Deployment of Deep Learning Workloads to Power-Constrained Platforms

Development of a Power-Aware Deep Learning Compiler for Ultra-Low-Power Edge Devices

Submitted to the School of Computation, Information and Technologyof the Technical University of Munich for the degree of Master of Science (M.Sc.)

Supervised by	Dr.-Ing. Daniel Müller-Gritschneder
	Chair of Electronic Design Automation
Submitted by	Fabian Georg Peddinghaus
Submitted on	March 31, 2023

Declaration of Authorship

I hereby declare that I have written this thesis independently and that I have not used any other sources or aids than those indicated.

Munich, March 31, 2023

Preface

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr.-Ing. Daniel Müller-Gritschneder, for his unwavering support and guidance throughout my master's studies and thesis work. I am also grateful to Prof. Dr.-Ing. Ulf Schlichtmann, head of the Chair of Electronic Design Automation at TU Munich, for providing me with great opportunities and for creating an excellent research environment at the chair. Moreover, I would like to sincerely thank Prof. Subhasish Mitra at Stanford University for hosting me as a visiting student and for the welcoming environment he created.

I would like to thank Arne Symons from KU Leuven. His knowledge, dedication, and friendship have inspired me, making our time as roommates in Menlo Park genuinely memorable. I am also deeply indebted to the Ph.D. students from Stanford: Robert Radway, Kartik Prabhu, and Jeffrey Yu, as well as from TU Munich: Philipp van Kempen and Rafael Stahl. Their assistance and camaraderie have enriched my understanding and greatly impacted this thesis.

Last but not least, I owe my heartfelt thanks to my family, especially my parents, for their unwavering love, support, and belief in me throughout this journey. They have been my rock and a constant source of encouragement, and I dedicate this achievement to them.

Abstract

This master's thesis presents PAMA, the Power- And Memory-Aware deep learning compiler. PAMA enables the deployment of state-of-the-art deep learning models to highly resource-constrained devices by minimizing power consumption and maximizing performance through pattern-based operator fusion and genetic algorithm-based co-optimization of energy and memory resources. While memory planning algorithms in existing deep learning compilers primarily focus on reducing peak memory usage, PAMA is, to the best of our knowledge, the first compiler to address both energy and memory constraints simultaneously. To this end, PAMA supports advanced on-chip memory technologies, such as fine-grained power gating and non-volatile RRAM (Resistive Random Access Memory), which are increasingly prevalent in ultra-low-power applications.

PAMA is also the first compiler to enable energy and latency-optimized deployment on resource-constrained multi-chip systems, bridging the gap between current distributed deep learning frameworks and emerging multi-chip edge devices. PAMA accomplishes this by leveraging the design space exploration capabilities of ZigZag and integrating with Stream, a novel multi-chip mapping framework capable of mapping deep learning models onto heterogeneous multi-chip accelerator systems.

To substantiate our claims, we compare PAMA to current state-of-the-art memory planning algorithms, such as those employed by TVM, across a diverse set of modern deep learning models, including CNNs (Convolutional Neural Networks) and Transformers. Utilizing a configurable multi-chip hardware model, PAMA can target a wide range of single and multi-chip-based edge devices. In its current implementation, PAMA provides backend support for the MINOTAUR SoC (System-on-Chip), a power-optimized deep learning accelerator developed at Stanford University. MINOTAUR incorporates a novel on-chip memory architecture and a highly efficient systolic array-based deep learning accelerator. By employing a unique floating-point-like data type and fine-grained power gating, MINOTAUR achieves state-of-the-art performance for both vision and natural language processing tasks. Multiple MINOTAUR SoCs can be interconnected to form a multi-chip system, capable of scaling to deploy arbitrarily large deep learning models with minimal overhead. The PAMA deep learning compiler fully supports all of MINOTAUR's advanced inference features, with on-device training and multi-chip support currently under development. Using PAMA, we demonstrate that up to 97.35% of MINOTAUR's static on-chip memory energy can be saved while maintaining the same performance and accuracy.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

API Application Programming Interface

BYOC Bring Your Own Codegen

C2C Chip-to-Chip

CLI Command Line Interface

CNN Convolutional Neural Network

CPU Central Processing Unit

DAG Directed Acyclic Graph

DEAP Distributed Evolutionary Algorithms in Python

DMA Direct Memory Access

DNN Deep Neural Network

DSE Design Space Exploration

GPU Graphics Processing Unit

ILP Integer Linear Programming

IoT Internet of Things

IP Intellectual Property

IR Intermediate Representation

LLM Large Language Model

MAC multiply-accumulate

MLP Multi-Layer Perceptron

NDA Non-Disclosure Agreement

ONNX Open Neural Network Exchange

PAMA Power And Memory Aware deep learning compiler

PE Processing Element

PME Pattern Matching Engine

QAT Quantization Aware Training

RRAM Resistive Random Access Memory

RTL Register Transfer Level

SoC System-on-Chip

SotA State-of-the-Art

SRAM Static Random Access Memory

TFLite TensorFlow Lite

TFLM TensorFlow Lite for Microcontrollers

TSMC Taiwan Semiconductor Manufacturing Company

1. Introduction

1.1. Motivation

Over the past decade, deep learning has demonstrated remarkable performance across various tasks, such as image classification [1–3], object detection [4–6], speech recognition [7–9], and natural language processing [10–12]. The exceptional accuracy of DNNs (Deep Neural Networks) in comparison to traditional data-driven methods [13] has led to their widespread adoption in various applications, such as autonomous driving [14, 15], robotics [16, 17], healthcare [18], and augmented/virtual reality (AR/VR) [19–22]. However, the remarkable performance of DNNs is accompanied by high computational complexity [23], necessitating novel hardware and software architectures to efficiently accelerate their computation.

While DNNs are primarily deployed on servers and cloud-based systems today, there is a growing demand to harness the power of deep learning on mobile and edge devices such as drones, smartphones, and the IoT (Internet of Things). Common reasons for deploying DNNs on edge devices instead of the cloud include low latency, privacy, and reduced power consumption requirements. User experience in real-time applications is severely impacted by the delay caused by data transmission to a remote server for processing, leading to issues such as motion sickness in AR/VR applications [24]. Wireless communication for data transmission also significantly contributes to power consumption, reducing the battery life of edge devices [25]. Although edge devices are becoming increasingly powerful and efficient, they still lack the computational resources and energy budgets available to servers. As a result, optimizing the execution of DNNs on edge devices to minimize power consumption and maximize performance is of paramount importance.

This has sparked a growing interest in designing specialized deep learning accelerator architectures that surpass the capabilities of general-purpose compute engines, such as GPUs (Graphics Processing Units). Dedicated deep learning accelerators enable more efficient and power-optimized execution of DNNs, meeting the strict requirements of mobile and edge devices. As a result, research on low-power deep learning accelerators has experienced significant growth, both in academia and industry.

1.2. Problem Statement

While SotA (State-of-the-Art) deep learning accelerators have demonstrated promising results, they still exhibit several limitations. First, they lack support for modern, heterogeneous DNN workloads, such as the multi-head attention mechanism found in Transformer models [12]. This leads to costly offloading of such operations to the CPU (Central Processing Unit), resulting in suboptimal performance and energy efficiency. Second, many accelerators assume full utilization of both compute and memory resources, which is rarely the case in practice. For example, most of an accelerator's weight memory remains unused during inference, with only a small portion being actively read from, leading to wasted energy. Third, the software stack, comprising DNN compilers and mapping tools, has been overlooked by hardware developers in the past. Consequently, much of the deployment of DNNs still relies on labor-intensive and suboptimal manual mapping.

On the software side, numerous deep learning compiler frameworks and mapping tools have emerged, with many catering to resource-constrained edge devices. Among them, the most prominent and mature deep learning compiler is TVM [26]. While TVM supports a variety of hardware backends, including GPUs and specialized accelerators, its capabilities for more advanced hardware features, such as power and memory-aware mapping, remain limited. Furthermore, many of its internal optimization features, such as auto-tuning, are cumbersome to use, do not scale well to large DNNs models, and are not suitable for early pre-silicon design stages.

The disconnect between hardware and software constitutes a major bottleneck in the development of modern deep learning accelerators. While the hardware community has made significant progress in creating efficient deep learning accelerators, the software community has struggled to keep pace with the rapid advancements in hardware development. The software community's efforts remain largely focused on general-purpose compute engines, such as GPUs, rather than specialized accelerators. Consequently, the performance and energy efficiency of deep learning accelerators have not been fully realized. A coordinated effort between hardware and software is needed to develop a holistic solution with comprehensive software support for advanced hardware features.

1.3. Contributions

This thesis aims to address the aforementioned challenges by developing a novel deep learning compiler framework called PAMA, the Power- And Memory-Aware deep learning compiler. PAMA bridges the gap between hardware and software through a number of key contributions:

1. PAMA features a novel annotation-based interface for deep learning hardware mapping tools, allowing for straightforward integration of their optimization and mapping strategies. Utilizing this interface, PAMA integrates with the SotA DNN exploration and mapping framework ZigZag [27]. ZigZag is an effective tool for exploring and optimizing the spatial and temporal mapping of DNNs on deep learning accelerators. PAMA leverages these advanced optimization and mapping strategies to generate optimal loop nests and tilings for DNNs operators.

2. Employing a comprehensive and scalable hardware description model, PAMA can be easily extended to support single-chip and multi-chip architectures, as well as advanced hardware features, like non-volatile on-chip memories. This model contains hardware costs and mapping constraints, which both PAMA and ZigZag use to guide the optimization and mapping process, enabling automated power- and memory-aware mapping of DNNs to deep learning accelerators.

3. Building on this hardware description model, PAMA offers advanced memory planning and optimization capabilities. Much of the focus in deep learning compiler research has been on the accelerator design itself, while the memory subsystem has received much less attention. However, as repeatedly demonstrated in the past [28], the memory hierarchy is a major contributor to energy consumption in DNN computation. To address this issue, PAMA provides a suite of memory planning and optimization strategies, including a novel genetic algorithm-based schedule and memory co-optimization algorithm. Utilizing these memory optimization strategies, PAMA's memory planner signigicantly outperforms SotA DNN compilers in terms of both peak memory usage and energy consumption.

4. Additionally, PAMA can generate code for architectures that support fine-grained power-gating and Bandwidth Aware memory scheduling, resulting in significant energy savings. PAMA also supports advanced on-chip memory technologies, such as non-volatile RRAM, which are increasingly popular in ultra-low-power applications [29–31].

5. During the early-stage development of deep learning accelerators, i.e., before the first silicon tape-out, frequent verification of the accelerator's functionality and performance is crucial. Previous compilers and mapping tools either require significant manual effort or do not support verification at all. To address these challenges, PAMA offers the ability to automatically generate layer-by-layer and end-to-end test benches for a given DNN model, enabling fully automated verification of the accelerator's functionality and performance.

6. Unlike previous mapping tools, PAMA supports a wide range of modern deep learning models, including fully-connected, convolutional, and transformer-style DNNs. With an ONNX (Open Neural Network Exchange)-compatible Frontend, PAMA can be used with popular deep learning frameworks such as PyTorch and TensorFlow.

7. Lastly, PAMA can scale from single to large multi-chip architectures, addressing the rapidly increasing size of DNNs models, such as LLMs (Large Language Models). For optimal mapping across multi-chip architectures, PAMA utilizes Stream [32], an innovative multi-chip mapping framework capable of mapping deep learning models to heterogeneous multi-chip accelerator systems. At the time of writing this thesis, Stream and its integration into PAMA remain under active development.

1.4. Thesis Organization

The organization of this thesis is as follows: Chapter 2 will provide an overview of the MINOTAUR SoC and its architecture, as well as the development and verification challenges associated with taping out a deep learning accelerator chip. Chapter 3 will introduce PAMA, the Power- And Memory-Aware deep learning compiler, covering its integration with external mapping frameworks such as ZigZag, its internal architecture from Frontend to Backend, and its use as a tool for early-stage development and verification of deep learning accelerator chips. Chapter 4 will showcase experimental results and a detailed performance analysis of PAMA on a variety of SotA DNN models. Chapter 5 will offer a summary of the contributions made by this thesis and explore future work. Supplementary background information and details on specific topics can be found in the appendices.

2. MINOTAUR Deep Learning Accelerator

This chapter presents the MINOTAUR deep learning accelerator SoC, highlighting some of its core design decisions and providing a high-level architectural overview pertinent to the development of a compiler such as PAMA. A comprehensive understanding of the SoC is essential for grasping the DSE (Design Space Exploration) and mapping algorithms discussed in subsequent chapters.

2.1. Background

2.1.1. Related Work

Several research groups have designed specialized accelerators to improve the energy efficiency of deep learning applications. Notable examples of previous work on low-power deep learning accelerators include the following:

Envision [33], developed by KU Leuven's ESAT-MICAS research group, is an energy-adaptable CNN processor tailored towards always-on computer vision in wearable devices. It targets efficient and scalable visual recognition tasks, employing a series of increasingly complex CNNs for object recognition and action processing. Envision uses a novel subword-parallel Dynamic-Voltage-Accuracy-Frequency Scaling (DVAFS) technique, achieving up to 10 TOPS/W peak efficiency.

The UltraTrail chip [34], developed by researchers from the University of Tuebingen, is a deep learning accelerator featuring an 8x8 MAC (multiply-accumulate) array specialized for the inference of temporal convolutional residual neural networks (TC-ResNets) for keyword spotting. The chip has a power consumption of 8.2 µW and reaches a 93% accuracy on the Google Speech Command Dataset.

An *all-weights-on-chip* AI processor [35], which, similar to CHIMERA, features full on-chip weight storage using RRAM, was proposed by a research team from the University of Michigan, Ann Arbor, and Meta[1]. The design employs a 4-Core architecture with 24×1 MB em-

[1] Formerly called Facebook.

bedded RRAM. Using on-the-fly weight decompression combined with a mesh-connected PE (Processing Element) architecture and 8 MB shared SRAM (Static Random Access Memory), the proposed accelerator operates at 120 MHz, achieving a efficiency of up to 0.96 TOPS/W.

Despite the potential exhibited by these and similar accelerators [36–38], they encounter several limitations. First, they do not sufficiently support heterogeneous DNN workloads, such as the multi-head attention mechanism in Transformer models, leading to CPU offloading and diminished performance and energy efficiency. Second, these accelerators often fail to fully utilize compute and memory resources, resulting in energy wastage. Third, the software stack, encompassing DNN compilers and mapping tools, has been historically neglected, necessitating suboptimal manual mapping for DNN deployment.

2.1.2. Motivation

MINOTAUR's primary objective was to establish a configurable research platform for ultra-low-power deep learning on mobile and edge devices. It was designed to address the above limitations by supporting a broad range of DNNs and by providing a comprehensive software stack for DNN mapping and compilation.

Similar to its direct predecessor, CHIMERA [29, 39], MINOTAUR incorporates non-volatile RRAM for weight and bias storage, SRAM for activations and intermediate results, and a systolic array-based accelerator. The architecture and physical design of all components have been significantly improved, with the chip expected to be, on average, 2.5x faster and 5x more energy-efficient than CHIMERA. A key enhancement is the adoption of a novel floating-point-like data type called Posit [40, 41]. Additionally, MINOTAUR supports a broader range of DNNs, including modern CNNs architectures [2, 3, 42] and Transformer models [12]. Furthermore, MINOTAUR enables full on-device training of DNNs using an innovative layer-by-layer training method that minimizes the memory required for training.

Developed at Stanford University during the most of 2022 and taped out in early 2023, MINOTAUR utilizes a TSMC 40nm CMOS ULP fabrication process and is capable of operating at frequencies up to 200MHz with a core voltage of 0.9 Volts and a memory system voltage of 1.1 Volts. At the time of writing this thesis, the MINOTAUR chip is currently in fabrication and expected to return for testing and validation in May 2023.

