optimized register layout.

6.3 C Language Extension

The proposed flow revolves around specifying and using the hardware/software interface with a language extension to the C programming language. The extension covers the declaration of the interface components along with their relations and interactions on one hand and the use of these components in the remaining software, for example in drivers.

6.3.1 Bit Field Group Definition

The core construct of C struct definitions is extended upon, because they enable a concise way to define members of a type. Struct definitions are familiar to C programmers and are flexible enough to allow additional annotations. To distinguish the new custom definitions from standard structs, instead of the keyword `struct`, the keyword `bfGroup`, short for *bit field group* is used. These groups have an important distinction from regular C structs in that the order of their fields is not necessarily laid out in the order of their declarations as mandated by the C Standard [44]. This keeps the order flexible to be transformed into an optimized register layout based on the bit field usage in software. Another extension is required to define the exact length of bit fields within the bit field groups. The available data types in C do not support bit-accurate sizes and the smallest data type is `char` with a size of at least 8-bit. C already has a language feature called *bit fields* to define fields of a struct with bit-accurate sizes, but it neither supports hierarchy nor multiplicity. Instead, the proposed extension adds keywords for new type names from `uint1` (with alias `bit`) up to `uint64` to define the bit-accurate size of fields in bit field groups. A bit field group is defined with the following pattern that exactly mirrors a struct definition.

```
1  bfGroup group_name {
2      type field_name;
3      // ...
4  };
```

The following shows this feature in the definition of a general-purpose input/output (GPIO) IO pad of the PULPino SoC. In particular, the order of bit fields is not fixed such that `in` might be ordered before `dir` for example.

```
1  bfGroup GPIOPad {  // bit field group definition
2      bit dir;  // bit field definition; 1 bit size; order not fixed
3      bit in;
4      bit out;
5      bit intEn;
6      uint2 intType;
7      uint8 cfg;
8  };
```

Besides bit-accurate fields, bit field groups are also allowed to contain fields that have the type of other bit field groups. The composition of bit field groups like this allows a hierarchical representation that enables code reuse. This is very important as register layouts of peripherals often have a hierarchical structure. As another feature of the language extension, any bit fields and bit field groups can be specified as array types. The following partial definition of the PULPino GPIO peripheral bit field group showcases both hierarchy and array multiplicity. The array feature is used to define that thirty-two IO pads are contained in the top-level bit field group along with another singular bit field.

```
1  bfGroup GPIOType {
2      // GPIOType contains 32 instances of dir, in, out, ... from above
3      bfGroup GPIOPad pads[32];
4      uint32 intStatus;
5  };
```

Once all the bit field groups have been defined for a peripheral device, the device has to be instantiated because the toolchain is not able to guess which of the defined bit field groups represent the device and which are just defined as elements of the type hierarchy. This instantiation is done with `make_device(group_name, global_name, base_addr)`, which is

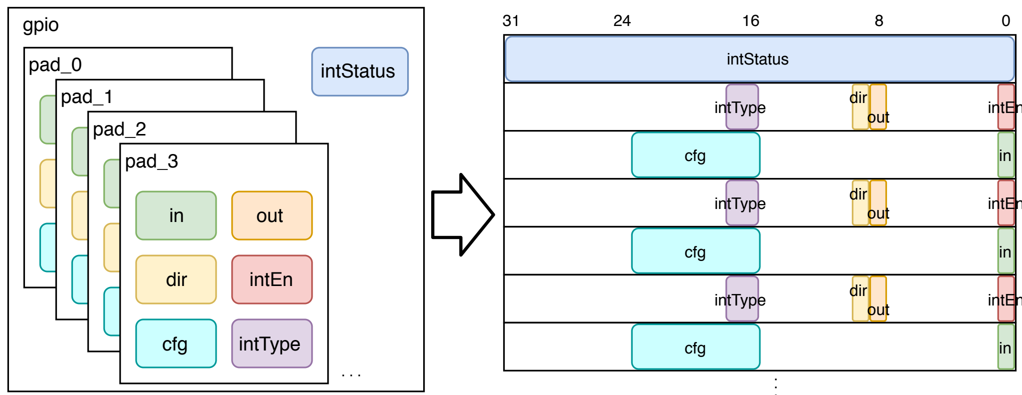


Figure 6.4: Mapping from abstract bit fields to a concrete layout.

equivalent to creating a global pointer of type `group_name` with name `global_name` and initializing it with the base address `base_addr`. The following snippet shows the instantiation of the PULPino GPIO device.

```
1  make_device(GPIOType, gpio, 0x1a101000);
2  // equivalent to:
3  // bfGroup GPIOType *gpio = 0x1a101000;
```

This part of the C language extension allows for a very compact definition of peripheral interfaces and provides sufficient flexibility for convenient usage patterns. The abstract definition of bit fields presented here is free from actual layout information and only later becomes transformed into a mapping to registers. This process is shown on the GPIO example in Figure 6.4.

6.3.2 Hardware Side Effects

Modern compilers find highly optimized representations of the behavior described in source code in terms of run time as well as memory size. The goal of this work is not to improve the compiler itself for marginal gains but instead to tackle a limitation in the source code description that is especially relevant for the hardware/software interface. In code using a peripheral interface, it is necessary to use the `volatile` type qualifier whenever memory-mapped registers are accessed directly [44]. Without `volatile`, the compiler may optimize the low-level accesses and thereby change behavior in an unwanted way. For example, a write access might be completely eliminated if the same address is never read again. Or, if a sequence of write and read accesses occurs to the same address, a simple optimization can remove both memory access instructions in favor of just reusing the to-be-written value. However, bit fields may be modified spontaneously by the peripheral itself, or accesses may have side effects in the peripheral device or any further hardware that interacts with the outside world. For example, the clear-on-read concept would clear a bit field value to zeroes after it has been read, such that any subsequent read access should receive a different value. Such side effects are often described in data sheets or can sometimes even be retrieved from company internal machine-readable representations. The issue with `volatile`

is that it places a very broad constraint on the compiler that prevents most optimization even if some optimization transformations are compatible with the behavior of a particular peripheral. For example, when multiple bit field values are read sequentially from the same register, it might be possible to read them all at once, which is a prohibited optimization for volatile accesses. By starting the flow from hardware design, there is sufficient information and flexibility to describe these register accesses in a more precise way than with volatile qualifiers. This added information can be expressed as bit field properties in the proposed C language extension. The register layout optimization then takes these additional constraints into account, and the code generator can produce more efficient and compact code.

There are multiple ways to provide the required side effect information to a generation framework. It could be explicitly provided in a dedicated file, but this has the disadvantage that closely related information is split into different places. In the approach presented here, the dependencies between bit fields are described in a way that allows the code generator to apply such optimization steps. Dependencies between fields can be expressed as annotations right where they are defined in their bit field group. The keyword `rse` is used for *read side effects* and `wse` for *write side effects*. They mark side effects for reading from or writing to the field and specify which other fields may be influenced by the side effect. The general concept of side effect definitions is that a read or write operation to the annotated bit field will cause a side effect on the specified influenced bit field(s) that invalidate any potentially cached or assumed values. In detail, there are several scenarios for side effects:

1. `type rse(a) a`: Whenever `a` is read, `a` is invalidated. This means that multiple subsequent read operations on the field must not be optimized into a single read operation since the value could have been updated between two reads. An example of this is a counter register. After reading it, it might already have changed to a new value. There is no restriction for sharing registers with fields of this side effect type.

2. `type rse(b) a`: Whenever `a` is read, `b` is invalidated. A read operation on the field must cause a read operation on the affected field if it is requested afterwards. This behavior is displayed, for example, by hardware lock registers. A read from them prepares the data in another memory location for reading. It is undesired to share registers with fields of this side effect type because a read operation from the unrelated shared field will then unnecessarily invalidate the affected field as well, possibly with unwanted side effects as in the case of locking.

3. `type wse(a) a`: Whenever `a` is written to, `a` is invalidated. Additionally, as a special case, it is defined that this definition may cause some side effect outside the abstract state machine. This definition is equivalent to the full constraints of the volatile type qualifier. For example, a register that causes a buffer to be sent on a communication peripheral may not have any visible effect on the device register interface, but has an effect in the outside world. Sharing a register that contains this side effect type with other fields is not possible at all, since it is very likely that it causes wrong behavior.

4. `type wse(b) a`: Whenever `a` is written to, `b` is invalidated. For example, a reset register shows this behavior since it causes other fields to be set to their reset value. Obviously, an unrelated write operation to another field may not cause such an effect, so sharing registers with fields of this side effect type is not possible.

Identifiers to declare the influenced fields, which are given to the `rse` or `wse` annotations can be bit fields in the same bit field group, other bit field groups, or even bit fields in other groups with the notation `group_name::field_name`. The above example of the PULPino GPIO peripheral was incomplete and actually contains a side effect definition for the `intStatus` bit field, because it may spontaneously change to indicate changes in internal status. The full definition is given below.

```
1  bfGroup GPIOType {
2      bfGroup GPIOPad pads[32];
3      uint32 rse(intStatus) intStatus;
4  };
```

6.3.3 Behavior Description

Embedded C driver functions typically dereference a volatile pointer to access bit fields, as shown in the following example of a PULPino GPIO driver function.

```
1  void set_gpio_pin_value(int pinnumber, int value) {
2    volatile int v;
3    v = *(volatile int*) (GPIO_REG_PADOUT);
4    if (value == 0)
5      v &= ~(1 << pinnumber);
6    else
7      v |= 1 << pinnumber;
8    *(volatile int*) (GPIO_REG_PADOUT) = v;
9  }
```

Since this code performs bit field masking directly in the driver implementation instead of using HAL functions, its appearance is overly complex and obfuscates the intended behavior. With the proposed C language extension, it is possible to use the bit field group definitions as if they were standard C struct definitions. The regular struct field accesses will later be transformed into appropriate volatile memory accesses by the code generator. The driver developer can exploit hierarchical bit field declarations and arrays to write high-level code that facilitates reuse and is free from bit manipulation as shown in the following code of the same GPIO driver function, now with the language extensions.

```
1  void set_gpio_pin_value(int pinnumber, int value) {
2    gpio->pads[pinnumber].out = value;
3  }
```

6.3.4 Implementation

In summary, the grammar of the language extension can be formalized as follows.

⟨*hw-param-def*⟩ ::= '`bfGroup`' ⟨*identifier*⟩ '`{`' ⟨*hw-member-list*⟩ '`};`'

⟨*hw-member-list*⟩ ::= ⟨*hw-member-decl*⟩ ⟨*hw-member-list*⟩$_{\text{opt}}$

⟨*hw-member-decl*⟩ ::= ⟨*any-type*⟩ ⟨*side-effect-list*⟩$_{\text{opt}}$ ⟨*identifier*⟩ ⟨*array-decl*⟩$_{\text{opt}}$ '`;`'

⟨*any-type*⟩ ::= ⟨*bit-accurate-type*⟩ | '`bfGroup`'

⟨*bit-accurate-type*⟩ ::= '`bit`' | '`uint`' ⟨*number-1-to-64*⟩

⟨*side-effect-list*⟩ ::= ⟨*side-effect-spec*⟩ '(' ⟨*identifier*⟩ ')' ⟨*side-effect-list*⟩$_{\text{opt}}$

⟨*side-effect-spec*⟩ ::= '`rse`' | '`wse`'

⟨*array-decl*⟩ ::= '`[`' ⟨*int-literal*⟩ '`]`'

⟨*hw-instance*⟩ ::= '`make_device(`' ⟨*identifier*⟩ '`,`' ⟨*identifier*⟩ '`,`' ⟨*int-literal*⟩ '`);`'

The symbols ⟨*identifier*⟩ and ⟨*int-literal*⟩ are defined as they are in the C language, and ⟨*number-1-to-64*⟩ expands to any literal number between one and 64. Behavior code is unchanged from regular C grammar, except for the usage of '`bfGroup`' instead of '`struct`' whenever a bit field group type is used, for example, as a function parameter.

The language extension is implemented by including a support header that defines all new keywords as preprocessor directives. The full content of the support header is given below.

```
#define bfGroup struct
#define bit int
#define uint1 int
// ...
#define uint64 int
#define rse(target)
#define wse(target)
#define make_device(group_name, global_name, base_addr) \
    static bfGroup group_name *const global_name =      \
        (bfGroup group_name *)base_addr
```

The only goal of these macros is to keep any code that uses the language extension valid in standard C. That way, an unmodified standard C compiler can be used to parse and process any code written with the language extensions. Any code that accesses bit fields within the defined bit field groups will be completely removed and replaced with customized volatile pointer accesses. Therefore, it is not an issue that the struct layouts are nonsensical and most keywords have no effect. The following snippet shows the PULPino GPIO code that was introduced so far after it was transformed by the C preprocessor.

```
struct GPIOPad {
    int dir;
    int in;
    int out;
    int intEn;
    int intType;
    int cfg;
};
struct GPIOType {
    struct GPIOPad pads[32];
    int intStatus;
};
static struct GPIOType *const gpio = (struct GPIOType *)0x1a101000;

void set_gpio_pin_value(int pinnumber, int value) {
    gpio->pads[pinnumber].out = value;  // will be replaced
}
```

The language extension toolchain is based on the Clang compiler libraries that did not need to be modified. The first step is to load all source files and to prepend an include statement to the support header. Then the Clang preprocessor is used to extract all macros and attach their meaning to the corresponding source code. After replacing the bit field accesses (detailed in Section 6.5), the code can be compiled as standard C with any other

C toolchain. This flow does not require additional effort to create a new language syntax with an own parser. As a bonus, the entire flow supports C++ as well, because Clang does so, too.

6.4 Heuristic Optimization

From the defined bit field usage behavior, a register layout can be generated that is optimized for this specific usage. A major optimization step of the proposed method is the combination of multiple read or write operations into one single operation. However, this is only possible as long as there is no data-flow, control-flow or side effect dependency between the accesses. The optimization flow first has to determine these dependencies, before optimizing the register layout heuristically.

6.4.1 Control-Data-Flow Analysis

Besides the explicit side effects in the bit field group definitions, the control-flow and data-flow add further dependencies between read and write accesses. Hence, this information is extracted from the behavior code with a Control-Data-Flow Graph (CDFG) [5]. It combines the two widely used static analysis tools, control-flow graph (CFG) and data-flow graph (DFG). The top-level of its structure is made up of the CFG, and each node of that graph itself contains their own DFGs. The different DFGs are interconnected by additional data-flow edges that may cross the CFG node boundaries. Since the language extension is compatible with the C language, the Clang libraries can be used directly to obtain a source-level CFG. In addition, a custom Clang-based data-flow analysis is performed. Although data-flow analysis at the source level is challenging, because of the various complex statements compared to the IR or binary level, this route was chosen to be compatible with other source-level tools (e.g. verification tools) and compiler toolchains. It is also challenging and inaccurate to map the analysis on those lower levels back to source-level statements. The two analysis graphs are then merged and interconnected to form the complete CDFG for each function. The CDFG then contains edges between bit field accesses that also specify whether the dependency is caused by a write following a read, a write following a write or a read following a write. CDFG analysis is used in the proposed method because it reveals data-flow dependencies between bit field accesses that prevent statement reordering and merging as valid optimization transformations. Otherwise, if there is no explicit side effect specified in the bit field group definitions, reordering and merging is generally allowed for the most optimization potential.