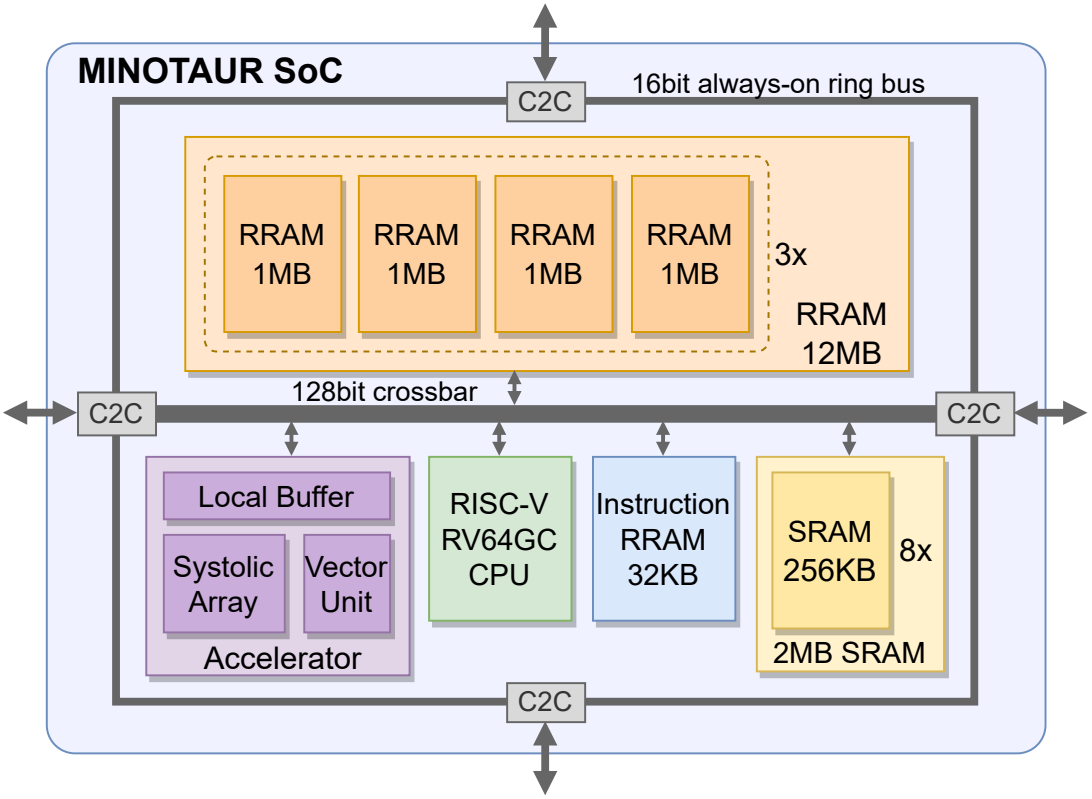

Figure 2.1.: Architectural overview of the MINOTAUR SoC.

2.2. Architecture Overview

The MINOTAUR SoC, as illustrated in Figure 2.1, consists of six primary components: the accelerator, the RISC-V CPU, the activation SRAM, the instruction RRAM, the parameter RRAM, and the C2C (Chip-to-Chip) interconnect. All components connect to a 128-bit-wide crossbar interconnect, facilitating high-speed on-chip data transfer. The accelerator handles the majority of the DNN computation, with the option of offloading more exotic operations to the CPU. Additionally, the CPU manages data offloading and onloading and dispatches accelerator operations. The memory system stores the parameters and activations of the DNN, while the C2C interconnect links multiple MINOTAUR chips to form a multi-chip system. Such a multi-chip configuration, consisting of many interconnected MINOTAUR chips, can accommodate the requirements of increasingly larger DNNs, with minimal overhead, giving the impression of a single, large dream chip that would be unfeasible to produce as a single chip using current technology [30].

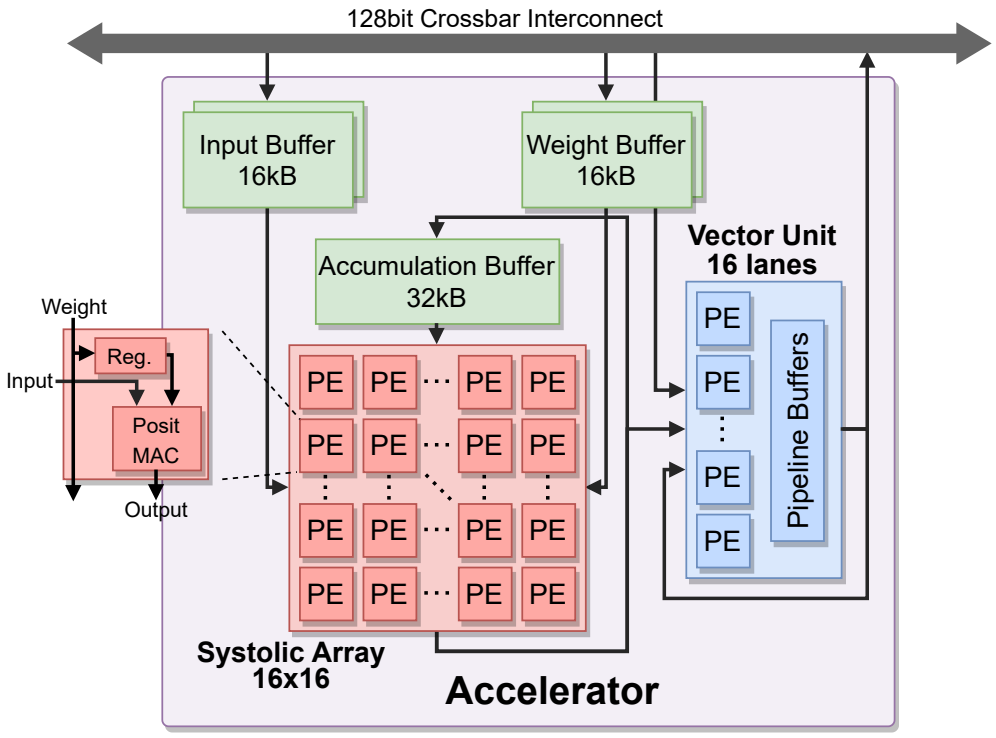

Figure 2.2.: Architecture of the deep learning accelerator in MINOTAUR. It consists of a 16x16 systolic array, a 16-lane pipelined vector processing unit, and three Local Buffers.

2.3. Accelerator

The custom-designed deep learning accelerator lies at the heart of the MINOTAUR SoC, comprising three main components: a 16x16 systolic array, a 16-lane-wide vector processing unit, and three Local Buffers.

Prior to delving into the intricacies of the accelerator's architecture, it is imperative to first establish the tensor notation and computational model that underpin the design principles of the accelerator.

Algorithm 1: Convolutional loop nest

```
1 for b ← 0 to B − 1 do // Batch
2   for k ← 0 to K − 1 do // Output channel
3     for c ← 0 to C − 1 do // Input channel
4       for oy ← 0 to OY − 1 do // Output height
5         for ox ← 0 to OX − 1 do // Output width
6           for fy ← 0 to FY − 1 do // Weight height
7             for fx ← 0 to FX − 1 do // Weight width
8               Out[b, k, oy, ox] ← Out[b, k, oy, ox] + In[b, c, oy + fy, ox +
                  fx] × Weight[k, c, fy, fx]
```

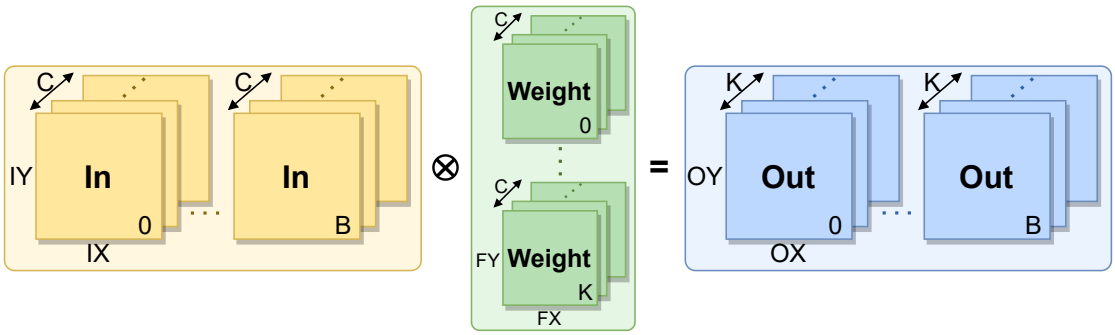

Figure 2.3.: Visual representation of a convolution operation.

2.3.1. Tensor Notation

All common DNNs consist of a series of layers[2], where each layer performs a specific operation on the input data. The most common layer types are convolutional layers, which perform a convolution operation, and fully-connected layers, which either perform a matrix-matrix or matrix-vector multiplication. The notation used to describe the input, output, and weight (also known as filter or kernel) of a layer will be In, Out, and $Weight$, respectively. In the case of the convolutional operator, the input and output of a layer are typically represented as a 4D tensor. For example, a 4D input tensor with dimensions $B \times C \times IY \times IX$ represents a batch of B images, each with C channels, and each channel has a height of IY and a width of IX. The output would be $B \times K \times OY \times OX$, where K is the number of output channels, and OY and OX are the output height and width, respectively. The

[2]The terms *layer*, *node*, and *operator* will be used somewhat interchangeably throughout this thesis. However, in general, we use *node* to refer to a DNN *layer* in a graph, and *operator* when emphasizing the mathematical operation.

Type	Batch	Out Channel	In Channel	Out Height	Out Width	Filter Height	Filter Width
Convolution	B	K	C	OY	OX	FY	FX
Depthwise Conv.	B	1	1	OY	OX	FY	FX
Pointwise Conv.	B	K	C	OY	OX	1	1
Matrix-Matrix	1	K	C	1	OX	1	1
Matrix-Vector	1	K	C	1	1	1	1

Table 2.1.: Convolution operation parameters and its relation to other operations (derived from [27]).

input and output of a layer are typically *connected* via a weight tensor, which is a tensor with dimensions $K \times C \times FY \times FX$, where K is the number of output channels, C is the number of input channels, and FY and FX are the filter height and width, respectively.

Using this notation, one can express a convolution operation using a loop nest consisting of seven nested for loops (see Algorithm 1), each iterating over one of the seven dimensions[3]. Similarly, one can visualize the involved tensors and their dimensions graphically, as depicted in Figure 2.3. While this particular loop nest represents a convolution operation, it is also applicable to other operations such as depthwise convolutions, pointwise convolutions, and fully-connected layers. By setting specific dimensions to 1, one can reduce the convolution operation to a matrix-matrix or matrix-vector multiplication operation (see Table 2.1). For example, setting all but K and C to 1 reduces the convolution operation to a matrix-vector multiplication operation, where the input is a vector and the weight is a matrix. This is because the convolution operation is a generalization of the matrix-matrix and matrix-vector multiplication operation.

The key observation here is that the loop nest depicted in Algorithm 1 is general enough to represent any DNN operation. It is simply a matter of specifying the respective loop boundaries to arrive at the desired operation. Furthermore, we note that the ordering of the loops has no impact on the correctness of the operation. The loop order only affects the order in which the data is read from and written to memory. One can thus reorder the loops for each individual operation to optimize the memory access pattern, i.e., increase both spatial and temporal locality, and thus improve the performance of the operation [43]. This type of loop is also referred to as a *perfectly nested loop* and it forms the basis of all core operations in the MINOTAUR SoC.

[3]For illustrative purposes, we are omitting any padding, stride, or dilation.

Algorithm 2: MINOTAUR loop nest

```
1  for oy1 ← 0 to OY1 − 1 do                                          Memory Level
2      for ox1 ← 0 to OX1 − 1 do
3          for k2 ← 0 to K2 − 1 do
4  ──────────────────────────────────────────────────────────────────
5              for c1 ← 0 to C1 − 1 do                                Buffer Level
6                  for k1 ← 0 to K1 − 1 do
7                      for fy ← 0 to FY − 1 do
8                          for fx ← 0 to FX − 1 do
9                              for oy0 ← 0 to OY0 − 1 do
10                                 for ox0 ← 0 to OX0 − 1 do
11 ──────────────────────────────────────────────────────────────────
12                                 parfor c0 ← 0 to C0 − 1 do         Spatial Level
13                                     parfor k0 ← 0 to K0 − 1 do
14                                         oy ← oy0 + oy1 × OY0
15                                         ox ← ox0 + ox1 × OX0
16                                         c ← c0 + c1 × C0
17                                         k ← k0 + k1 × K0 + k2 × K1 × K0
18                                         Out[1, k, oy, ox] ← Out[1, k, oy, ox] +
                                             In[1, c, oy+fy, ox+fx]×Weight[k, c, fy, fx]
```

2.3.2. Systolic Array

The systolic array in MINOTAUR is responsible for executing the majority of computations required by DNN operations. A systolic array is a parallel computing architecture consisting of a 2D array of PEs, each responsible for performing a MAC operation, receiving inputs from its neighboring PEs, and passing the results to adjacent PEs (see Figure 2.2) [44]. The systolic array is designed to maximize data reuse, minimizing data movement and enhancing energy efficiency. This configuration allows for efficient pipelining and parallel execution of computations, reducing the need for extensive intermediate storage and communication bandwidth.

The loop nest used to feed the systolic array inside MINOTAUR is derived from the standard seven nested loop introduced earlier (see Algorithm 2). It was generated using the Interstellar DSE framework [45] and is optimized for the DNNs expected to be executed on MINOTAUR. It exhibits several key differences from the standard loop nest:

MINOTAUR omits the batch dimension B, primarily because it is designed for real-time inference, where each data sample must be evaluated immediately. Similarly, resource-constrained training at the edge also does not use any batching, as this would increase the required SRAM beyond available resources. Consequently, the input and output tensors used by MINOTAUR reduce to $1 \times C \times IY \times IX$ and $1 \times K \times OY \times OX$, respectively (refer to Algorithm 2).

MINOTAUR employs three distinct loop-level groups: *Memory*, *Buffer*, and *Spatial*. Memory (lines 1-3) contains three loops iterating over $OY1$, $OX1$, and $K2$. Data iterated over by these loops resides in memory. Therefore, for every iteration of these loops, data is read from and written to SRAM or RRAM, necessitating the data to be sent over the crossbar interconnect and making iterations over these loops relatively slow. Buffer (lines 5-10) contains six loops, iterating over $C1$, $K1$, FY, FX, $OY0$, and $OX0$. Data iterated over by these loops resides in the local buffers of the accelerator. As depicted in Figure 2.2, there are three local buffers: the Input Buffer, holding the layer's input data; the Weight Buffer, holding the layer's weight data; and the Accumulation Buffer, holding output data that accumulates before being sent to the vector processing unit for further processing. Spatial (lines 12-13) contains two loops iterating over $C0$ and $K0$. These two loops are *parallel for*-loops, meaning they are unrolled across the two spatial dimensions of the systolic array. As a result, no *actual* iteration over these loops occurs; instead, any computation that is performed in these loops is executed in parallel.

Due to this *splitting* of the K, C, OY, and OX loops, also known as *tiling* or *blocking*, the index computation inside the loop nest must be adjusted from the standard loop nest to account for the three loop-level groups. This new index computation is illustrated in lines 14-17.

Loops within a loop-level group can be rearranged during runtime, allowing the loop nest to be optimized for the specific requirements of the current operation being performed. For instance, the data reuse requirements of a convolution can differ significantly from those of a matrix-matrix multiplication encountered in the multi-head attention mechanism of a Transformer network.

The complete loop nest runtime configuration can be fully characterized using three ordered dictionaries: a three-entry dictionary for the memory level, a six-entry dictionary for the buffer level, and a two-entry dictionary for the spatial level. Each dictionary contains the loop variable name as the key and the number of iterations as the value. For example, a possible configuration for the first convolution layer of ResNet18, which convolves the $1 \times 3 \times 224 \times 224$ input with a $64 \times 3 \times 7 \times 7$ filter to produce a $1 \times 64 \times 112 \times 112$ output, could be as follows:

memory = { 'OY1': 14, 'OX1': 16, 'K2': 4 }
buffer = { 'C1': 1, 'K1': 1, 'FY': 7, 'FX': 7, 'OY0': 8, 'OX0': 7 }
spatial = { 'C0': 3, 'K0': 16 }

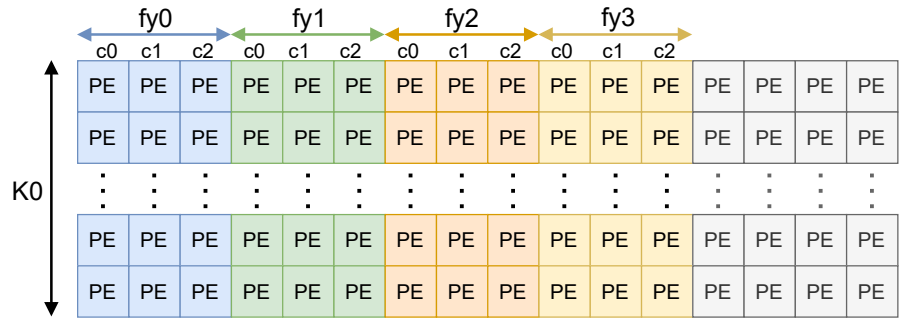

Figure 2.4.: Example of 4x replication used on the first convolutional layer in ResNet. The FY channel is replicated 4x across the spatial $C0$ dimension, leaving only 4 $C0$ channels unused instead of 13.