6.4.2 Bit Field Access Conflict Graph (BFACG)

To represent all dependencies, a Bit Field Access Conflict Graph (BFACG) is defined as graph $BFACG(A, E)$ with nodes $A = \{a_1, a_2, ..., a_n\}$ representing specific accesses and edges $E = \{e_1, e_2, ..., e_m\}$ representing conflicts between accesses. An access a_i represents an individual read or write bit field access in a source code function. A conflict e_i arises in the following situations.

- A control-flow dependency in the CDFG causes a conflict if the predecessor needs to be evaluated to determine the result of a branch that determines whether or

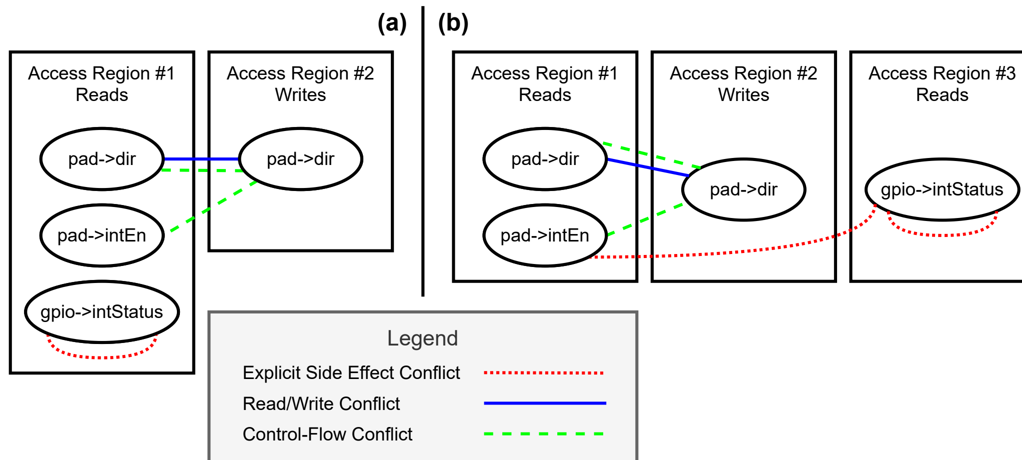


Figure 6.5: BFACG of the function combine_example.

not the successor is evaluated. For example, in the statement `if(gpio->dir) { gpio->out = 1; }`, the second bit field access may not be reordered before the first one, and they may not be combined because either operation would change the behavior of the code.

- A data-flow dependency in the CDFG causes a conflict if a write access follows a read access to the same bit field or vice versa. While it would have been possible to define the language extension in a way that generally permits reordering of such sequences and requiring explicit side effect definitions to prevent it, it would be highly unintuitive to any C programmer and there are diminished benefits because read and write accesses can not be combined into single operations anyways.

- Lastly, explicit side effect definitions cause conflicts whenever the operation type (read or write) and target bit field specification match the side effect definition. For example, if a bit field is defined as `bit wse(other) reset;`, a write to `reset`, followed by a read from `other` will cause a conflict between the two accesses.

One BFACG is created per source code function. By using standard graph coloring on each BFACG, nodes without any conflicts are revealed as ones with the same color. From this, *access regions* are defined that represent a group of accesses which may all be combined into a single one.

Figure 6.5 shows a BFACG for the following driver function.

```
1  void combine_example(bfGroup GPIOPad *pad) {
2    int dir = pad->dir;
3    if (pad->intEn && dir)
4      pad->dir = gpio->intStatus != 0;
5  }
```

Variant (a) represents the BFACG with the bit field definitions as described so far. The explicit side effect definition does not have an effect since there is only one read access to `intStatus`. There is a write following a read to `dir` which adds an implicit conflict edge.

The further edges between the write to dir and reads from dir and intEn are caused by the control-flow dependency of the if expression. In summary, all read accesses could be combined in this case. The second variant (b) shows the BFACG if there would be an additional explicit side effect rse(intEn) on the intStatus bit field. In that case, the read from intStatus is isolated in a separate access region and may not be combined with the other two reads. The upcoming register layout optimization can take into account that it may be beneficial to place all bit fields in the same access regions into a common register.

6.4.3 Bit Field Group Simplification

The bit field group definitions represent a high-level hierarchy of different types with possible array multiplicity. Iterating through an array should produce fast and compact code, which is only possible if all elements of the array have a regular interval in the address space. Accessing bit field groups through dynamic pointers similarly requires all occurrences to have the same layout to achieve fast and compact code. When this abstraction is not exercised in behavior code by dynamic pointers or dynamic array indexing, they cause unnecessary constraints for the creation of the register layout. This is resolved by flattening out the bit field group hierarchy depending on the behavior code. If array fields are only accessed by statically determinable array indices, they are expanded into their individual array elements. That way, the array elements are no longer required to be contiguous in memory and can be reordered more flexibly depending on their usage, for example, to be combined with other bit field accesses. Furthermore, the bit fields of a bit field group are integrated into their parent bit field group if the child group is never dynamically accessed by pointer. This again increases flexibility, since the child and parent bit fields can now be placed in an arbitrary order. These two methods are applied repeatedly on the tree of definitions until no longer possible. The following code snippet shows both transformations on the GPIO peripheral example, which would only be possible when assuming that there are no dynamic accesses.

```
1  bfGroup GPIOType {
2      bit pads_0_dir;
3      bit pads_0_in;
4      bit pads_0_out;
5      bit pads_0_intEn;
6      uint2 pads_0_intType;
7      uint8 pads_0_cfg;
8      bit pads_1_dir;
9      // ...
10     uint8 pads_31_cfg;
11     uint32 rse(intStatus) intStatus;
12 };
```

6.4.4 Heuristic Algorithm

Using the BFACG of each source code function, the proposed heuristic algorithm creates the register layout by mapping all bit fields into registers. The goal is to create a layout that results in software with minimized code size, run time, number of bus accesses and size of the memory map for the device. Although it is possible to formulate an ILP description of the code size optimization program, its complexity and number of constraints make it infeasible to solve real problems this way. To achieve a good result across all mentioned metrics at the cost of an increased register map size, the following heuristic is suggested. First, a sorted

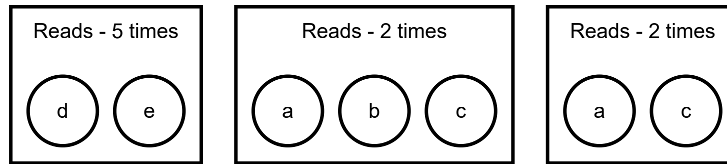


Figure 6.6: Access regions of example for heuristic optimization approach.

list `C` of list of bit fields is created. Each entry, called *combination*, holds a subset of bit fields that could be accessed jointly in a function, because they are part of the same access region (the same color in the colored BFACG). The sorting of `C` is done first by the number of access regions that the combination is part of, that is, at how many positions in the code the accesses may be combined. If that is equal, `C` is sorted by the number of bit fields inside the combinations to favor larger ones. Sorting ensures that frequent combinations in the code are prioritized over infrequent ones, because they restrict which further combinations may be formed.

The algorithm starts at the deepest level of the bit field group hierarchy, with the groups that only contain bit fields and no bit field group members. For each bit field, the sorted list `C` is iterated. If the bit field is part of a combination, all bit fields of that combination are mapped to a common register of the register layout, as long as they still fit. If the bit field is not part of any combination, it is mapped into a new exclusive register. The higher levels of hierarchy may be bit field groups that contain both bit fields and other child groups as members. The bit fields are mapped as before to registers, while the registers that have already been determined for the child groups are appended to the register list of the parent group. It is important to note that if there are different instances of the same bit field group type, then the mapping of the contained bit fields and bit field groups is only done once and reused. As already mentioned in Section 6.4.3, this ensures that the functions that use the base pointer of this group type are homogeneous for all possible instances. For the same reason of small and fast homogeneous code, if a combination contains arrays of bit fields that have the same array size, the heuristic maps them element-wise to registers, e.g., the first register holds `a[0],b[0]`, the second holds `a[1],b[1]`, etc. This allows to generate access functions for combined accesses that support dynamic indexing.

To visualize this heuristic approach, suppose the following bit field group definitions are given along with the access regions in Figure 6.6.

```
1  bfGroup Inner1 {
2      uint8 a;
3      uint8 b;
4      uint8 c;
5  };
6  bfGroup Inner2 {
7      uint8 d;
8      uint8 e;
9  };
10 bfGroup Outer {
11     bfGroup Inner1 inner1;
12     bfGroup Inner2 inner2;
13     uint8 f;
14 };
15 make_device(Outer, outer, 0x1000);
```

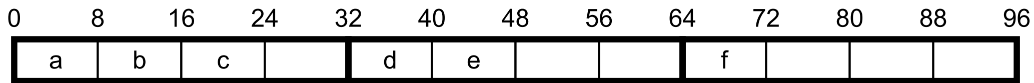


Figure 6.7: Register layout of example for heuristic optimization approach.

First, the list `C` is built by adding access combinations with the most uses. In this example, the first access region is used the most with 5 times. The two remaining access regions are both used twice, so they are sorted by the number of bit fields contained in them. `C` is now a sorted list with the combinations `d,e`, `a,b,c` and `a,c`. Next, the deepest level bit fields are placed first, in this case starting with the bit field group `Inner1` and its first bit field `a`. The first combination in `C` that contains `a` is `a,b,c`, so `a` is placed in a register together with `b` and `c`. The remaining fields of `Inner1` are checked next, but they have already been placed, so this bit field group is complete. `Outer` can still not be processed because it contains another bit field group that has not been processed, yet. When processing `Inner2` and its first bit field `d`, the combination `d,e` is found first and causes these two to be placed into a common register. Finally, the bit field `f` in `Outer` can be processed, but since it is not part of any combination, it is placed in its own separate register. With a register size of 32 bits, the final obtained register layout is shown in Figure 6.7.

6.5 Automated Code Generation

With the final register layout available, it is possible to generate the final output C source code from the input using the language extension. The core task is the transformation from bit field accesses to actual memory accesses, realized through HAL functions. Again, since the extension is compatible with standard C, the Clang libraries are used for source-to-source transformations [53].

For every bit field access or combination of bit field accesses, a HAL function call is generated. For example, a write access to the `pad->dir` bit field in the `GPIOPad` bit field group will emit a function call to `HAL_WRITE_GPIOPad_dir` with the arguments `pad` of pointer type `GPIOPad` and the value to write. Every function call needs a corresponding HAL function definition. These can then be homogeneous for any given input pointer. The full definition of the example HAL function from the code generator is given below.

```
1  static inline uint8_t HAL_WRITE_GPIOPad_dir(struct GPIOPad *grp,
2                                              uint8_t write_value0) {
3      uint8_t volatile *p =
4          (uint8_t volatile *)((uintptr_t)grp + STATIC_BYTE_OFFSET);
5      uint8_t r = 0;
6      r |= write_value0 << STATIC_BIT_OFFSET;
7      *p = r;
8      return write_value0;
9  }
```

The core idea is that the dynamically given bit field group pointer is used as base pointer and the static offset from the register layout optimization is added to form the final access address. The implementation makes use of shift and mask operations to extract bit fields from registers. If bit fields are set that share their containing register with other bit fields that should not be set, a read-modify-write operation must be emitted. If there was

Table 6.1: Required operations for different access types.

Type	Aligned	Exclusive	#Bus Access	#Shift Ops	#Mask Ops
Read	Yes	Yes	1	0	0
Read	No	Yes	1	1	0
Read	Yes	No	1	0	1
Read	No	No	1	1	1
Write	Yes	Yes	1	0	0
Write	No	Yes	1	1	0
Write	Yes	No	3	0	1
Write	No	No	3	1	1

no dynamic access through a `GPIOPad` pointer and the bit field groups were simplified as in Section 6.4.3, the HAL function would instead be named `HAL_WRITE_GPIOType_pads_0_dir` and take a parameter of pointer type `GPIOType`. Further complicating factors are dynamic array accesses, accesses into registers that occupy multiple bit fields, and, of course, combined accesses. One major motivation for register layout optimization is the combination of accesses. When the code generator identifies that certain read or write accesses have no conflict in the BFACG of a given function and are located in the same register of the optimized layout, a HAL function is generated that allows combined access to those bit fields. This improves code size, run time and the number of bus accesses as only a single read or write is required. An example implementation of a combined access that also requires a read-modify-write sequence, i.e. a third bit field shares the register, is given below.

```
1  static inline uint8_t HAL_WRITE_GPIOPad_dir_intEn(struct GPIOPad *grp,
2                                                  uint8_t write_value0
3                                                  uint8_t write_value1) {
4      uint32_t volatile *p =
5          (uint32_t volatile *)((uintptr_t)grp + STATIC_BYTE_OFFSET);
6      uint32_t r = (*p & STATIC_MASK);
7      r |= write_value0 << STATIC_BIT_OFFSET0;
8      r |= write_value1 << STATIC_BIT_OFFSET1;
9      *p = r;
10     return write_value0;
11 }
```

Keeping the hierarchical bit field groups and adding a HAL function layer before the low-level code makes the behavior code highly readable and debugable. HAL functions of bit field arrays will take an additional parameter that represents the array index. HAL functions of combined accesses take as many additional arguments as the number of bit fields they combine. For read HALs those are output parameters of pointer type and for write HALs those are the values to be written. Table 6.1 gives a reference as to how many low-level operations are required for different types of accesses. If an access is aligned to a memory address boundary, no shift operation is required. If the accessed bit field is exclusive, or the only one in a register, no masking operations are required. Write accesses to shared registers require three accesses for the read-modify-write sequence.

Group simplifications break the direct relation between bit field groups and the access expressions in the behavior code. However, the source code may use the original bit field group names that would no longer be recognized after simplification. One solution to this is the generation of *support structs* whose sole purpose is to establish a relation between hierarchical types. These consist of just bit field group member definitions without all the bit fields, since those are accessed through HAL functions by address and not by name. The

support structs of the PULPino GPIO driver look like the following.

```
1  struct GPIOPad
2  {
3      uint8_t pad0[20];
4      // size: 0x00000014
5  } __attribute__((packed));
6  struct GPIOType
7  {
8      uint8_t pad0[4];
9      // 0x00000004
10     struct GPIOPad pads[32];
11     uint8_t pad24[620];
12     // size: 0x00000284
13 } __attribute__((packed));
```

With this, an existing expression that represents a bit field group pointer such as `gpio->pads[i + 1]` can be reused in the generated code to pass to a HAL function and avoid complex expression parsing. For this to work, the size of the support structs has to match their size after register layout optimization, which can be realized with padding fields and the `packed` attribute.