This loop nest configuration is passed to the accelerator at runtime via the generated C-Struct (see Section 3.5.3), and the accelerator then configures itself accordingly. Notice that the loop nest configuration is not symmetric, i.e., the number of iterations for OY and OX differ in the same loop level. This is perfectly legal and helps to match the reuse requirements to the available buffer space in the accelerator.

Furthermore, one might notice that the spatial loop nest configuration does not fully utilize all available spatial input channels ($C0 = 3$ vs. 16 available). This mismatch in workload parallelism vs. hardware parallelism needs to be accounted for by the compiler by zero-padding the input and weight tensors (see Section 3.3.1). However, this would result in significant performance degradation for this specific layer, as the input and weight tensors would contain a large number of zero values. Therefore, MINOTAUR supports a special *replication* mode, where the accelerator is configured to additionally unroll one of the kernel dimensions across the spatial dimension. This use of replication is depicted in Figure 2.4, where the FY channel is replicated 4x across the spatial $C0$ dimension[4]. The need for replication must be detected by the compiler, and the compiler must generate the correct loop nest configuration for the accelerator.

2.3.3. Vector Processing Unit

Apart from the primary layer types, there exist several other layers, including pooling layers, activation layers, and normalization layers. These layers exhibit considerable heterogeneity and involve orders of magnitude fewer operations compared to convolution and matrix multiplication operations. As a result, they are offloaded to a dedicated vector processing

[4]The reason for 4x replication, and not the theoretically possible 5x, lies in the fact that non-power-of-two replication factors significantly complicate the address computation and are therefore not supported by the hardware.

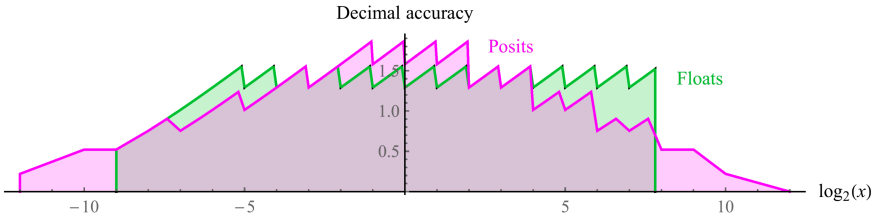

Figure 2.5.: Decimal accuracy comparison of floating-point and Posit, highlighting the greater dynamic range of Posit [41].

unit (refer to Figure 2.2). The vector unit in MINOTAUR comprises 16 lanes and can be conceptualized as a *1D slice* of the systolic array. This compact design enables the vector unit to support a greater number of more complex operations. For instance, the vector unit in MINOTAUR accommodates a broad spectrum of activation functions, such as ReLU, Softmax, and Tanh. Furthermore, the vector unit is capable of executing *layer normalization* operations, a normalization technique employed in models such as MobileBERT as a resource-efficient alternative to batch normalization [46], as well as bias addition and other element-wise operations.

Internally, the vector unit is structured as a *configurable pipelined datapath*, meaning that multiple computational units can be interconnected to create a single, extended pipeline. This configuration enables the vector unit to perform intricate operations while utilizing only a few simple compute units, thereby minimizing area overhead and power consumption. For instance, the vector unit can be configured to perform a fused bias, residual, and ReLU operation, a common operation found in residual networks [2], by chaining together two *add* units, and one *maximum* unit.

2.3.4. Datatype

The accelerator, encompassing both the systolic array and the vector processing unit, utilizes a novel floating-point-like data type called *Posit* [40, 41]. Designed to deliver superior accuracy and resource efficiency in comparison to conventional floating-point or fixed-point implementations, the Posit data type uses a configurable number of bits for the exponent and mantissa representation. This leads to a higher dynamic range and a *bell-curve*-like accuracy distribution, which more closely aligns with the distribution found in DNN workloads. This is demonstrated in Figure 2.5, where the Posit data type is contrasted with traditional floating-point. Additionally, the Posit data type can be implemented more efficiently in terms of hardware overhead compared to traditional floating-point functional units.

In MINOTAUR, the Posit size for activations and weights is fixed at 8 bits, with 1 bit reserved for the sign and 1 bit for the exponent (referred to as *es*)[5]. Internally, both the systolic array and the vector unit accumulate results utilizing a 16-bit Posit data type, which is converted back to 8 bits before being written to memory.

Employing an 8-bit data type would typically necessitate meticulous tuning and QAT (Quantization Aware Training) to achieve reasonable accuracy. Although this implementation still benefits from QAT, it attains remarkable accuracy even in its absence. This is attributed to the Posit data type's scaling behavior, which resembles a considerably *wider* floating-point data type, consequently delivering better performance than comparable 8-bit fixed or floating-point data types on deep learning workloads [47]. This makes Posit an ideal choice for the MINOTAUR accelerator, as it allows for efficient implementation of deep learning models while maintaining high accuracy, even in resource-constrained environments.

2.4. Memory System

As previously discussed in Section 2.3, the MINOTAUR memory hierarchy comprises three levels: 1) on-chip memory in the form of SRAM and RRAM, 2) buffers and caches, 3) registers local to the accelerator's processing elements and the CPU. The program's instructions are stored in a 32 KB instruction RRAM (see Figure 2.1), while the program stack and heap reside in SRAM. The word size employed in MINOTAUR's memory system corresponds to the crossbar interconnect's size, which is 128-bit or 16 bytes. This configuration also aligns with the spatial data width of the systolic array (i.e., 16 using 8-bit Posit). Although this is an ideal system-wide configuration, MINOTAUR's memory system is highly configurable and can be adapted to other word sizes. Similarly, the number of RRAM and SRAM banks can be adjusted to any desired quantity.

Given that the memory hierarchy associated with the CPU is relatively standard, we will not delve into further details here. Instead, we will concentrate on the memory directly involved in inference and training of DNNs, i.e., the accelerator's memory hierarchy. We will place particular emphasis on the RRAM and SRAM employed in the accelerator, as they represent the primary sources of power consumption during operation.

However, since both MINOTAUR's RRAM and SRAM are implemented using proprietary IP (Intellectual Property) provided by TSMC (Taiwan Semiconductor Manufacturing Company) and are subject to NDA (Non-Disclosure Agreement), we can only offer a rough estimate of their characteristics. Consequently, we must approximate the figures for bandwidth, latency, and, particularly power consumption. Nevertheless, this limitation does not pose a problem, as all concepts and ideas presented in this section are applicable to any comparable RRAM

[5]The following blog post offers a concise yet accessible introduction to the Posit data type and its arithmetic: `https://www.johndcook.com/blog/2018/04/11/anatomy-of-a-posit-number/`.

and SRAM implementation [36–38]. It is worth noting that the numbers presented in this section, the results discussed in Chapter 4, and numbers found in future MINOTAUR publications may potentially differ from one another, as the RRAM and SRAM IP is still under development.

2.4.1. RRAM

MINOTAUR incorporates 12 MB of non-volatile storage in the form of on-chip RRAM, utilized for storing the weights and biases of the DNNs. The RRAM is organized into 12 banks, each with a capacity of 1 MB (see Figure 2.1). Internally, each bank is divided into four macros, each with a capacity of 256 KB. However, from a software perspective, the RRAM is only accessible at the bank level.

Each RRAM bank can be individually power-gated through software-controlled registers, exhibiting a negligible number of shutdown and wake cycles. Owing to its non-volatile nature, RRAM banks can be powered down without losing the information stored within their cells. The compiler is responsible for determining which banks to power up and which to power down during memory planning step (see Section 3.5.2). This fine-grained power gating enables aggressive optimization of the power consumption of the RRAM by analyzing the current workload and its memory requirements.

In addition to bank-level power-gating, MINOTAUR's RRAM supports a number of bandwidth configurations. Reading one word from a single RRAM takes 4 cycles. To hide this latency, multiple banks can be combined to form a *super bank*. MINOTAUR supports super bank configurations consisting of 1, 2, or 4 banks. The resulting access latencies are 4, 2, or 1 cycles/word, respectively. These three modes are referred to as *Single*, *Dual*, and *Quad* (see Figure 2.6). Super banks are runtime-configurable and controlled through a software-programmable register. This allows the accelerator to access the RRAM at a maximum continuous bandwidth of 1 word per cycle, or 16 bytes per cycle (128bit). However, not every DNN operator requires such high bandwidth. Consequently, it is the compiler's responsibility to configure the accelerator to match the bandwidth mode to the bandwidth required by the current operator. This allows the compiler to optimize the power consumption of the RRAM based on the current workload.

Furthermore, each RRAM bank is equipped with an *offset* register, facilitating access to the RRAM at an arbitrary offset. This enables non-sequential accesses to consecutive super banks, permitting the compiler to optimize the memory access pattern of the DNN operators across multiple super banks. The importance of this feature will be elucidated during the discussion of RRAM Bandwidth Aware memory planning in Section 3.5.2.

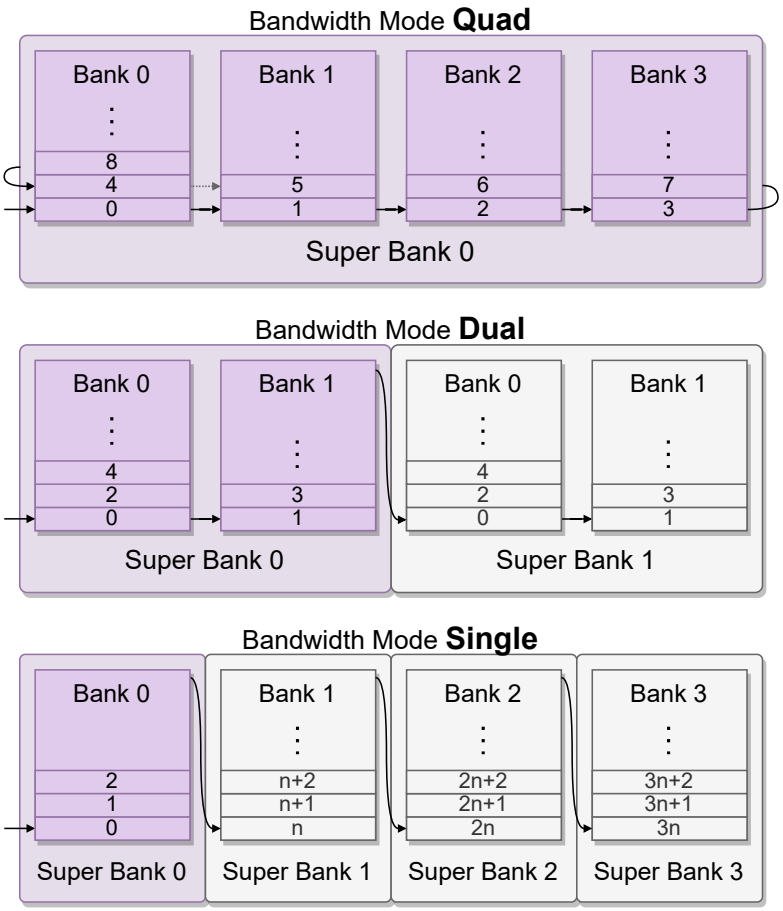

Figure 2.6.: RRAM supports three different bandwidth modes: Single, Dual, and Quad. They combine either 1, 2, or 4 banks into a super bank. The numbers and arrows indicate the order in which the words are read. Greyed out banks are powered down.

2.4.2. SRAM

Similar to the RRAM, MINOTAUR features 2 MB of volatile storage in the form of on-chip SRAM, which is used to store the inputs, outputs, and activations of the DNNs. The SRAM is organized into 8 banks, each with 256 KB of capacity. Each bank is internally divided into 2 macros, each 128 KB in size. The SRAM is only accessible at the bank level and can be individually power-gated through software-controlled registers.

In contrast to RRAM, SRAM is volatile and thus loses its contents when powered down. However, it is much faster than RRAM, as each bank can be accessed in a single cycle. Therefore, there is no need for SRAM to support different bandwidth modes.

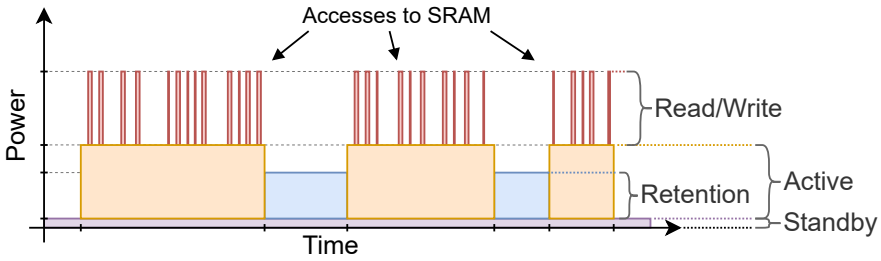

Figure 2.7.: Power breakdown of the SRAM. Standby power is negligible, active and retention power are the static power consumption, while reads and writes constitute the dynamic power consumption.

Due to its volatile nature, power gating inactive banks whose information is required at a later point in time is not possible. Instead, each SRAM bank can be put into a low-power mode, which reduces the power consumption of the SRAM without losing its contents. This is achieved by powering down the SRAM control logic, responsible for accessing the SRAM cells, while keeping the SRAM itself powered. This mode will be referred to as *Retention*[6].

A power consumption breakdown of SRAM is shown in Figure 2.7. SRAM power is composed of two main parts: static and dynamic power. Static power refers to the power consumed by the SRAM when powered on (i.e., not being accessed). Dynamic power consumption refers to the additional power consumed by the SRAM when being accessed, such as during read or write operations. The energy consumed can be determined by integrating the total power consumption over time.

Similar to RRAM power-gating, SRAM power gating is managed by the compiler during memory planning, as detailed in Section 3.5.2. This approach allows the compiler to optimize the power consumption of the SRAM based on the current access patterns of the DNN operators.

2.4.3. Local Buffers

As outlined in Section 2.3, the accelerator incorporates a set of local buffers designed to maintain data in close proximity to the accelerator for low-latency access. The 16 KB input and weight buffers are double-buffered, enabling concurrent read and write operations from the accelerator and the crossbar interconnect, respectively. The 32 KB accumulation buffer

[6]This terminology differs from the terminology used in SRAM literature, where *Retention* refers to the power-down of the SRAM itself, and the here described *Retention* is actually termed *Periphery Power Down* [48]. However, from a compiler viewpoint, the *Retention* mode is analogous to the *Periphery Power Down* mode and is used instead due to its more intuitive nomeclature.

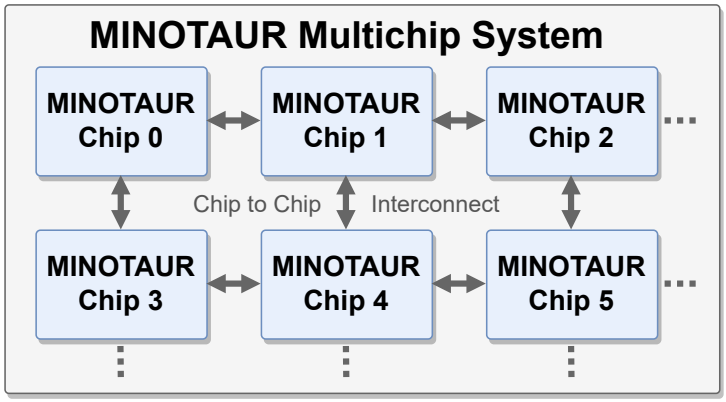

Figure 2.8.: MINOTAUR multi-chip system in a 2D mesh topology.

is employed to accumulate the results of the MAC operations prior to them being forwarded to the vector unit for additional processing. A separate output buffer is unnecessary, as the vector unit's outputs are directly written to the SRAM via the crossbar interconnect.