The code generation is illustrated on the example introduced in Section 6.4.2. The BFACG in variant (b) shows that the accesses to `dir` and `intEn` can be combined. In the generated code, this produces a HAL function call that retrieves both values with a single read, as shown in the generated code below.

```
1  void combine_example(bfGroup GPIOPad *pad) {
2    uint8_t tmp1, tmp2;
3    HAL_READ_GPIOPad_dir_intEn(pad, &tmp1, &tmp2);
4    int dir = tmp1;
5    if (tmp2 && dir)
6      HAL_WRITE_GPIOPad_dir(pad,
7        HAL_READ_GPIOType_intStatus(gpio) != 0);
8  }
```

The code generators are easily adaptable to new required formats because they are built on a common abstract software model. For example, for the evaluation in Section 6.6, SystemC headers are generated that contain the chosen register layout. Similarly, it is possible to generate IP-Xact descriptions or VHDL or Verilog code to export the final register layout to other tools for the hardware implementation.

6.6 Experimental Results

The proposed flow was implemented to generate the drivers for the GPIO and Serial Peripheral Interface (SPI) peripherals of a PULPino SoC. The static driver code size was retrieved from the compiled binary optimized for size with `-Os`. The size of the register memory map is a trivial output of the generator when allocating registers.

To evaluate the performance and correctness of the driver generator, a SystemC virtual prototype (VP) based on the instruction set simulator ETISS [70] was used. The SystemC modules for the peripherals were implemented so that the register layout can be dynamically changed using a generated header file that specifies the bit field offsets. The VP simulations allow to measure the estimated number of CPU cycles and the number of peripheral bus accesses. The driver is exercised from a main function with some initialization and a simple loop over four GPIO pads. The SPI is exercised by sending and receiving a data frame.

Table 6.2: PULPino driver evaluation for GPIO and SPI.

Variant	Runtime (cycles)	Memory map size (bytes)	Code size (bytes)	Number of accesses	Halstead Effort
original GPIO	639	64	448	100	115k
optimized GPIO	257	840	276	62	17k
original SPI	557	40	728	25	142k
optimized SPI	318	68	640	18	141k
original GPIO+SPI	1196	104	1176	125	257k
optimized GPIO+SPI	575	908	916	80	158k

The generated drivers were formally verified with the ACCESS method [85]. It initially raised issues with the alignment of read and write accesses because the modeled platform only allowed 4-byte memory accesses at 4-byte boundaries. Another issue was found with a missing side effect annotation for the bit field `in` in the `GPIOPad` bit field group. Both issues were trivial to fix.

Table 6.2 compares the metrics investigated for the original PULPino GPIO driver and the optimized solution based on the heuristic described above. As can be seen, code size, run time and number of required accesses could be reduced significantly by 38%/12%, 60%/43% and 38%/28% for GPIO and SPI, respectively, but at the cost of a larger memory map. Another improvement lies in the simpler description of the driver behavior using the C language extension, which is measured with the Halstead effort [39]. Here, the effort could be reduced by 85% and 1%. The complexity of SPI code is not reduced because the original driver forwards many registers directly to the user, so bit field extraction happens in the application.

The results show that the C language extension with the register layout optimization and code generator produces software that has significantly reduced run time, code size and number of bus accesses. The language extension encourages and supports clean and reusable behavior code, as demonstrated by the Halstead effort.

6.7 Summary

This chapter presented an approach to the automated generation and optimization of hardware/software interfaces in device drivers. The interface is first defined in a novel C language extension using bit field groups and side effect definitions. From that representation and the analyzed driver source code, conflicts between bit field accesses are gathered as constraints for the presented heuristic optimization method. The generated HAL functions of the approach use base-pointers to be more generic and facilitate code reuse for small memory footprints. The presented flow additionally allows generation of an array index as an argument to the HAL function to exploit more generic and reusable code patterns.

Chapter 7

Conclusion and Outlook

This thesis contributed to the optimization of software in constrained devices. Specific focus has been laid on memory-constrained devices because memory is a major contributor to cost, power consumption and size of low-power edge devices such as microcontroller units. First, the use case of machine learning inference on constrained devices was considered. There is increasing demand to run powerful machine learning models on tiny devices. The central contribution of this thesis is the concept of Fused Depthwise Tiling (FDT) as a way to transform the operations of machine learning models to reduce the memory usage of inference tasks. One application is the optimization of working memory during inference, as presented in Chapter 4. A state-of-the-art end-to-end deployment flow was developed for its evaluation. In TinyML scenarios, integrating FDT either allows the use of smaller memories or increases the run time budget. Another application of this novel method is the optimization of distributed inference using multiple cooperating devices as presented in Chapter 5. FDT allows to reduce the static memory usage and requires less communication than existing methods, which results in a faster run time through lower communication demand. This was combined with existing deployment flows to achieve distributed inference that is fully able to scale down memory usage by the number of cooperating devices. Finally, in Chapter 6 a driver generation flow was proposed that reduces the development effort and memory usage of MCUs by abstracting traditional fixed bit field offsets while being able to define simple constraints that promote code reuse. This abstraction was implemented as a C language extension in which the bit fields and their side effects can be described. The approach finds an optimized solution for the layout of the registers with dependency analysis and heuristic grouping of access operations. All presented approaches have in common that they would require intricate fine-tuned implementations of specific use cases that do not translate well to different problems. The contributions of this thesis solve this with a strong emphasis on automated code generation. They are generalized to a wide range of applications, in particular, different machine learning models and different drivers.

The results obtained could be extended further, especially in the area of fused tiling. Additional model types and operations could be supported to demonstrate the benefits in a wider range of models. Furthermore, there is room for exploration how fused tiling interacts with neural architecture search and how it performs in additional use cases such as a heterogeneous system with different types of processor cores and accelerators.

Acronyms

BFACG Bit Field Access Conflict Graph. 85–88, 90, 91

CDFG Control-Data-Flow Graph. 85, 86

CFG control-flow graph. 85

CNN convolutional neural network. 23, 24, 56, 70

DFG data-flow graph. 85

DNN deep neural network. 22–27, 29–31, 33, 34, 36–39, 41, 42, 44–47, 50–52, 54, 55, 57, 59–65, 67, 68, 71–73, 76

DSL domain specific language. 35, 79

FDT Fused Depthwise Tiling. 37–40, 48, 52, 56, 57, 61, 64, 66–70, 72, 73, 76, 93

FFMT Fused Feature Map Tiling. 31–34, 37, 39–41, 50, 52, 56, 57, 61, 64, 68, 72, 76

GPIO general-purpose input/output. 80, 81, 83, 84, 87, 91, 92

HAL hardware abstraction layer. 19, 78, 83, 89–91

ILP integer linear programming. 35, 68–72, 76, 87

IoT Internet of Things. 20, 21

IR intermediate representation. 25, 26, 53, 85

MILP mixed integer linear programming. 42–45, 55, 57

NAS neural architecture search. 21, 30

OWP Optimized Weight Partitioning. 69, 71–76

RAM random-access memory. 19, 30, 35, 39, 55, 56, 61, 72

ROM read-only memory. 19, 30, 35, 39, 55, 57, 61

SPI Serial Peripheral Interface. 91, 92

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283. 2016.

[2] Maytham S Ahmed, Azah Mohamed, Raad Z Homod, Hussain Shareef, Ahmad H Sabry and Khairuddin Bin Khalid. "Smart Plug Prototype for Monitoring Electrical Appliances in Home Energy Management System". In: *2015 IEEE Student Conference on Research and Development (SCOReD)*, pages 32–36. IEEE, 2015.