Moreover, the buffers lack a direct control mechanism and are instead implicitly controlled by the accelerator loop nest. Consequently, it falls upon the compiler to ensure that the cumulative data residing in the buffer loop level does not surpass the buffers' capacity. During loop tiling and loop ordering, it is imperative to ascertain that the buffers do not become overfilled. This is elaborated upon in Section 3.3.2.

2.5. Chip-to-Chip Interconnect

To facilitate communication between chips, MINOTAUR incorporates a C2C (Chip-to-Chip) interconnect. This interconnect is a low-power, low-bandwidth, and fully digital communication protocol. Each MINOTAUR SoC is equipped with four interfaces (north, east, south, and west), which are all connected in a ring topology around the chip (refer to Figure 2.2). Although each interface can transfer 16 bits per cycle, transmitting one message (16 bytes) across the C2C interconnect necessitates at least 13 cycles due to address overhead and local on-chip transfers to and from the interface. While the interface can be employed to transfer data from external sensors, such as cameras and microphones, its primary use case involves data transfer between multiple SoCs. This interconnection enables SoCs to be linked in a multi-chip configuration (see Figure 2.8), allowing for the deployment of arbitrarily large deep learning models with minimal overhead.

The C2C software interface consists of a set of registers that are used to configure and initiate transfers. The general procedure is as follows:

1. Set the destination in the *Destination* register (one of the four interfaces).
2. Write the source address, local to the source chip's address space, into the *Source* register.
3. Write the destination address, local to the destination chip's address space, into the *Destination* register.
4. Write the package size into the *Size* register.
5. Initiate the transfer by writing into the *Command* register.

Following these steps, the C2C interface will transfer the specified amount of data from the source address in the local address space to the destination address in the remote address space using an independent state machine which operators analogous to a DMA (Direct Memory Access) controller. The transfer is initiated by the source chip, and the destination chip will synchronize its C2C clock and acknowledge the transfer. As soon as the transfer is complete, the destination chip will trigger an interrupt that the chip's runtime uses to synchronize itself. This way, the C2C interface can operate fully asynchronously and does not require any synchronization between the source and destination chip's software. Furthermore, the C2C interface is capable of transferring data while both the accelerator and CPU are in a low-power state. The C2C interconnect even remains fully functional when the chip is powered down, allowing the chip to serve as a gateway to other chips in a multi-chip system while preserving energy in deep sleep. This is achieved by using a separate power domain for the C2C interface, which is only powered on when the C2C interface is in use.

As is evident, the C2C interface is a pure point-to-point interface. It is not capable of communicating with chips that are not directly connected to it. This makes it analogous to Ethernet in the TCP/IP stack. Similarly, it is the task of the MINOTAUR runtime in conjunction with the DNN compiler to determine the optimal communication pattern between the SoCs and plan any necessary transfers accordingly (see Figure 2.8).

2.6. Conclusion

In this chapter, we have introduced the MINOTAUR architecture and its salient features. Owing to its flexible loop-nest configuration, the utilization of Posit data types, and the integration of innovative memory technologies, the MINOTAUR architecture is adept at executing SotA DNNs with minimal power consumption, rendering it an appealing platform for machine learning research. Alongside the innovative fine-grained power gating and adjustable bandwidth modes, the MINOTAUR architecture presents a formidable platform for deploying deep learning models. In the ensuing chapter, we will explore the design and implementation of a software and compiler stack that effectively leverages the full potential of the MINOTAUR architecture's capabilities.

3. PAMA Deep Learning Compiler

This chapter presents a comprehensive examination of the PAMA deep learning compiler. We start by discussing the motivation behind PAMA's development, followed by an in-depth explanation of its internal architecture and core components. Next, an extensive overview of PAMA's Frontend, Core, and Backend is provided. Furthermore, we emphasize the integration of the ZigZag mapping framework into the PAMA compiler. Lastly, we highlight PAMA's role as a verification and testing tool during the early stage development and tape-out of deep learning accelerators, using the MINOTAUR SoC as a case study, as well as its potential trajectory as a compiler for distributed machine learning on resource-constrained multi-chip architectures.

3.1. Background

3.1.1. Related Work

In recent years, a myriad of deep learning compilers and frameworks have emerged as promising solutions to the challenge of efficiently deploying deep learning models on resource-constrained hardware. While a comprehensive exploration of available deep learning compilers is beyond the scope of this thesis, a succinct overview of some of the most prominent ones is presented below.

Deep Learning Compilers

As one of the most mature deep learning compilers, TVM [26] is actively developed by a large community under the Apache Software Foundation. TVM offers extensive support for various hardware backends, including CPUs, GPUs, and specialized accelerators. It also accommodates a wide array of deep learning frameworks such as TensorFlow [49], PyTorch [50], and MXNet [51]. The internal architecture of TVM includes a high-level graph IR (Intermediate Representation) called Relay [52] and a low-level representation for operator scheduling, referred to as TensorIR. One of the distinguishing features of TVM is its capacity to auto-tune the performance of a deep learning model for a specific hardware backend

[53]. This feature has undergone several iterations and has reached a mature state. The fundamental concept behind all auto-tuning approaches involves iteratively optimizing the schedule by measuring the performance of the generated code and adjusting the schedule accordingly. MicroTVM supports devices with limited resources lacking fully-fledged operating systems, thus extending the powerful features of TVM to embedded systems. However, integrating new hardware backends is non-trivial, necessitating changes to multiple components within the TVM stack. Efforts to simplify and streamline this process are underway but remain in the early stages [54].

Glow [55], developed by Meta, is an open-source deep learning compiler that aims to optimize deep learning models for inference on both server and edge devices. Similar to TVM, Glow features high-level and low-level intermediate representations (HIR and LIR), facilitating fine-grained optimizations. While Glow is actively developed, its frontend support, particularly concerning ONNX, is limited. Moreover, Glow's support for resource-constrained hardware has not yet reached maturity, as evidenced by numerous open issues on GitHub [56].

TFLite (TensorFlow Lite) [57], a deep learning framework originating from the TensorFlow project, is designed to run on resource-constrained hardware such as mobile devices and embedded systems. Contrary to the previously discussed compilers, TFLite is a runtime-based library that invokes hand-optimized kernels. TFLM (TensorFlow Lite for Microcontrollers) [49] is an even more resource-aware version tailored to run on microcontrollers. Similar to microTVM, it supports various hardware platforms. Nonetheless, both TFLite and TFLM offer only limited support for hardware acceleration, partly due to their lack of compiler infrastructure. Correspondingly, they do not possess the ability to auto-tune or optimize deep learning models for a specific hardware backend.

Deep Learning Mapping and Exploration Frameworks

In contrast to deep learning compilers, deep learning mapping and exploration frameworks are not intended to generate executable code for a specific hardware backend. Instead, their primary purpose is to provide a high-level interface for early DSE of deep learning accelerators and the optimal mapping of deep learning workloads. The majority of these frameworks are developed by academia and are not actively maintained. When authors decide to open-source their work, it is often cumbersome to use, making the validation of claimed results challenging. The most prominent deep learning mapping and exploration frameworks include Maestro [58, 59], Timeloop and Accelergy [60, 61], and Interstellar [45]. Delving into the details of all these frameworks is beyond the scope of this thesis, and interested readers are directed to the original papers.

A notable exception among these frameworks is ZigZag [27], which is fully open-source [62], well-documented [63], and actively maintained by researchers from KU Leuven. Now in its second generation, ZigZag provides a range of easy-to-use examples. It offers a suite of mapping algorithms and comes bundled with several hardware cost models for a variety of different accelerators. Furthermore, a multi-chip version of ZigZag, named Stream [32], has recently been released. Due to their open-source nature and advanced mapping algorithms, ZigZag and Stream are highly integrable with existing deep learning compilers. Consequently, we chose to incorporate them into the PAMA compiler.

3.1.2. Motivation

Although some of its internal components draw inspiration from earlier deep learning compilers and frameworks, PAMA neither incorporates nor utilizes any existing deep learning compiler. The primary motivations for developing a new compiler are as follows:

1. Utilization of a novel hardware cost model, which is not supported by existing deep learning compilers.
2. Implementation of innovative genetic-algorithm-based memory planning and scheduling algorithms.
3. Integration with deep learning mapping and optimization frameworks, such as ZigZag and Stream.
4. Adoption of a custom datatype, called Posit, unsupported by existing compilers[1].
5. Requirement to access and emit activations from within the network graph, including intermediate tensor results, for verification and testing purposes.
6. Necessity to generate code for a custom runtime in a multi-file format, which would demand extensive modifications to existing deep learning compilers.
7. Imperative need to scale the compiler to accommodate arbitrarily large deep learning models across single and, in particular, multi-chip architectures with multiple accelerators.

While all of these requirements could arguably be integrated into any existing deep learning compiler, the necessary changes were deemed too invasive. Consequently, it was determined that the effort of developing a new compiler would be justified.

[1] While TVM does offer support for custom datatypes, the documentation is sparse, and integration within the compiler's core functionality is limited.

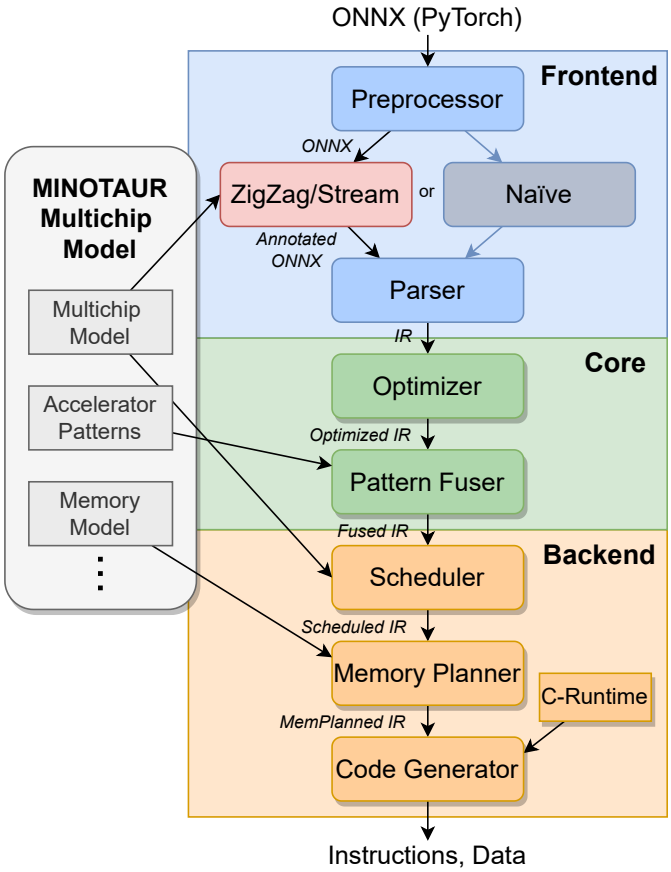

Figure 3.1.: High-level overview of PAMA's modular architecture and the MINOTAUR multi-chip model.

3.2. Architecture Overview

The PAMA deep learning compiler comprises three main components: the Frontend, the Core, and the Backend (see Figure 3.1). The Frontend is responsible for calling into external mapping frameworks for annotations, parsing the input model, and converting it into PAMA's IR. The Core of the compiler is responsible for optimizing the IR and performing pattern matching and operator fusion. The Backend is responsible for scheduling, memory planning, and generating the final code for the accelerator. PAMA's internal components utilize a hardware cost model to query information about the chip and accelerator, such as the systolic array and vector unit dimensions, the sizes of the on-chip memories, and the latency of the supported operators.

All of PAMA's core functionality is implemented in approximately 6,000 lines of Python code and 500 lines of C++ code. This includes around 1,000 lines of code for the Frontend, 2,000 for the Core, and 3,000 for the Backend. Although the codebase is not particularly large, it is still extensive enough to warrant a modular design, hence the three main components. Furthermore, this thesis is not concerned with the implementation details of the compiler but rather with the design decisions and the overall architecture. Therefore, the interested reader is referred to the source code for more details. Nevertheless, the following sections will strive to provide a comprehensive overview of the essential components of the PAMA compiler.

3.3. Frontend

The first of the three main components of the PAMA compiler is the Frontend. The Frontend is primarily responsible for 1) preprocessing, 2) calling into any external mapping frameworks for annotations, 3) parsing the input model, and converting it into the IR (see Figure 3.1).

3.3.1. Preprocessing

Preprocessing, as the initial step of the Frontend, prepares the input model for all subsequent steps of the compiler. Its tasks include 1) running a first pass of optimizations to canonicalize the input model, 2) detecting the type of the input model, and 3) adjusting the operator dimensions to meet the requirements of the Backend. All of these steps must be performed before the model is passed to ZigZag for mapping. This is because ZigZag requires the model to be in an *optimally conditioned* format.

Optimization

The first step of the preprocessing is to optimize and canonicalize the input model to the extent that it can be passed to external mapping tools, such as ZigZag or Stream, for processing. This is achieved by applying a set of transformations to the model, which are implemented as a series of passes. Since all these optimizations are performed before the model is parsed to the internal representation, they need to operate on the ONNX model directly. For this purpose, PAMA leverages the ONNX Graph Surgeon library [64], which provides a set of utilities for manipulating ONNX models. Akin to other compilers, PAMA also implements common graph optimizations, such as constant folding, common sub-expression elimination, dead code elimination, etc. These optimizations are performed partly here,

during the import of the model in the Frontend, and partly during the optimization phase in the Core (see Section 3.4.1). The reasons for optimizing the model are twofold: 1) to improve performance and reduce memory footprint, and 2) to simplify and canonicalize the model, which makes it easier to work with in the following steps. Optimizations are relevant for the annotation phase and therefore need to be performed before the model is passed to the ZigZag annotation tool. The general philosophy of these optimizations is to be as hardware-independent as possible, with the idea being that we want to be able to reuse the optimizations for other hardware targets and future SoCs. A comparison of the different stages of the model and the IR throughout the various phases of the compiler is presented in Appendix C.

Model Type Detection

In the next step, PAMA detects what kind of model it is working with. Currently, supported types are CNN and Transformer. The type is not relevant for any core optimization, mapping, or planning features of the compiler. However, it is essential for the Backend code generator, which needs to know which kind of dataset and data preprocessing it needs to use to generate verification test benches. Since PAMA is currently only used for testing and verification of the MINOTAUR accelerator, the two supported kinds of datasets include image classification for CNNs (e.g., ImageNet [65] and CIFAR [66]) and natural language understanding for Transformers (e.g., subsets of both GLUE [67] and SQuAD [68]). More on the Backend code generation in Section 3.5.

The type is detected by examining several key features, e.g., which operators are present in the model, which input and output tensor dimensions the model has, and what the macro architecture looks like. These features are then used to classify the model into one of the supported types through a simple set of rules.

Zero-Padding

DNNs operators that are to be run on the systolic array and vector unit need to have their spatially unrolled dimensions (C and K) zero-padded to the nearest multiple of the systolic array and vector unit dimension, i.e., multiples of 16 the case of MINOTAUR. While there exist accelerators that can handle arbitrary spatial dimensions by padding the input data during runtime or by turning off unused PEs [69], this comes at the cost of increased latency, area, and power consumption. It was therefore decided to zero-pad the input data for MINOTAUR during compile time instead. The increased tensor size incurred by zero-padding the tensors is negligible compared to the overall size of the tensor data and can thus generally be assumed to be of almost zero cost. Additionally, the vast majority of tensor dimensions already are multiples of the systolic array dimension and thus do not need to be zero-padded.

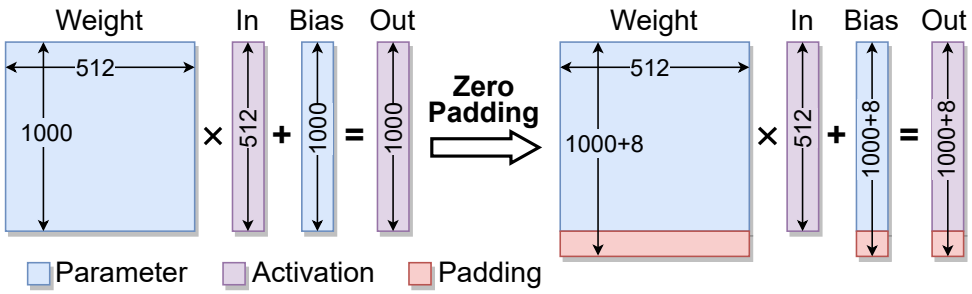

Figure 3.2.: Example of tensor zero-padding on a classification layer in ResNet.

Zero-padding occurs as the final step of the Frontend's preprocessing and is implemented as a custom compiler pass operating on the ONNX graph. It iterates over each node in the model, detects the operator type of the node, and determines whether the node's spatial dimensions need to be zero-padded. The reason for implementing zero-padding this early in the compiler is that zero-padding needs to happen before passing the ONNX model to ZigZag for annotation so that ZigZag (or any other mapping framework) can operate on the zero-padded tensors and thus be able to correctly determine optimal loop-order and tiling parameters.

Figure 3.2 shows an example of zero-padding on a fully-connected layer, as commonly found in ResNet [2]. This layer is the last layer responsible for the classification of the input image into one of the 1000 classes of ImageNet. The flattened input tensor has dimensions 512×1, weight and bias have dimensions 1000×512 and 1000×1, respectively, and the output tensor has dimensions 1000×1. Since fully-connected layers spatially unroll to $1 \times C$ for input, $K \times C$ and $1 \times K$ for weight and bias, and $1 \times K$ for output, all dimensions need to be zero-padded to multiples of 16. While activations only need to have their dimensions adjusted, weights and biases need to additionally have their actual data tensors zero-padded. The process of detecting and zero-padding the tensors is somewhat convoluted and is therefore omitted from this section. The reader is encouraged to look at the source code for more details.

3.3.2. Annotation

After preprocessing, the ONNX model is forwarded to the next step in the compiler's Frontend, the annotation phase. During annotation, the optimal loop order and tiling parameters for each node in the ONNX model are determined. This process is sometimes also referred to as intra-operator scheduling [70]. The loop order and tiling parameters are annotated, i.e., appended, as additional attributes to each node of the ONNX model. The notation used is similar to the one presented in Section 2.3.2.

Annotation can happen in two different ways: either by using an external mapping tool, like ZigZag or Stream, or by using PAMA's internal *naive* annotator.

ZigZag

ZigZag [27] is the preferred and default annotation option due to its advanced optimization and mapping features. PAMA makes use of ZigZag's API and passes the ONNX model, alongside the MINOTAUR cost model, to ZigZag for annotation. ZigZag then internally parses both the ONNX and cost model. Following this, ZigZag performs a number of steps to determine the optimal loop order and tiling parameters for each node in the ONNX model with respect to the resources provided by the cost model. Going into any more details would be out of scope for this thesis. For a more detailed description of ZigZag with respect to PAMA and MINOTAUR, please refer to Appendix B, as well as the ZigZag documentation and the ZigZag paper [27, 71].

Furthermore, ZigZag is able to estimate the performance of the annotated model in terms of latency (clock cycles) and energy consumption (dynamic energy only, refer to Figure 2.7). Additionally, ZigZag calculates how many words are read from RRAM on average per clock cycle. All of this information is then appended to the ONNX model as attributes to the corresponding nodes and returned to PAMA. It is important to note that ZigZag does not perform any optimizations on the ONNX model. It does not alter the graph structure or fuse any operators; it only annotates the model with information.

An example of the attributes added by ZigZag to the ONNX model is shown in Listing 3.2. ZigZag extends the default node attributes, as depicted in Listing 3.1, by adding, amongst others, the attributes `loop_order`, `tiling`, `latency`, `energy`, and `rram_accesses`. These attributes are added to the ONNX model as strings, which PAMA then parses.

Naive Annotation

Apart from ZigZag and other external mapping tools, PAMA offers the alternative of using a *naive* internal annotation mechanism. This option is primarily intended for debugging purposes, as it does not run any optimizations; instead, it merely annotates the model employing a default loop order and simplistic tiling parameters. The loop tiling process is executed by greedily accumulating the buffer loop level dimensions in a round-robin manner until each buffer level is filled. Subsequently, the memory loop level is assigned the remaining dimensions, enabling the naive annotator to consistently generate valid loop-tilings.

```
name: "Conv_5"
op_type: "Conv"
attribute {
  name: "dilations"
  ints: 1
  ints: 1
  type: INTS
}
attribute {
  name: "group"
  i: 1
  type: INT
}
attribute {
  name: "kernel_shape"
  ints: 3
  ints: 3
  type: INTS
}
attribute {
  name: "pads"
  ints: 1
  ints: 1
  ints: 1
  ints: 1
  type: INTS
}
attribute {
  name: "strides"
  ints: 1
  ints: 1
  type: INTS
}
```

Listing 3.1: Default ONNX Attributes

```
attribute {
  name: "spatial"
  strings: "(\'K\',␣16)"
  strings: "(\'C\',␣16)"
  type: STRINGS
}
attribute {
  name: "buffer"
  strings: "(\'OX\',␣4)"
  strings: "(\'OY\',␣8)"
  strings: "(\'K\',␣2)"
  strings: "(\'FX\',␣3)"
  strings: "(\'FY\',␣3)"
  strings: "(\'C\',␣4)"
  type: STRINGS
}
attribute {
  name: "sram"
  strings: "(\'K\',␣2)"
  strings: "(\'OY\',␣7)"
  strings: "(\'OX\',␣14)"
  type: STRINGS
}
attribute {
  name: "data_words_per_cc"
  f: 0.5
  type: FLOAT
}
attribute {
  name: "estimated_latency"
  i: 903551
  type: INT
}
attribute {
  name: "estimated_energy"
  f: 549091.0
  type: FLOAT
}
```

Listing 3.2: Attributes added by ZigZag

Due to the unsophisticated nature of the annotator, its utilization of hardware resources is poor, and its use for serious workloads is not advised. Nevertheless, it may prove valuable for debugging purposes, as it can annotate any ONNX model irrespective of the support provided by the mapping tool.

3.3.3. ONNX Parser

Following the annotation phase, the ONNX model is forwarded to the ONNX parser. The ONNX parser is responsible for parsing the ONNX model and generating the IR used throughout PAMA's stack. PAMA's IR is built on top of a NetworkX DAG (Directed Acyclic Graph) [72]. The IR comprises of node objects, with each node representing a DNN operator, and tensor objects that represent edges, i.e., dependencies between nodes. The NetworkX Python library offers a comprehensive set of utilities and graph algorithms, which PAMA utilizes alongside its custom algorithms for various optimizations and transformations. As the IR is a DAG, topological order traversal of the graph is possible. This traversal represents the default order for any passes operating on the IR in PAMA. An in-depth discussion of parsing the ONNX model and IR is beyond the scope of this text. However, interested readers are encouraged to consult the PAMA source code for further details.

Upon parsing the ONNX model, including all associated attributes, and after assembling the DAG, the IR is transferred to the compiler's core, where optimization, transformation, and canonicalization take place.

3.4. Core

Once the Frontend has parsed the ONNX model and generated the IR, the compiler's Core is invoked. Its responsibilities encompass 1) graph-level optimizations and 2) pattern matching and operator fusion.

3.4.1. Optimization

In contrast to the Frontend optimizations (refer to Section 3.3.1), the Core optimizations in PAMA target the IR rather than the ONNX model. These optimizations are notably more aggressive, as they can significantly modify the graph structure. Such alterations are crucial for canonicalizing the model, ensuring that pattern matching and operator fusion effectively identify and fuse all potential patterns in subsequent steps. It is essential to maintain the

functional correctness of the graph throughout this process. For further details on this topic, readers are directed to the seminal works by Jia et al. [73, 74], which discuss functional correctness of the DNN graph during optimization.

In PAMA, optimizations are implemented as passes, and executed in a user-specified, fixed order. The design of PAMA's passes emphasizes independence to facilitate straightforward reuse and combination. However, the order in which the passes are combined can be critical, as some passes depend on the output of previous ones. This functionality is analogous to the Pass Manager infrastructure in TVM [26].

As an exhaustive list of all optimizations and transformations exceeds the scope of this text, we present a single illustrative example: PyTorch generates multiple operators within the ONNX model that do not directly execute any computation, such as the `Flatten` operator. While these operators are crucial for maintaining model correctness, their functions are implicitly carried out by the MINOTAUR accelerator. To keep the MINOTAUR patterns simple, it is advantageous to eliminate these operators from the model. For instance, in the case of ResNet-style models, the functionality of the `Flatten` operator will be implicitly performed by the preceding `GlobalAvgPool` operator during execution on MINOTAUR. Consequently, the `Flatten` operator can be removed using the `RemoveFlatten` optimization pass in PAMA. An example of the `RemoveFlatten` pass applied to the IR of ResNet18 is provided in Appendix Appendix C.

3.4.2. Pattern Matching and Operator Fusion

The Core component of the compiler is also responsible for performing pattern matching and operator fusion, facilitated by PAMA's PME (Pattern Matching Engine) (refer to Algorithm 3). The primary objective of this process is to identify *explicit* operations in the graph that can be executed consecutively on the accelerator, reducing the number of required memory transfers and improving runtime performance. Upon detecting a fusible pattern, it is replaced with a single fused operator, which subsequently executes in place of the original set of operators. In essence, the PME of PAMA performs just another series of graph substitutions.

Designed to be as generic as possible, the PME aims to be compatible with any DNN model and hardware architecture. Furthermore, it is highly modular, allowing for easy extension to support additional patterns and operators.

The PME is outlined in Algorithm 3, with its input consisting of the IR and the patterns supported by the MINOTAUR accelerator (refer to Listing 3.3). These patterns are specified using a *regular expression-style* syntax, wherein each operator type is denoted by a distinct keyword: `CONV` for convolutions, `RELU` for ReLU, and so on. Pattern matching is conducted

Algorithm 3: PAMA Pattern Matching Engine

Data: Graph G, MINOTAUR Patterns P

Result: Graph G with fused operators

begin
  $pattern\_list \leftarrow [Pattern(G.top\_node)]$
  while $pattern\_list \neq [\,]$ **do**
    $active\_pattern \leftarrow pattern\_list[0]$
    $child\_nodes \leftarrow active\_pattern.child.getChildren()$
    if $Len(child\_nodes) > 1$ **or** $active\_pattern + child\_nodes[0] \notin P$ **then**
      `// Fuse the pattern into the parent nodes`
      $active\_pattern.parent.fuse(active\_pattern)$
      $pattern\_list.remove(active\_pattern)$
      foreach $child\_node \in child\_nodes$ **do**
        $pattern\_list.append(Pattern(child\_node))$
    else
      $active\_pattern.append(child\_nodes[0])$
    $pattern\_list \leftarrow TopologicalSort(pattern\_list)$
  return G

in a top-down manner, meaning that the PME starts at the top of the graph and matches the pattern from the top down. This approach stands in contrast with the bottom-up method employed by TVM's BYOC (Bring Your Own Codegen) pattern matching process [26].

Listing 3.3: MINOTAUR Patterns

```
conv_pattern =
    "(CONV|DENSE)(->ADD)?(->RELU)?(->(MAXPOOL|AVGPOOL))?(->TRANSPOSE)?"
transformer_pattern = (
      "(MATMUL(->BIAS)?)"
    + "|(MATMUL->BIAS(->(RESHAPE|RELU|ADD))?)"
    + "|(MATMUL->BIAS->RESHAPE->TRANSPOSE)"
    + "|(TRANSPOSE(->RESHAPE)?(->MATMUL)?(->(BIAS|DIV))?(->ADD)?)"
    + "|(SOFTMAX)"
    + "|(MUL(->BIAS)?)"
    + "|(GATHER(->DENSE)?)")
regex_pattern =
    ("^(INPUT|OUTPUT|("+conv_pattern+")|("+transformer_pattern+"))$")
```

At the heart of the PME is the *Pattern* object, which holds references to each operator or node within the graph that forms part of the pattern. The algorithm maintains a list of currently *open* patterns comprising of Pattern objects (`pattern_list`). This list is initialized

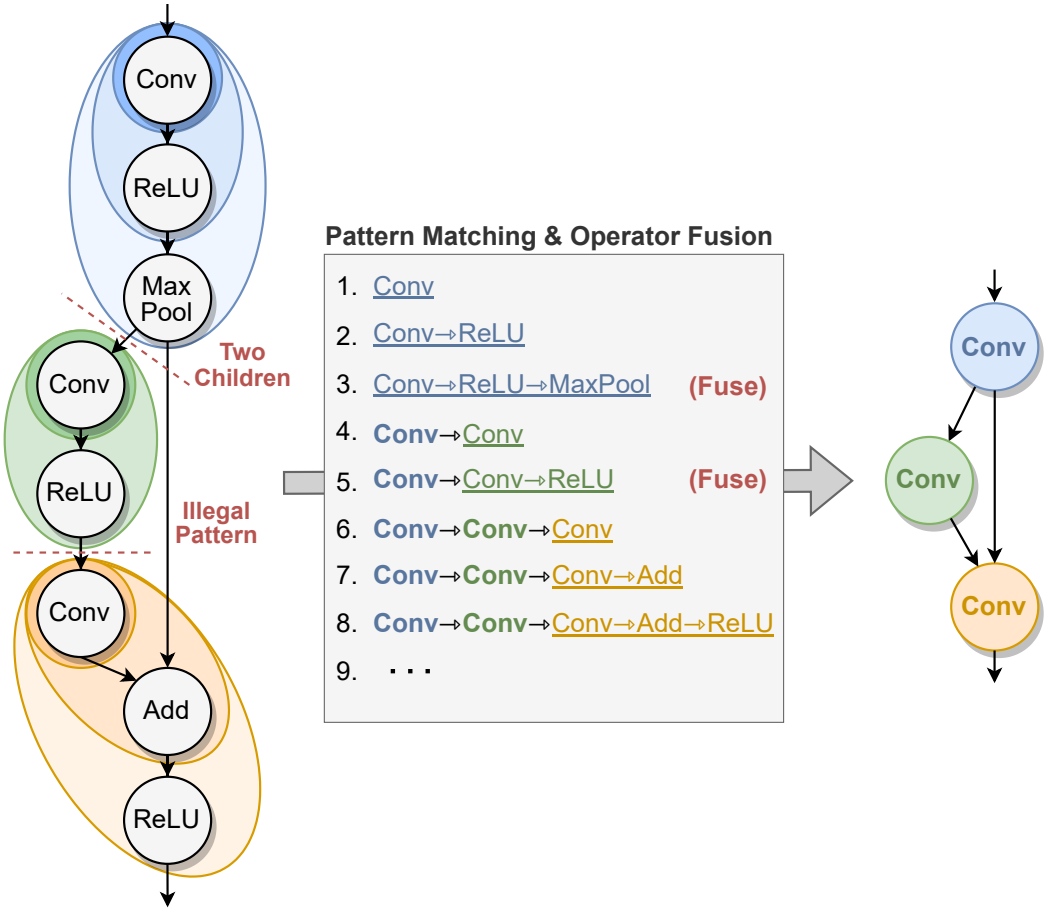

Figure 3.3.: Example of the pattern matching process running on the first part of a ResNet18 model.

with a single Pattern object containing the graph's starting node. Subsequently, the algorithm starts iterating over all child nodes of the current pattern, specifically all child nodes of the pattern's last matched node.

MINOTAUR, along with most existing accelerators, supports only linear patterns without branches. Consequently, if more than one child node exists, the pattern matching process is halted, and the currently matched pattern is fused. Similarly, if appending the current child node to the pattern would result in an unsupported pattern for the accelerator, pattern matching is terminated, and the pattern is fused. Otherwise, the current child node can be appended to the pattern. The patterns within the `pattern_list` are then topologically sorted, and the *shallowest* pattern is chosen next.

Fusing nodes is a non-trivial task, as it necessitates updating the incoming and outgoing edges of all nodes within the pattern. Due to the complexity of this process, we will not delve into the details here. Readers seeking further information are encouraged to consult the PME's source code.

A key advantage of PAMA's PME is the ease with which new patterns can be added to the pattern matching process. This is achieved by simply appending a new pattern to the list of patterns (refer to Listing 3.3). In contrast, TVM's BYOC pattern matching requires the laborious implementation of new pattern matching rules for each new pattern in C++ or Python.