[3] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou and Hadi Esmaeilzadeh. "Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices". *Proceedings of Machine Learning and Systems*, volume 2, pages 44–57, 2020.

[4] Manoj Alwani, Han Chen, Michael Ferdman and Peter Milder. "Fused-Layer CNN Accelerators". In: *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. 2016.

[5] Said Amellal and Bozena Kaminska. "Scheduling of a Control Data Flow Graph". In: *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1666–1669. IEEE, 1993.

[6] Yohei Arahori, Takashi Imamichi and Hiroshi Nagamochi. "An Exact Strip Packing Algorithm Based on Canonical Forms". *Computers & Operations Research*, volume 39, no. 12, pages 2991–3011, 2012.

[7] Moises Arredondo-Velazquez, Javier Diaz-Carmona, Cesar Torres-Huitzil, Alfredo Padilla-Medina and Juan Prado-Olivarez. "A Streaming Architecture for Convolutional Neural Networks Based on Layer Operations Chaining". *Journal of Real-Time Image Processing*, volume 17, pages 1715–1733, 2020.

[8] Kevin Ashton et al. "That 'Internet of Things' Thing". *RFID journal*, volume 22, no. 7, pages 97–114, 2009.

[9] Babajide O Ayinde, Tamer Inanc and Jacek M Zurada. "Redundant Feature Pruning for Accelerated Inference in Deep Neural Networks". *Neural Networks*, volume 118, pages 148–158, 2019.

[10] Egon Balas. “Disjunctive Programming”. *Annals of Discrete Mathematics*, volume 5, pages 3–51, 1979.

[11] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau et al. “MLPerf Tiny Benchmark”. *arXiv preprint arXiv:2106.07597*, 2021.

[12] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina and Paul Whatmough. “MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers”. *Proceedings of Machine Learning and Systems*, volume 3, pages 517–532, 2021.

[13] Matthew Barrett. “Miniaturizing Models for microNPUs: a Cascading Scheduler for TVM”. TVMCon Presentation, 2021. ARM, Online: https://tvmconf.org/2021/index.html%3Fp=818.html, Accessed: August 2023.

[14] Yoshua Bengio et al. “Learning Deep Architectures for AI”. *Foundations and trends® in Machine Learning*, volume 2, no. 1, pages 1–127, 2009.

[15] Sourav Bhattacharya and Nicholas D Lane. “Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables”. In: *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*, pages 176–189. 2016.

[16] Johannes Bisschop. *AIMMS Optimization Modeling*. Paragon Decision Technology, 2006. ISBN 978-1-84753-912-0.

[17] Ching-Han Chen, Ming-Yi Lin and Chung-Chi Liu. “Edge Computing Gateway of the Industrial Internet of Things Using Multiple Collaborative Microcontrollers”. *IEEE Network*, volume 32, no. 1, pages 24–32, 2018.

[18] Jienan Chen, Siyu Chen, Qi Wang, Bin Cao, Gang Feng and Jianhao Hu. “iRAF: A Deep Reinforcement Learning Approach for Collaborative Mobile Edge Computing IoT Networks”. *IEEE Internet of Things Journal*, volume 6, no. 4, pages 7011–7024, 2019.

[19] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang and Zheng Zhang. “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems”. *arXiv preprint arXiv:1512.01274*, 2015.

[20] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze et al. “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594. 2018.

[21] Hsin-Pai Cheng, Tunhou Zhang, Yukun Yang, Feng Yan, Shiyu Li, Harris Teague, Hai Li and Yiran Chen. “SwiftNet: Using Graph Propagation as Meta-Knowledge to

Search Highly Representative Neural Architectures". *arXiv preprint arXiv:1906.08305*, 2019.

[22] Shao-Yi Chien, Wei-Kai Chan, Yu-Hsiang Tseng, Chia-Han Lee, V Srinivasa Somayazulu and Yen-Kuang Chen. "Distributed Computing in IoT: System-on-a-Chip for Smart Cameras as an Example". In: *The 20th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 130–135. IEEE, 2015.

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

[24] Antonio Cipolletta and Andrea Calimera. "Dataflow Restructuring for Active Memory Reduction in Deep Neural Networks". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 114–119. IEEE, 2021.

[25] Steven Colleman and Marian Verhelst. "High-Utilization, High-Flexibility Depth-First CNN Coprocessor for Image Pixel Processing on FPGA". *IEEE Transactions on Very Large Scale Integration Systems*, volume 29, no. 3, pages 461–471, 2021.

[26] Christopher L Conway and Stephen A Edwards. "NDL: a Domain-Specific Language for Device Drivers". In: *ACM Sigplan Notices*, volume 39, no. 7, pages 30–36. ACM New York, NY, USA, 2004.

[27] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv and Yoshua Bengio. "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". *arXiv preprint arXiv:1602.02830*, 2016.

[28] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang et al. "TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems". *Proceedings of Machine Learning and Systems*, volume 3, pages 800–811, 2021.

[29] Ujjwal Dahal Deep, Brent R Petersen and Julian Meng. "A Smart Microcontroller-Based Iridium Satellite-Communication Architecture for a Remote Renewable Energy Source". *IEEE Transactions on Power Delivery*, volume 24, no. 4, pages 1869–1875, 2009.

[30] Wolfgang Ecker, Wolfgang Müller and Rainer Dömer. *Hardware-Dependent Software: Principles and Practice*. Springer, 2009. ISBN 978-1-4020-9435-4.

[31] John Forrest, Stefan Vigerske, Haroldo Gambini Santos, Ted Ralphs, Lou Hafer, Bjarni Kristjansson et al. "coin-or/Cbc: Version 2.10.5", 2020. doi:10.5281/zenodo.3700700. Zenodo.

[32] B Gayathri, K Sruthi and KA Unnikrishna Menon. "Non-invasive Blood Glucose Monitoring Using Near Infrared Spectroscopy". In: *2017 International Conference on Communication and Signal Processing (ICCSP)*, pages 1139–1142. IEEE, 2017.

[33] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney and Kurt Keutzer. "A Survey of Quantization Methods for Efficient Neural Network Inference". In: *Low-Power Computer Vision*, pages 291–326. Chapman and Hall/CRC, 2022.

[34] Google. "TensorFlow Text Classification", 2022. Online: https://www.tensorflow.org/lite/examples/text_classification/overview, Accessed: August 2023.

[35] Google. "TensorFlow Lite", 2023. Online: https://www.tensorflow.org/lite, Accessed: August 2023.

[36] Marc Monfort Grau, Roger Pueyo Centelles and Felix Freitag. "On-Device Training of Machine Learning Models on Microcontrollers with a Look at Federated Learning". In: *Proceedings of the Conference on Information Technology for Social Good*, pages 198–203. 2021.

[37] Jinyang Guo, Wanli Ouyang and Dong Xu. "Channel Pruning Guided by Classification Loss and Feature Importance". In: *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, no. 7, pages 10885–10892. 2020.

[38] Gurobi Optimization, LLC. "Gurobi Optimizer Reference Manual", 2022. Online: https://www.gurobi.com, Accessed: August 2023.

[39] Maurice Howard Halstead. *Elements of Software Science (Operating and Programming Systems Series)*, volume 7. Elsevier Science Inc., 1977. ISBN 978-0-444-00215-0.