Figure 3.3 depicts an example of the PME operating on the first few nodes of ResNet18. The PME is able to fuse the model's first three nodes into a single `Conv` operator (blue). Fusing more nodes is not possible due to the fact that the `MaxPool` node has two children, which halts the pattern matching process and triggers a *fuse* event. The PME then iterates over the children of the `MaxPool`, attempting to create new Patterns. While this is successful for the `Conv` node, it fails for the `Add` node, as Patterns can start with a `Conv` but not with an `Add` (see Listing 3.3). The pattern PME proceeds to fuse the `Conv` and `ReLU` nodes (green), as well as the `Conv`, `Add`, and `ReLU` operators (orange). It is important to note that there is no constraint on the number of incoming edges for a node in a pattern. The *fuse* mechanism of the PME handles updating the incoming and outgoing edges for all nodes within the pattern.

The complete ResNet18 IR prior to and following the application of the PME is presented in Appendix C. Through the utilization of the PME, the original 49 nodes in ResNet18 are consolidated into 21 fused nodes, signifying a substantial reduction in the number of required reads and writes. In the context of ResNet18, this fusion process can potentially yield up to a 6-fold decrease in memory accesses for each fused pattern[2].

3.5. Backend

The Backend represents the final component among the three main components in the compiler, running after the Core completes pattern matching and operator fusion processes. The Backend is responsible for 1) Scheduling, 2) Memory planning, and 3) Code and data generation. In the subsequent sections, we will delve into each of these tasks in greater detail.

[2] In Figure 3.3 the third (orange) pattern encompasses a `Conv`, with an implicitly fused `Bias`, an `Add`, and a `ReLU` node, culminating in three places where one read, and one write can be saved, i.e., a 6x reduction of memory accesses.

3.5.1. Scheduling

Within the context of the PAMA Backend, scheduling refers to the determination of the execution order of the nodes in the model. PAMA creates the schedule statically, i.e., at compile time, since all the nodes in the model and their interdependencies are known. Consequently, the compiler can perform holistic optimizations that are more aggressive than those attainable in a dynamic scheduling environment, where scheduling decisions are made at runtime [75].

Single-Chip Scheduling

Single-chip scheduling is relatively straightforward, as there is only a single computation occurring at any given moment. When only considering latency and due to the fact that all data needs to be written to main memory between nodes[3], the scheduling problem is reduced to the problem of finding a valid topological ordering of the graph. This is a well-known problem in computer science and can be solved in linear time using any valid topological sort. We will call this the *topological schedule*.

Finding the topological schedule for simple networks, such as AlexNet or ResNet18, is trivial as there is only one valid order in which the nodes can be executed. However, when dealing with more complex models featuring multiple parallel paths through the network, the scheduling problem becomes increasingly challenging. Such paths can also emerge due to depth-first execution or layer splitting (see Appendix B.2). In these instances, the compiler has the opportunity to reorder the nodes in the schedule, enabling optimizations. These optimizations are performed with respect to a specific objective function (also known as a cost function), such as minimizing execution time or energy consumption. Nevertheless, the compiler must consistently ensure that dependencies between nodes are maintained.

It is important to realize that scheduling decisions directly influence the memory footprint of the model. This concept is illustrated in Figure 3.4 using a simplistic synthetic network consisting of five layers (`L0` to `L4`). Each layer produces an output tensor, which becomes the input for the subsequent layer. The critical observation is that the output tensor of `L0` (`T0`) and the output tensor of `L3` (`T3`) are twice the size of the other tensors. As a result, it is beneficial to avoid keeping both of these tensors in memory simultaneously. In the network depicted in Figure 3.4, the compiler has three options for scheduling the network by *shifting* the execution of `L1`. In case A, where the layers are executed in the order L0, L2, L1, L3, L4, the compiler *separates* the execution of `L0` and `L3` to the greatest extent possible, ensuring that `T0` and `T3` are never in memory concurrently. This schedule is optimal for this specific

[3] We assume that keeping activations in local buffers between nodes achieves minimal savings.

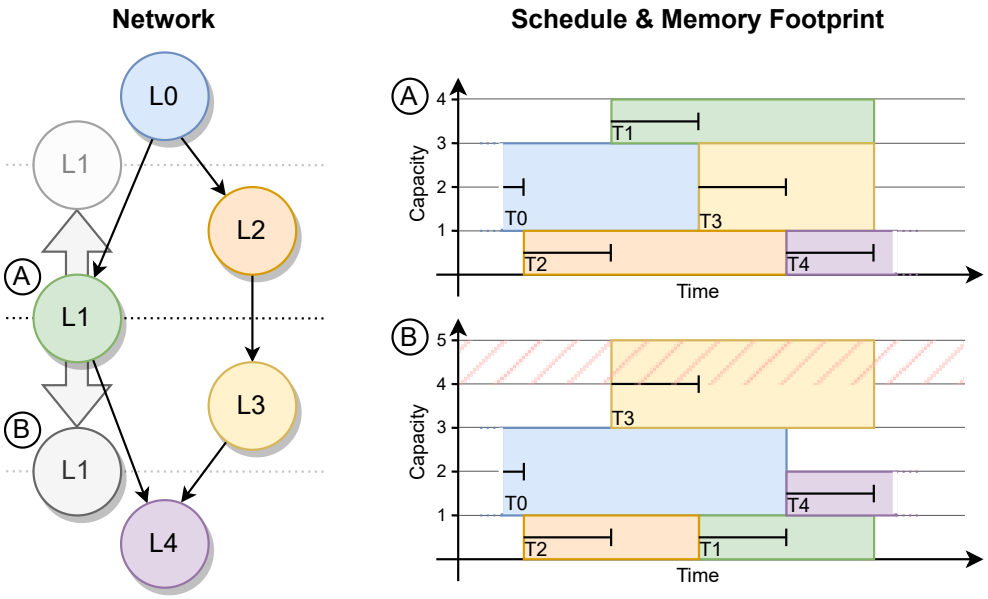

Figure 3.4.: Example illustrating the impact of scheduling on the memory footprint of a model.

network, as it minimizes the memory footprint. In case B, where the layers are executed in the order L0, L2, L3, L1, L4, `T0` and `T3` must be held in memory simultaneously, leading to a 25% increase in peak memory usage (red hatched area in schedule B in Figure 3.4).

Besides peak memory usage, the memory footprint is also a significant determinant of the system's energy consumption. Therefore, the compiler must be capable of scheduling the operators in a way that minimizes the overall memory footprint. Several heuristics attempt to address this problem by greedily shifting nodes according to their input or output tensor sizes. However, they have demonstrated poor performance in determining the optimal schedule with respect to energy constraints.

The above presented example highlights the importance of scheduling with respect to memory footprint and underscores the need for a holistic co-optimized approach to scheduling and memory planning. In the following Section 3.5.2, we will discuss novel algorithms implemented in PAMA to address this problem.

Multi-Chip Scheduling

The multi-chip scheduling problem is inherently linked to the challenges of partitioning and mapping nodes onto a multi-chip system [32]. Due to the complexity of this problem, a detailed discussion will not be provided here. However, once multi-chip scheduling and layer partitioning to each chip are performed, the scheduling problem essentially reduces to a single-chip scheduling problem. Nevertheless, inter-chip dependencies and communication costs must be considered. In order to implement the multi-chip schedule, PAMA needs to integrate tightly with the MINOTAUR runtime, losing the ability to perform scheduling (and memory planning) in isolation. This is because the scheduling decisions made by the compiler must be synchronized with the runtime, which is responsible for executing the schedule in a significantly more complicated environment than in the single-chip case. Arriving at a solution to this problem is beyond the scope of this thesis and will be discussed in future work. For further information, please refer to Section 5.2, as well as Appendix B.2, where we discuss the challenges of scheduling in a multi-chip environment and lay out a holistic approach to the problems of partitioning, scheduling and memory planning.

3.5.2. Memory Planning

Memory in embedded devices constitutes a valuable resource, both directly in terms of available memory size and indirectly through its power consumption. Memory planning, i.e., determining the placement of parameter and activation tensors in the memory, is, therefore, a central component of the PAMA Backend and one of the key contributions of this work. While previous compilers and frameworks focused exclusively on peak memory usage, this work takes a novel holistic approach by optimizing both peak memory usage and energy consumption of the memory simultaneously.

Preliminaries

The here presented memory planning algorithms address the *explicit* memory planning happening in the primary system memories, i.e., RRAM and SRAM. Implicit memory planning, wherein the compiler determines the placement of weights and biases in the memory hierarchy (i.e., local buffers and registers) through the tiling and loop-ordering processes, is described in Section 2.3.2, as well as in Appendix B and Symons et al. [71].

The input to the memory planning stage is the graph coming from scheduling. The order of execution for each node is thus fixed[4], and it is the compiler's task to determine the location of all tensors in the graph. This is achieved by assigning the address fields in every node in the PAMA IR. The following address fields are relevant for the memory planning stage:

- ***input*** $\rightarrow$ all nodes (SRAM)
- ***output*** $\rightarrow$ all nodes (SRAM)
- ***weight*** $\rightarrow$ if node has weights (RRAM) or a 2nd input tensor (SRAM)
- ***bias*** $\rightarrow$ if node has bias (RRAM)
- ***residual*** $\rightarrow$ if node has a residual connection (SRAM)

To elucidate the process of setting these addresses, we examine two representative nodes depicted in Figure 3.5. The first example (left side) is a fused `Conv` node from ResNet, which comprises a `Conv`, a bias `Add`, a `ReLU`, and a residual `Add`. The *input* tensor for the fused `Conv` is stored in SRAM, the weight is stored in RRAM. The *bias* for the initial `Add` is also stored in RRAM. The `ReLU` does not necessitate any additional data. The second `Add` is not a bias operation but a residual addition. As a result, its second input is stored in SRAM, and not in RRAM, as it was generated by a prior operation in the graph. This information is not explicitly stored in either the ONNX representation or PAMA's IR. Consequently, the compiler must deduce this information from the surrounding context during the memory planning phase. A similar challenge arises in transformer models, such as MobileBERT (right side). Here, the inputs to a fused `MatMul` operation may originate from either RRAM, in which case the `MatMul` is a dense layer, or SRAM, indicating that the `MatMul` is part of an attention head [12]. The compiler once again has to infer the source of such operations from the context.

These two examples underscore the complexities encountered beyond the core task of memory planning. The problem becomes progressively more intricate as the number of distinct memories within a system increases.

SRAM Planning

During SRAM planning, the compiler is responsible for determining the optimal placement of each activation tensor in SRAM. To achieve this, a unique address is assigned to each activation tensor generated by a node. This task is made more complex due to the fact that all nodes share the same SRAM. As a result, the compiler must ensure that activation tensors produced by distinct nodes do not overlap in memory, preventing tensors from being

[4] With the exception of the Genetic Co-Optimization memory planning strategy, where scheduling also occurs during the memory planning step

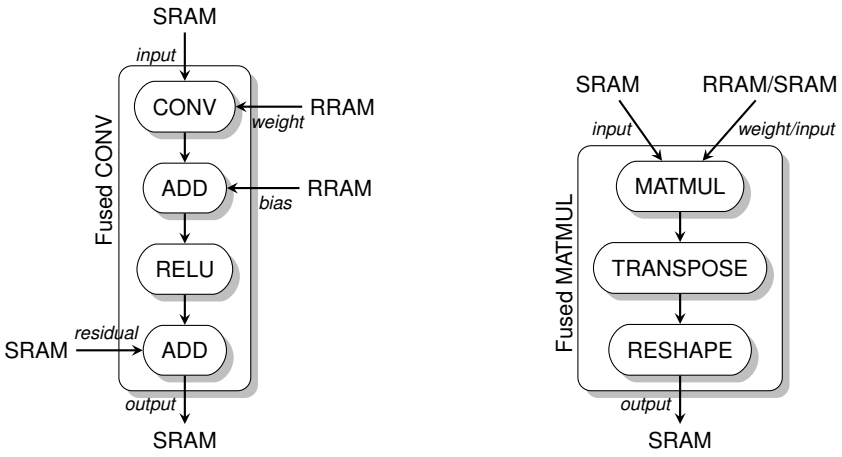

Figure 3.5.: Two examples of non-trivial memory planning based on surrounding context. Left: a fused convolutional layer. Right: a fused matrix multiplication layer.

overwritten before they are consumed. Furthermore, the memory in MINOTAUR is organized as a collection of memory banks rather than a single contiguous block. Each memory bank can be individually power-gated to conserve energy when not in use (see Section 2.4). Therefore, the compiler must not only prevent overwriting but also arrange activations in SRAM in a way that minimizes both power consumption and peak memory usage.

Greedy SRAM Planning The predominant memory planning algorithm for resource-constrained devices today was first proposed by Pisarchyk et al. in [76]. It is employed by both TVM and TFLM as the default option when compiling to resource-constrained systems. This algorithm also serves as the foundation for the greedy algorithms used in PAMA and is detailed in Algorithm 4. Instead of only using a fixed *by size* sorting function, PAMA allows the user to specify the sorting function *F* as a parameter, thus generalizing the greedy sorting heuristic. PAMA features the following cost functions as ready-to-use options:

- *Size*: The size of the tensor in bytes, i.e., spatial size (as per the original implementation).
- *Duration*: The duration of the tensor's lifetime, i.e., temporal size.
- *Area*: The product of the tensor's size and its duration.
- *Schedule*: The order of the nodes in the schedule.

The size of a tensor is the product of the tensor's dimensions and the data type's size: $\textit{size} = \prod_{i=1}^{n} \textit{dim}_i \times \textit{data\_type\_size}$. Since MINOTAUR uses a 1-byte (8-bit) datatype for activations, the tensor size is calculated as follows: $\textit{size} = OX \times OX \times K \times 1$. The tensor's

Algorithm 4: Greedy SRAM Planning

Data: Graph G, SRAM Model M, Cost Function F
Result: Graph G with SRAM addresses, SRAM usage

begin
  // Usage records are triplets: $(N_{prod}, d_{prod}, d_{cons})$
  $usage\_records \leftarrow GenerateUsageRecords(G)$
  $usage\_records \leftarrow Sort(usage\_records, F)$
  // Offset reserves space for stack+heap
  $peak\_usage \leftarrow M.offset$
  $allocated\_usage\_records = \{\}$
  foreach $ur \in usage\_records$ **do**
    $prev\_offset \leftarrow 0$
    $best\_offset \leftarrow None$
    $smallest\_gap \leftarrow \infty$
    foreach $aur \in allocated\_usage\_records$ **do**
      $max\_first\_op \leftarrow Max(aur.d_{prod},\ ur.d_{prod})$
      $min\_last\_op \leftarrow Min(aur.d_{cons},\ ur.d_{cons})$
      if $max\_first\_op \leq min\_last\_op$ **then**
        $gap \leftarrow aur.N_{prod}.output\_addr - prev\_offset$
        if $gap \geq ur.N_{prod}.output\_size$ and $gap < smallest\_gap$ **then**
          $best\_offset \leftarrow prev\_offset$
          $smallest\_gap \leftarrow gap$
        $prev\_offset \leftarrow$
          $Max(prev\_offset,\ aur.N_{prod}.output\_addr + aur.N_{prod}.output\_size)$
      if $best\_offset = None$ **then**
        $best\_offset \leftarrow prev\_offset$
      $ur.N_{prod}.output\_addr \leftarrow best\_offset$
      $peak\_usage \leftarrow Max(peak\_usage,\ best\_offset + ur.N_{prod}.output\_size)$
      if $peak\_usage \geq M.size$ **then**
        return $peak\_usage$
      $allocated\_usage\_records.append(ur)$
      $usage\_records \leftarrow Sort(usage\_records, F)$
  return G, $peak\_usage$

lifetime duration is the difference between the depths of the last consuming node and the producing node: *duration* $= d_{cons} - d_{prod}$. The tensor's area is the product of its size and duration: *area* $=$ *size* $\times$ *duration*. The schedule of a node refers to the order in which the nodes are scheduled in the graph.

The initial step of the greedy algorithm involves generating *usage records* for every tensor in the graph. Each usage record is a triplet consisting of a reference to the producing node, the depth of the producing node (i.e., the time the tensor is created), and the depth of the last consuming node (i.e., the time of the tensor's last use): $(N_{prod}, d_{prod}, d_{cons})$. The depth of a node represents the number of topological generations one has to traverse to get from the input node to the node in question.

Once the usage records are sorted, the algorithm iteratively assigns each tensor to the first available memory address where the tensor is able to *fit*. In this context, fitting entails avoiding both spatial and temporal overlap with any tensor already placed. This check occurs in lines 13-14 of Algorithm 4. If the tensor does not fit, the algorithm assigns it a memory address above the allocated space and increments the *peak_usage*. Upon completing this process, the algorithm assigns the tensors addresses to their corresponding nodes. The algorithm then returns the graph with the assigned memory addresses and the SRAM's peak usage.

Following the allocation of tensors, power-gating of the SRAM banks is performed. During power-gating, the compiler turns off banks that are not in use at the current point in time. It does this by evaluating the memory usage at every *memory state change* in time. In addition, it is able to put banks into retention in case the tensors are not actively accessed but required at a later stage.

An example of SRAM allocation with power gating for ResNet18 is illustrated in Figure 3.6. Black lines inside the tensors indicate that the accelerator is actively computing, i.e., writing to, the specific tensor. As can be seen, the compiler is able to turn off a vast majority of the banks completely and put some of the banks into retention for short periods of time. This leads to a significant reduction in power consumption. The lowest 128 KB are occupied by the system's runtime stack and must remain available at all times.

Although this greedy algorithm is straightforward to implement, it possesses several drawbacks. Firstly, it is oblivious to the banked architecture of the SRAM, which means it naively assigns tensors to memory banks without considering power consumption. Secondly, it cannot leverage potential temporal flexibility in the schedule (see Figure 3.4), resulting in possibly suboptimal allocations.

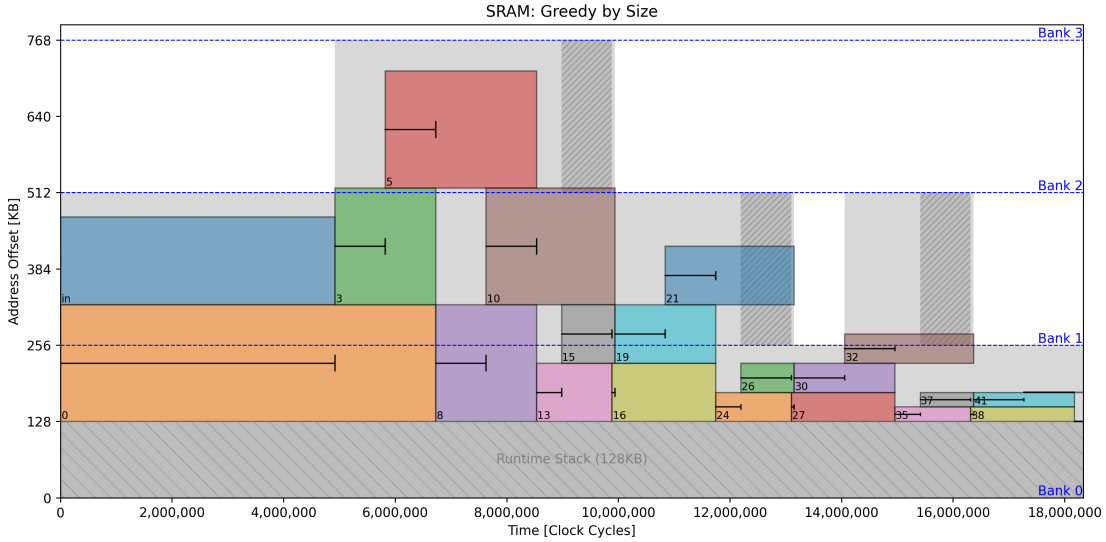

Figure 3.6.: SRAM allocation and power gating of ResNet18 using the MINOTAUR memory model. White: Bank off, Hatched-Grey: Bank in retention, Gray: Bank on. The colors assigned to each tensor are randomly generated and do not carry any deeper meaning.

Genetic SRAM Planning Owing to the intricate nature of power-aware memory planning, it was decided to forgo implementing a power-aware memory planning algorithm *by hand*[5] and instead employ a genetic algorithm to optimize the memory allocation. This approach also facilitates the co-optimization of the schedule and memory allocation.

The genetic algorithm employed in PAMA is implemented using the DEAP (Distributed Evolutionary Algorithms in Python) library [77]. Genetic algorithms are heuristic search algorithms inspired by the process of natural selection and based on the principle of *survival of the fittest*. The fittest individuals are selected for reproduction, while the less fit individuals are discarded. The algorithm implemented for SRAM planning in PAMA is delineated in Figure 3.7. It starts by generating a population of individuals, where each individual encompasses two permutations. The first permutation represents the schedule of the nodes in the graph, and the second permutation determines the SRAM allocation order of usage records, supplanting the sorting function F previously used for sorting. This hybrid approach, which maintains reliance on the greedy allocation algorithm instead of permitting the genetic algorithm to choose arbitrary memory addresses, drastically reduces the search space, thus enabling the genetic algorithm to converge in a reasonable time frame.

[5]This could be achieved by, for instance, formulating the problem as a ILP (Integer Linear Programming) and using a solver to optimize it.

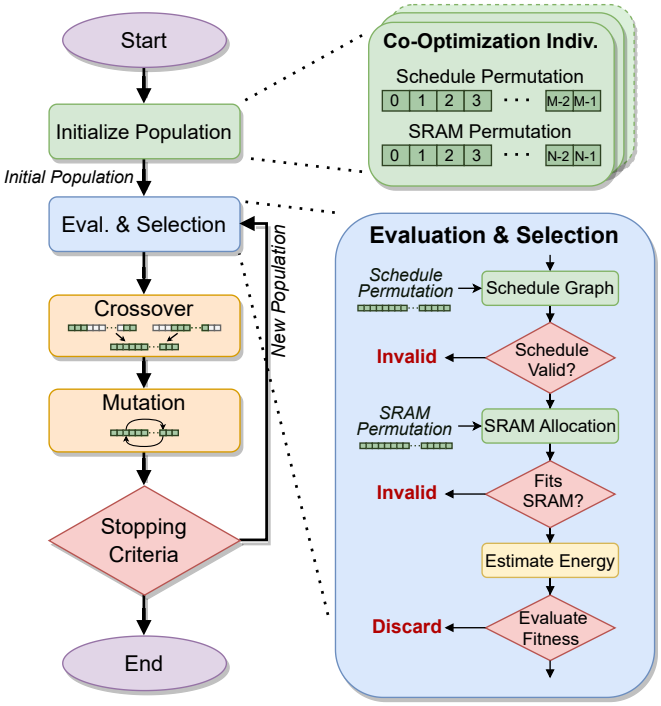

Figure 3.7.: Control flow of the genetic algorithm based memory planning algorithm.

Subsequently, the individuals are evaluated, and the fittest subset is selected for reproduction. The evaluation is conducted by first scheduling the graph using the schedule permutation, followed by allocating the tensors according to the SRAM permutation. The SRAM planning step also incorporates power gating. Utilizing the MINOTAUR memory model, the energy cost of this specific allocation is estimated. The fitness of an individual is the inverse of the peak memory usage and the estimated energy. Based on the weighted sum of these fitness values, the least fit individuals are discarded. The remaining individuals are then paired up and crossed over. Crossover is executed by randomly selecting a crossover point in the permutation and exchanging it between the two individuals. The resulting individuals are then mutated. The mutation is carried out by randomly swapping the permutation. The resulting individuals are added to the population, and the process is reiterated until either a termination condition is met or the desired number of generations has been reached. For a more comprehensive overview of the genetic algorithm internals and its parameters, refer to Appendix A. In addition to the co-optimization variant, where both permutations are optimized, a variant optimizing only the memory allocation order is also implemented. They are referred to as *Genetic* (allocation only) and *Genetic Co-Optimization* (allocation and schedule), respectively.

The property that individuals can be evaluated independently from each other renders the genetic algorithm highly parallelizable, allowing for close to linear speedup on multi-core CPUs. This makes it possible to evaluate a large number of individuals in a short amount of time, which is crucial for the genetic algorithm to converge to a good solution.

A salient issue with the genetic SRAM planning algorithm is that, depending on the constraints, a substantial number of *illegal* individuals are generated. The two pertinent constraints in the case of SRAM planning, as depicted in evaluation and selection in Figure 3.7, involves whether the schedule adheres to all graph dependencies and whether the peak memory usage fits within the SRAM. While these constraints are trivial to verify, generating valid individuals is not. The rationale behind this is that the genetic algorithm is not intrinsically cognizant of the graph dependencies or the SRAM size. Generating a large number of *unwanted* illegal individuals reduces efficiency and can lead to a significant slowdown of evolution. See Figure 4.9 in Section 4.3.2 for a comparison of the performance of the genetic algorithm with and without constraints.

RRAM Planning

Similar to SRAM, RRAM in MINOTAUR is also banked. However, unlike tensor data in SRAM, the parameters stored in RRAM must be retained in memory across multiple inferences, eliminating the possibility for temporal reuse of RRAM. At first glance, this appears to simplify the task of RRAM planning. Nonetheless, RRAM cells exhibit considerably higher power consumption compared to their SRAM counterparts, making the energy consumption of RRAM banks a critical consideration. As a result, distinct bandwidth modes (see Section 2.4.1) were devised, necessitating the development of a novel Bandwidth Aware memory planning algorithm.

PAMA offers three distinct RRAM planning algorithms: Min Size, Min Overlap, and Bandwidth Aware. As their names suggest, Min Size focuses on minimizing the utilized RRAM size, while Min Overlap aims to reduce the overlap of weight and bias tensors with bank boundaries. The Bandwidth Aware algorithm, in contrast, prioritizes power efficiency by allocating parameter tensors in a manner that matches the specific bandwidth requirements of each node. By default, tensors are assigned a bandwidth mode of 4 (Quad), corresponding to an access latency of a single clock cycle and necessitating distribution across four RRAM banks. However, if a layer does not demand a new word during each clock cycle, the bandwidth mode can be decreased. For instance, a layer requiring a new word every two clock cycles can be assigned a bandwidth mode of 2 (Dual), enabling allocation across only two RRAM banks and thereby reducing static RRAM power consumption by 50%.

Min Size Both Min Size and Min Overlap are greedy algorithms. They naively assume all parameters have the highest bandwidth requirement, i.e., Quad. In the case of Min Size, the parameter tensors are allocated to RRAM banks one after another. Due to the fact that all accesses to RRAM need to be word aligned, padding is inserted between addresses where necessary. This process is repeated until all tensors have been allocated. While this guarantees the lowest possible number of RRAM banks is used, it leads to tensors overlapping with bank boundaries, resulting in the need to keep more RRAM banks active than necessary.

Min Overlap In the case of Min Overlap, the parameter tensors are first sorted by size and then allocated to RRAM in a similar fashion to Min Size. However, the allocation is performed in such a way that the overlap with bank boundaries is minimized. This is done by searching over all possible super bank allocations for each tensor and selecting the one with the lowest amount of overlap. The algorithm is essentially an unweighted version of the Bin Packing problem with *overflow* into the next super bank. The bins are the RRAM banks and the items are the tensors. After planning is done, there might be *wasted space* in the final allocation. However, overall power consumption is lower than for Min Size, as the number of tensors overlapping with bank boundaries should be minimal. The Min Overlap strategy for ResNet18 parameters is depicted in Figure 4.1.

Bandwidth Aware Bandwidth Aware RRAM planning aligns the bandwidth provided by the RRAM with the bandwidth demanded by the layer, which further decreases the number of active banks needed at any given moment, thereby reducing power consumption. The general procedure for Bandwidth Aware memory allocation in MINOTAUR is outlined in Algorithm 5. Nonetheless, as with all algorithms presented here, the actual Python implementation is considerably more intricate. The interested reader is once again referred to the source code for a comprehensive description, including any edge cases not addressed in this section.

Since not all parameter bandwidth configurations are allocatable, the Bandwidth Aware algorithm is not guaranteed to find a valid solution. If no solution is found, the algorithm elevates the Bandwidth Mode of the tensor during whose allocation the algorithm failed to the next higher mode. It then resets the addresses and retries (lines 50-55). If the tensor is already in Quad bandwidth mode, the largest tensor in the graph not in Quad mode is selected. This procedure is repeated until a solution is found or all tensors are promoted to Quad mode. In the latter case, the RRAM is too small to allocate the graph using Bandwidth Aware allocation, and the algorithm reverts to Min Size (lines 20-25).

The actual Bandwidth Aware RRAM planning begins in line 2 by sorting the parameter tensors of all nodes by bandwidth mode and size in descending order. This ensures that the *bulkiest* tensors are allocated first. This is crucial because the largest tensors are most likely

Algorithm 5: RRAM Bandwidth Aware

Data: Graph G, RRAM Model M

Result: Graph G with RRAM addresses, RRAM usage

Function PlanRRAM*(G, M)*:

    // Sort nodes by bandwidth mode, then by parameter size
    $sorted\_nodes \leftarrow Sort(G.nodes,\ bw\_mode + \epsilon \times param\_size)$
    $fill\_level \leftarrow \{0, \ldots, M.num\_banks - 1\}$
    foreach $node \in sorted\_nodes$ **do**
        $num\_superbanks \leftarrow M.num\_banks / node.bw\_mode$
        $allocation\_costs \leftarrow \{0, \ldots, num\_superbanks - 1\}$
        $leftover\_space \leftarrow \{0, \ldots, num\_superbanks - 1\}$
        $tensor\_size \leftarrow roundUp(node.weight\_size,\ M.word\_size) + roundUp(node.bias\_size,\ M.word\_size)$
        foreach $superbank\_start \in \{0, \ldots, num\_superbanks - 1\}$ **do**
            $remaining\_tensor\_size \leftarrow tensor\_size$
            foreach $super\_bank \in \{superbank\_start, \ldots, num\_superbanks - 1\}$ **do**
                $allocation\_costs[superbank\_start] \leftarrow allocation\_costs[superbank\_start] + 1$
                $free\_space\_in\_superbank \leftarrow (M.bank\_size - fill\_level[super\_bank]) \times node.bw\_mode$
                $remaining\_tensor\_size \leftarrow remaining\_tensor\_size - free\_space\_in\_superbank$
                if $remaining\_tensor\_size \leq 0$ **then**
                    $leftover\_space[superbank\_start] \leftarrow -remaining\_tensor\_size$
                    break
            else
                $allocation\_costs[superbank\_start] \leftarrow \infty$
        if $All(allocation\_costs = \infty)$ **then**
            if $All(sorted\_nodes.bw\_mode = 4)$ **then**
                $Allocation\ impossible, fall\ back\ to\ Min\ Size...$
            else
                $Promote(node.bw\_mode)$
                // Alternatively, promote node with largest params and bw_mode $\neq 4$
                return -1
        // Get starting bank with lowest allocation cost, tiebreak by lowest leftover space (best fit)
        $min\_cost\_start \leftarrow ArgMin(allocation\_costs + \epsilon \times leftover\_space)$
        $num\_allocations \leftarrow allocation\_costs[min\_cost\_start]$

```
30
31
        // RRAM controller can only handle bank_offset in last allocated
           super bank (NumPy slicing syntax)
32      if Any(fill_level[min_cost_start :
         (num_allocations − 1) × node.bw_mode] > 0) then
33          Promote(node.bw_mode)
34          return −1
35      remaining_tensor_size ← tensor_size
36      foreach superbank ∈ {min_cost_start ×
         node.bw_mode, . . . , M.num_banks − 1} step node.bw_mode do
37          free_space_in_bank ←
             (M.bank_size − fill_level[superbank]) × node.bw_mode
38          this_allocation ← min(free_space_in_bank, remaining_tensor_size)
39          remaining_tensor_size ←
             remaining_tensor_size − free_space_in_bank
40          if node.weight_addr = None then
41              node.weight_addr ←
                 fill_level[superbank] × node.bw_mode + superbank × M.bank_size
42          else
                // (NumPy-like slicing and broadcasting syntax)
43              node.bank_offsets[superbank : superbank + node.bw_mode] ←
                 fill_level[superbank]
44          fill_level[superbank : superbank + node.bw_mode] ← this_allocation
45          if remaining_tensor_size ≤ 0 then
46              node.bias_addr ← fill_level[superbank] × node.bw_mode +
                 superbank × M.bank_size − roundUp(node.bias_size, M.word_size)
47              break
48  rram_used ← Sum(fill_level)
49  return rram_used
50 begin
51  rram_used ← 0
52  repeat
53      Reset(RRAM addresses ∈ G)
54      rram_used ← PlanRRAM(G, M)
55  until rram_used > 0
```

to overlap with bank boundaries, and allocating tensors spanning *more banks* first minimizes the risk of creating *wasted space* (or *empty holes*). Throughout the allocation process, the algorithm keeps track of the `fill_level` for each bank (line 3), which indicates the number of bytes already allocated to each bank. It then iterates over each parameter tensor in the previously sorted list (line 4). For each tensor, it iterates over all super banks, attempting to allocate the tensor starting at the specific super bank (line 9). For each super bank, we record the allocation's cost, i.e., the number of banks that must be active for this particular allocation (line 12). After completing this process for all super banks, we choose the allocation with the lowest cost (line 26). We use the `leftover_space` in the allocation's last bank as a tiebreaker, selecting the allocation with the lowest leftover space (*best fit*). Due to constraints imposed by the RRAM architecture, an allocation spanning multiple super banks may only use offset registers in the allocation's last bank. This is verified by lines 27-34. If the allocation is invalid, we proceed similarly to the explanation above.