[40] Mehedi Hasan, Maruf Hossain Anik and Sharnali Islam. "Microcontroller Based Smart Home System with Enhanced Appliance Switching Capacity". In: *2018 Fifth HCT Information Technology Trends (ITT)*, pages 364–367. IEEE, 2018.

[41] Yihui He, Xiangyu Zhang and Jian Sun. "Channel Pruning for Accelerating Very Deep Neural Networks". In: *Proceedings of the IEEE International Conference on Computer Vision*, pages 1389–1397. 2017.

[42] Loc Nguyen Huynh, Rajesh Krishna Balan and Youngki Lee. "DeepSense: A GPU-Based Deep Convolutional Neural Network Framework on Commodity Mobile Devices". In: *Proceedings of the 2016 Workshop on Wearable Systems and Applications*, pages 25–30. ACM, 2016.

[43] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally and Kurt Keutzer. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size". *arXiv preprint arXiv:1602.07360*, 2016.

[44] ISO. *ISO/IEC 9899:2018 Information Technology — Programming languages — C.* ISO, 2018.

[45] Pragathi Jayaram. *Real-Time Hand Gesture Classification on a MCU with Continuous Wave Radar and a Convolutional Neural Network.* Master's thesis, Technical University of Munich, 2021.

[46] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12. 2017.

[47] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars and Lingjia Tang. "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge". *ACM SIGARCH Computer Architecture News*, volume 45, no. 1, pages 615–629, 2017.

[48] Enver Kayaaslan, Thomas Lambert, Loris Marchal and Bora Uçar. "Scheduling Series-Parallel Task Graphs to Minimize Peak Memory". *Theoretical Computer Science*, volume 707, pages 1–23, 2018.

[49] Hakima Khelifi, Senlin Luo, Boubakr Nour, Akrem Sellami, Hassine Moungla, Syed Hassan Ahmed and Mohsen Guizani. "Bringing Deep Learning at the Edge of Information-Centric Internet of Things". *IEEE Communications Letters*, volume 23, no. 1, pages 52–55, 2018.

[50] Alex Krizhevsky, Geoffrey Hinton et al. "Learning Multiple Layers of Features from Tiny Images". Technical report, University of Toronto, 2009.

[51] Alex Krizhevsky, Ilya Sutskever and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". *Communications of the ACM*, volume 60, no. 6, pages 84–90, 2017.

[52] Hamed F Langroudi, Vedant Karia, Tej Pandit and Dhireesha Kudithipudi. "TENT: Efficient Quantization of Neural Networks on the Tiny Edge with Tapered Fixed Point". *arXiv preprint arXiv:2104.02233*, 2021.

[53] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004.

[54] Andrew Lavin and Scott Gray. "Fast Algorithms for Convolutional Neural Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021. 2016.

[55] Yen-Lin Lee, Pei-Kuei Tsung and Max Wu. "Techology Trend of Edge AI". In: *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–2. IEEE, 2018.

[56] Edgar Liberis and Nicholas D Lane. "Neural Networks on Microcontrollers: Saving Memory at Inference via Operator Reordering". *arXiv preprint arXiv:1910.05110*, 2019.

[57] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan and Song Han. "MCUNetV2: Memory-Efficient Patch-Based Inference for Tiny Deep Learning". *arXiv preprint arXiv:2110.15352*, 2021.

[58] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan and Song Han. "MCUNet: Tiny Deep Learning on IoT Devices". *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 11711–11722, 2020.

[59] Kuan Jen Lin, Shih Hao Huang and Shih Wen Chen. “Optimal Allocation of I/O Device Parameters in Hardware and Software Codesign Methodology”. In: *Proceedings of the Embedded and Ubiquitous Computing: International Conference (EUC), Taipei, Taiwan*, pages 541–552. Springer, 2007.

[60] G Jack Lipovski. *Introduction to Microcontrollers: Architecture, Programming, and Interfacing for the Freescale 68HC12*. Elsevier, 2004. ISBN 978-0-08-047041-2.

[61] Joseph WH Liu. “An Application of Generalized Tree Pebbling to Sparse Matrix Factorization”. *SIAM Journal on Algebraic Discrete Methods*, volume 8, no. 3, pages 375–395, 1987.

[62] Jonathan Long, Evan Shelhamer and Trevor Darrell. “Fully Convolutional Networks for Semantic Segmentation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440. 2015.

[63] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng and Christopher Potts. “Learning Word Vectors for Sentiment Analysis”. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150. 2011.

[64] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger and Yiran Chen. “MoDNN: Local Distributed Mobile Computing System for Deep Neural Network”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1396–1401. IEEE, 2017.

[65] Fabíola Martins Campos de Oliveira and Edson Borin. “Partitioning Convolutional Neural Networks to Maximize the Inference Rate on Constrained IoT Devices”. *Future Internet*, volume 11, no. 10, page 209, 2019.

[66] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet and Gilles Muller. “Devil: An IDL for Hardware Programming”. In: *Fourth Symposium on Operating Systems Design and Implementation (OSDI)*. 2000.

[67] Svetlana Minakova and Todor Stefanov. “Buffer Sizes Reduction for Memory-Efficient CNN Inference on Mobile and Embedded Devices”. In: *23rd Euromicro Conference on Digital System Design (DSD)*, pages 133–140. IEEE, 2020.

[68] Svetlana Minakova and Todor Stefanov. “Memory-Throughput Trade-off for CNN-Based Applications at the Edge”. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, volume 28, no. 1, pages 1–26, 2022.

[69] Mohammad Motamedi, Daniel Fong and Soheil Ghiasi. “Fast and Energy-Efficient CNN Inference on IoT Devices”. *arXiv preprint arXiv:1611.07151*, 2016.

[70] Daniel Mueller-Gritschneder, Keerthikumara Devarajegowda, Martin Dittrich, Wolfgang Ecker, Marc Greim and Ulf Schlichtmann. “The Extendable Translating Instruction Set Simulator (ETISS) Interlinked with an MDA Framework for Fast RISC Prototyping”. In: *Proceedings of the 28th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, pages 79–84. 2017.

[71] Johannes Obermaier and Stefan Tatschner. "Shedding too much Light on a Microcontroller's Firmware Protection". In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. 2017.

[72] George Papandreou, Tyler Zhu, Liang-Chieh Chen, Spyros Gidaris, Jonathan Tompson and Kevin Murphy. "PersonLab: Person Pose Estimation and Instance Segmentation with a Bottom-Up, Part-Based, Geometric Embedding Model". In: *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 269–286. 2018.

[73] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga et al. "Pytorch: An Imperative Style, High-Performance Deep Learning Library". *Advances in neural information processing systems*, volume 32, 2019.

[74] Saman Payvar, Mir Khan, Rafael Stahl, Daniel Mueller-Gritschneder and Jani Boutellier. "Neural Network-Based Vehicle Image Classification for IoT Devices". In: *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 148–153. IEEE, 2019.

[75] John B Peatman. *Design with PIC Microcontrollers*. Pearson Education India, 1998. ISBN 978-0-13-759259-3.

[76] Laurent Perron and Vincent Furnon. "OR-Tools", 2019. Google, Online: https://developers.google.com/optimization, Accessed: August 2023.

[77] Yury Pisarchyk and Juhyun Lee. "Efficient Memory Management for Deep Neural Net Inference". *CoRR, arXiv preprint arXiv:2001.03288*, 2020.

[78] Joseph Redmon. "Darknet: Open Source Neural Networks in C". Online: `http://pjreddie.com/darknet/`, Accessed: August 2023, 2013.

[79] Joseph Redmon. "Darknet Pretrained Models". Online: `https://pjreddie.com/darknet/yolov2`, `https://pjreddie.com/darknet/imagenet/#pretrained`, Accessed: August 2023, 2013.

[80] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7263–7271. 2017.

[81] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein et al. "Glow: Graph Lowering Compiler Techniques for Neural Networks". *arXiv preprint arXiv:1805.00907*, 2018.

[82] Amit Sabne. "XLA: Compiling Machine Learning for Peak Performance", 2020. Google, Online: https://www.tensorflow.org/xla, Accessed: August 2023.

[83] Yuvraj Sahni, Jiannong Cao and Lei Yang. "Data-Aware Task Allocation for Achieving Low Latency in Collaborative Edge Computing". *IEEE Internet of Things Journal*, volume 6, no. 2, pages 3512–3524, 2018.

[84] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov and Liang-Chieh Chen. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520. 2018.

[85] Michael Schwarz, Rafael Stahl, Daniel Mueller-Gritschneder, Ulf Schlichtmann, Dominik Stoffel and Wolfgang Kunz. “ACCESS: HW/SW Co-Equivalence Checking for Firmware Optimization”. In: *Proceedings of the 56th Annual Design Automation Conference (DAC)*, pages 1–6. 2019.

[86] Taro Sekiyama, Takashi Imamichi, Haruki Imai and Rudy Raymond. “Profile-Guided Memory Optimization for Deep Neural Networks”. *CoRR, arXiv preprint arXiv:1804.10001*, 2018.

[87] Jinfang Sheng, Jie Hu, Xiaoyu Teng, Bin Wang and Xiaoxia Pan. “Computation Offloading Strategy in Mobile Edge Computing”. *Information*, volume 10, no. 6, page 191, 2019.

[88] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. *arXiv preprint arXiv:1409.1556*, 2014.

[89] Rafael Stahl, Alexander Hoffman, Daniel Mueller-Gritschneder, Andreas Gerstlauer and Ulf Schlichtmann. “DeeperThings: Fully Distributed CNN Inference on Resource-Constrained Edge Devices”. *International Journal of Parallel Programming*, volume 49, pages 600–624, 2021.

[90] Rafael Stahl, Daniel Mueller-Gritschneder and Ulf Schlichtmann. “Driver Generation for IoT Nodes with Optimization of the Hardware/Software Interface”. *IEEE Embedded Systems Letters*, volume 12, no. 2, pages 66–69, 2019.

[91] Rafael Stahl, Daniel Mueller-Gritschneder and Ulf Schlichtmann. “Fused Depthwise Tiling for Memory Optimization in TinyML Deep Neural Network Inference”. In: *TinyML Research Symposium.* arXiv:2303.17878, 2023.

[92] Rafael Stahl, Zhuoran Zhao, Daniel Mueller-Gritschneder, Andreas Gerstlauer and Ulf Schlichtmann. “Fully Distributed Deep Learning Inference on Resource-Constrained Edge Devices”. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation: 19th International Conference (SAMOS)*, pages 77–90. Springer, 2019.

[93] Arthur Stoutchinin, Francesco Conti and Luca Benini. “Optimally Scheduling CNN Convolutions for Efficient Memory Access”. *arXiv preprint arXiv:1902.01492*, 2019.

[94] Jun Sun, Wanghong Yuan, Mahesh Kallahalla and Nayeem Islam. “HAIL: A Language for Easy and Correct Device Access”. In: *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT)*, pages 1–9. 2005.

[95] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang and Joel S Emer. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. *Proceedings of the IEEE*, volume 105, no. 12, pages 2295–2329, 2017.

[96] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke and Andrew Rabinovich. "Going Deeper with Convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9. 2015.

[97] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens and Zbigniew Wojna. "Rethinking the Inception Architecture for Computer Vision". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826. 2016.

[98] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard and Quoc V Le. "MnasNet: Platform-Aware Neural Architecture Search for Mobile". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828. 2019.

[99] Mingxing Tan and Quoc Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[100] Surat Teerapittayanon, Bradley McDanel and Hsiang-Tsung Kung. "Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices". In: *37th International Conference on Distributed Computing Systems (ICDCS)*, pages 328–339. IEEE, 2017.

[101] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K Gurkaynak and Luca Benini. "PULPino: A Small Single-Core RISC-V SoC". In: *3rd RISCV Workshop*. 2016.

[102] Ing Jyh Tsang, Federico Corradi, Manolis Sifalakis, Werner Van Leekwijck and Steven Latré. "Radar-Based Hand Gesture Recognition Using Spiking Neural Networks". *Electronics*, volume 10, no. 12, page 1405, 2021.

[103] Ya Tu and Yun Lin. "Deep Neural Network Compression Technique Towards Efficient Digital Signal Modulation Recognition in Edge Device". *IEEE Access*, volume 7, pages 58113–58119, 2019.

[104] Pete Warden and Daniel Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2019. ISBN 978-1-4920-5199-2.

[105] Lea Wittie. "Laddie: The Language for Automated Device Drivers (Ver 1". Technical report, Bucknell Computer Science, 2008.

[106] Zhuoran Zhao, Kamyar Mirzazad Barijough and Andreas Gerstlauer. "DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters". *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, volume 37, no. 11, pages 2348–2359, 2018.

List of Figures

2.1 Common components of an MCU. 20
2.2 Example machine learning model architecture. 21
2.3 Generic TinyML flow. 22
2.4 Example DNN with three layers. 23
2.5 Example CNN with two layers. 25

3.1 RAM usage of DNN intermediate buffers. 31
3.2 FFMT applied to two consecutive CNN layers. 32

4.1 Automated tiling exploration flow. 38
4.2 FDT applied to two consecutive dense layers. 39
4.3 FDT applied to two consecutive CNN layers. 40
4.4 Architecture of the running example. 41
4.5 Example DNN graph for scheduling. 41
4.6 Example DNN graph transformed as task graph. 42
4.7 Flow of the memory-aware scheduling. 44
4.8 Example of memory buffer planning. 45
4.9 Memory layout of the running example. 46
4.10 Example layout for critical buffer selection. 47
4.11 Path discovery building blocks and supported operations. 48
4.12 Architecture of the running example after transformation. 51
4.13 Memory layout of the running example after transformation. 52

5.1 Memory requirements for the computation of individual model layers. 60
5.2 4-Layer example on a single device. 63
5.3 4-Layer example with sequential layer mapping. 63
5.4 4-Layer example partitioned with FDT Fan-Out. 65
5.5 4-Layer example with FDT. 67
5.6 4-Layer example with Optimized Weight Partitioning. 70
5.7 Optimized Weight Partitioning for different CNNs. 72
5.8 Memory savings results. 74
5.9 Run time speedup of FDTO vs. OWP. 75

6.1 PULPino code size distribution. 77
6.2 Traditional development flow. 78

6.3 Proposed development flow. 78
6.4 Mapping from abstract bit fields to a concrete layout. 81
6.5 BFACG of the function combine_example. 86
6.6 Access regions of example for heuristic optimization approach. 88
6.7 Register layout of example for heuristic optimization approach. 89

List of Tables

3.1 Comparison of Inference Partitioning Methods 35

4.1 Memory reduction of FDT compared to FFMT 56
4.2 Tiling Design Space Exploration with FFMT and FDT 57
4.3 ROM usage of FDT compared to FFMT. 57

5.1 Memory footprint reduction for ten devices. 71
5.2 Communication savings of OWP over LOP for six devices. 71
5.3 Baseline run time values and standard deviation for one device. 76

6.1 Required operations for different access types. 90
6.2 PULPino driver evaluation for GPIO and SPI. 92