Upon determining a the best allocation, the actual allocation process commences. The algorithm iterates through all super banks, starting with the super bank where the allocation begins (line 36). The available space within the current super bank is calculated (line 37), and if it is the first iteration over the super banks, the weight address is assigned (line 41). In the event that it is not the first iteration, the RRAM offset registers address is set (line 43) using the banks' `fill_level`. In both instances, the `fill_level` is updated subsequently. During the final iteration, when the entirety of the tensor has been allocated, the tensor bias address is assigned by subtracting the rounded bias size from the `fill_level` of the last bank (line 46), and the loop is exited. The final step involves summing the `fill_level` of all banks to obtain the total size of the RRAM required for this specific allocation (line 47).

The Bandwidth Aware algorithm applied to the ResNet18 parameters is illustrated in Figure 4.2. By setting all tensors to operate at full bandwidth, i.e., Quad mode, the Bandwidth Aware algorithm becomes functionally equivalent to the Min Overlap algorithm discussed earlier.

Wear Leveling An essential consideration in RRAM planning is long-term wear leveling. Non-volatile memories, such as RRAM, possess limited write endurance before memory cells succumb to permanent failure. In the absence of a wear leveling algorithm in the memory controller, the memory planner has to incorporate one. This is especially important in on-device learning scenarios, where the RRAM is constantly being written to during training. A simplistic wear leveling approach might involve randomizing tensor locations within the RRAM array. Nevertheless, this method would be suboptimal, as it does not consider the spatial locality of the data. More advanced concepts for RRAM wear leveling exist, with some being directly applicable to deep learning applications.

CHIMERA [29, 39] presented ENDURER, a technique that enhances the lifespan of non-volatile memories by filtering writes to frequently written memory words and distributing writes evenly across the memory using a pseudo-random number generator to update address offsets for remapping. ENDURER was implemented inside the RRAM controller to minimize impact on system performance.

At present, neither the compiler nor the RRAM controller implement a wear leveling algorithm. However, the compiler's memory planner has been designed for easy extensibility, rendering the implementation of a wear leveling algorithm, similar to ENDURER, relatively straightforward. This aspect is left for future work.

3.5.3. Code Generation

Once the memory planning has been completed, and all addresses have been determined, the final stage in the Backend process is code generation, also known as *Codegen*. Currently, PAMA features two Codegen backends, both targeting the MINOTAUR testing and verification infrastructure, as described in Section 3.6. This section delves into the implementation details of these two Codegen backends. Due to the modular nature of PAMA, integrating additional Codegen backends is relatively straightforward. A new Codegen needs to inherit from the Codegen abstract base class and implement the necessary methods, which serve as entry points for the new Codegen. For further information, the reader is encouraged to examine the source code of the existing backends.

C-Struct Gen

The primary Codegen Backend is the *C-Struct* Codegen, which generates a C++ header file containing an array of C-Structs and several supplementary data structures. An example of code generated by the C-Struct Codegen for a ResNet18 model can be found in Appendix C.

A comprehensive discussion of all implementation nuances or the specification of each field would exceed the scope of this thesis. So instead, we will provide a high-level overview of the general architecture of the Codegen and provide insights into the most pertinent files and their contents.

The C-Struct Codegen inherits from the Codegen base class `CStructCodeGen(CodeGen)`, which mandates the implementation of several methods, including the `run()` method. This method serves as the callback function when the Codegen is invoked. Its parameters include the IR graph, an output directory for the artifacts, and an integer specifying the number of reference tensor sets to generate. If this number is greater than zero, the original ONNX

model and a path to the directory containing the dataset used to generate the reference tensor are also required. For instance, in the case of a typical CNN, the ImageNet dataset would be used.

The Codegen begins by iterating over the graph, creating a `CStructInstrGen()` object for every node. This object is responsible for generating a single C-Struct. The list of C-Structs is then concatenated and written to a file named `paramsCodeGen.h`. Additional information and data structures are appended to the top and bottom of this file. They include a map associating each C-Struct with files containing the binary weight and bias tensor data used for verification.

Subsequently, the weight and bias tensors for each node are generated. This is achieved by iterating over all nodes in the graph, extracting the weight and bias tensors from the nodes in the IR, and writing them to binary files. Depending on the node type, some data rearrangement may be necessary. For instance, in the case of convolutional nodes, the weight tensor must be transposed to match MINOTAUR's data layout.

In the next step, reference tensors are generated. They allow for the creation of layer-by-layer test benches later in the verification process. For convolutional networks, the raw image data must be read, cropped, and normalized. For transformer networks, the text dataset is loaded, tokenized, and run through the model's embedding layer to generate the embedding outputs. This is due to the limited on-chip memory of MINOTAUR, which necessitates performing this step during compile time. Furthermore, in case the requested number of samples is smaller than the number of samples in the dataset, and random subset is selected. Once the model inputs for each sample have been generated, they are written to individual directories.

Following this, a list of all required input and output tensors and their names is generated. This list of required tensors is used to modify the ONNX model by adding *fake* outputs to each ONNX node. The modified ONNX model is then executed on the ONNX runtime. This is done for each data sample from the dataset of the previous step. Each input/output tensor pair is subsequently saved. This approach allows the testing infrastructure to retrieve each node's inputs and reference outputs and compare them against the accelerator-generated outputs when provided with the same inputs as the ONNX runtime. As a result, the ONNX runtime serves as a *gold model reference* for verifying the accelerator.

However, generating *all* intermediate tensors for *all* samples from the dataset creates a tremendous volume of data[6]. Consequently, intermediate tensors are only generated for the first data sample of the dataset. This approach enables the testing infrastructure to verify the system layer-by-layer while significantly reducing the overall memory footprint.

[6] During the development process, the author of this thesis unintentionally rendered his computer inoperative on multiple occasions by exhausting the capacity of its 2TB SSD with the aforementioned tensor data.

Proto Gen

Managing individual files for each weight, bias, and activation tensor can be cumbersome and inconvenient. An additional code generation method, dubbed *Protogen*, has been developed to address this issue. As the name implies, it generates a single Protocol Buffers file containing a nested hierarchy of Protocol Buffer objects [78]. This Backend's architecture and generation process closely resembles that employed for generating the C-Struct artifacts; thus, we will refrain from delving into the specifics of this particular Codegen.

While the primary advantage of the Protogen Backend lies in the reduction of required files, parsing the generated data is also drastically simplified, as the Protocol Buffer compiler automatically generates parsing code for both Python and C++. Moreover, the Protocol Buffer file sizes are considerably smaller compared to plain ASCII source files, resulting in reduced compilation time, particularly for larger networks. However, a notable drawback of the Protogen approach is its opaque binary format. In contrast, C header files can be more readily inspected for errors in the generated code. Consequently, one could argue that the C-Struct Codegen Backend functions as a *debug* variant, while the Protogen serves as a *production* Backend.

3.6. PAMA Integration

In this section, we explore the integration of PAMA into the MINOTAUR SoC verification and testing infrastructure. One of the primary design objectives was to enable the use of PAMA during the early development and pre-tape-out validation phases of MINOTAUR. Consequently, PAMA has been incorporated as a tool among the existing suite of design, simulation, and testing tools. We will discuss PAMA's role in the testing and verification process in this section.

3.6.1. Verification and Testing

When developing a new SoC, it is crucial to verify the design's correctness and ensure it performs as expected. This verification occurs at various abstraction levels and stages throughout the design process. PAMA serves to verify the accuracy of the complete system as a whole by 1) executing individual layers in a *layer-by-layer* manner and 2) running the entire network in an *end-to-end* fashion.

The verification and testing infrastructure of MINOTAUR is relatively intricate and comprises multiple layers. We will, therefore, only highlight aspects pertinent to PAMA. From a high-level perspective, the following verification and testing steps are executed:

1. *Gold Model* verification against the reference tensors generated by PAMA.
2. *HLS* verification of the hand-written HLS code, i.e., SystemC, against the reference tensors and the Gold Model.
3. *RTL* verification of the synthesized RTL code against the reference tensors and the Gold Model.
4. *Emulator* verification of the same RTL code against the reference tensors and the Gold Model, but on a significantly larger scale.

Compiler like TVM require users to manually implement the kernel schedule, including loop order and loop tiling. When it comes to auto-tuning, some of this tedious work is delegated to a data-driven optimization algorithm that repeatedly executes the kernel on the accelerator and measures its performance. While this approach works well for chips capable of running in *real-time* at *full speed*, it is not feasible for early-stage verification and validation, where only simulations are available, rather than actual hardware. Even when running simulations on tens of cores or at high abstraction levels (Gold Model and HLS), they take orders of magnitude longer than the kernel execution on the taped-out hardware. Furthermore, expensive hardware-backed emulation, available only to select companies or powerful research institutions, is still an order of magnitude slower. Thus, the auto-tuning process could take days, weeks, or even months, which is impractical. However, verifying the design's correctness at an *application level*, in addition to the RTL test bench and unit tests, is crucial for detecting integration errors between different subsystems.

PAMA is capable of generating a kernel schedule optimized for a specific accelerator in a matter of seconds. This is partially thanks to ZigZag, which optimizes the loop order and loop tiling using the same cost model as the rest of the compiler, i.e. without the need of time-consuming auto-tuning. This allows entire networks to be optimized, compiled, and executed automatically in a matter of minutes. This is orders of magnitude faster than an a comparable auto-tuning process and enables the verification of the system's correctness at early stages when the design is still evolving, and changes occur daily. Furthermore, pseudo-randomized brute-force tests can be conducted, expanding test coverage and enhancing confidence in the design. For MINOTAUR, we can execute hundreds of tests, uncovering bugs and integration errors that would have otherwise gone unnoticed. This is especially relevant in cases where the system has reached a complexity level where it is no longer feasible to apply formal verification techniques.

To the best of our knowledge, this automated verification and validation process applied to deep learning accelerator verification is not yet documented in literature. Nonetheless, we can assume that this or similar techniques are already in common use in industry. Thus, it is

not our intention to claim that this is a novel approach. However, we believe that PAMA is a valuable contribution to the field of deep learning accelerator verification and validation, as it enables early development and pre-tape-out validation of deep learning accelerators.

3.6.2. Deployment

Although PAMA has thus far served as a verification and testing tool, it is fully capable of serving as a production compiler, generating code for deployment. To achieve this, PAMA can be configured to generate only instructions and parameter tensors, as activation tensors and Python artifacts are not necessary for deployment. However, the final validation of this deployment scenario is still outstanding, as the MINOTAUR SoC has yet to return from tape-out.

3.7. Conclusion

This chapter provided a thorough examination of the PAMA deep learning compiler, detailing its internal architecture and core components. PAMA's modular design allows for adding new features and optimizations quickly. Thanks to its cost-model-driven scheduling and optimization and the use of ZigZag and Stream, PAMA is able to generate highly efficient code for state-of-the-art deep learning accelerators. In addition, the integration of the ONNX Frontend makes PAMA a versatile tool capable of compiling a wide range of deep learning models. The subsequent chapter will delve into experimental results, offering valuable insights into PAMA's performance and impact on developing advanced deep learning solutions.

4. Experimental Results

The PAMA deep learning compiler was used to optimize and compile a set of representative deep learning workloads targeting the MINOTAUR SoC. These workloads encompass a wide range of neural network architectures, including CNNs and Transformer models. A representative selection of these results is presented in this chapter.

4.1. Performance Metrics

To evaluate the performance of the PAMA compiler, the following performance metrics are used:

- **Memory Size**: The total amount of memory required to store the model's parameters, both in SRAM and RRAM. This is measured in bytes (B).
- **Energy**: The energy consumed to execute a single inference. This is measured in joules (J).
- **Latency**: The time it takes to execute a single inference on the SoC. This is measured in clock cycles (cc).

While PAMA is capable of performing multi-chip mapping, the focus of this thesis is on PAMA's single-chip performance, as no reliable multi-chip mapping results are yet available. Furthermore, it is crucial to note that all power consumption figures presented here are estimates derived from simulation tools since the MINOTAUR SoC is not available in a physical form for measurements.

The data presented in the following was primarily generated by PAMA using the MINOTAUR hardware model and subsequently validated against more accurate estimates procured from cycle-accurate RTL (Register Transfer Level) simulations using VCS [79]. Due to NDA restrictions, it is not possible to fully disclose more detailed figures. However, the available data is sufficient to draw meaningful conclusions, as these estimates are anticipated to closely approximate the values obtained from physical measurements.

Name	Size (MB)	Total Act. (MB)	Accuracy @1/@5	Dataset
ResNet18 [2]	11.7	2.1	56.52 / 79.07	ImageNet
ResNet34 [2]	21.8	3.4	73.31 / 91.42	ImageNet
ResNet50 [2]	25.6	10.7	80.86 / 95.43	ImageNet
ResNet101 [2]	44.5	15.8	81.89 / 95.78	ImageNet
ResNet152 [2]	60.2	22.1	82.28 / 96.00	ImageNet
Inception v2 [85]	27.2	5.2	77.30 / 93.45	ImageNet
GoogLeNet [86]	6.6	4.2	69.78 / 89.53	ImageNet
MobileBERT [46]	25.3	20.2	92.8	SST-2

Table 4.1.: DNN models and datasets used for the experiments. Top 1 / Top 5 accuracy shown for ImageNet.

4.2. Models and Datasets

The models used in the experiments are listed in Table 4.1. The CNN models are trained on the ImageNet dataset [65], and the MobileBERT model is trained on SST-2 [80]. The ImageNet dataset comprises 1.2 million images, divided into 1.28 million training images and 50,000 validation images, while the SST-2 dataset contains 67,349 sentences, split into 52,490 training sentences and 4,859 validation sentences. No dedicated MLP (Multi-Layer Perceptron) models (i.e., models containing only fully connected layers) are used in the experiments, as both the CNN and Transformer models contain fully connected layers. Nevertheless, it could be interesting to explore the performance of *pure* MLP models, including architectures like Autoencoders [81], in the future, especially since recent work has demonstrated that their performance is comparable to CNN models [82].

PAMA provides these DNNs, amongst others, in a collection of ready-to-use ONNX models. This *model zoo* can be used to validate the correctness and performance of both hardware and software stacks. These ONNX models are all exported from PyTorch's Vision Library [50, 83], as well as the Hugging Face Transformers Library [84].

In this study, several model source files have been modified to accommodate the limitations of the current MINOTAUR version, which does not support all operators found in the original models. To address the impact of these modifications, the adapted models undergo retraining to ensure their accuracy remains consistent. Furthermore, we employ a QAT technique that utilizes 8-bit Posit fake quantization during the retraining process. This approach maintains the accuracy of the modified models, with a deviation of approximately 0.5% from their original 32-bit floating-point counterparts.

4.3. Memory Planning

In this section, we focus on the memory planning performance of the PAMA compiler when targeting the MINOTAUR SoC. This is particularly important as PAMA's advanced memory planning capabilities represent a key contribution of this work. We specifically examine the static power consumption of the SRAM and RRAM, as PAMA's memory planning can only reduce static power consumption through power gating of individual memory banks. Dynamic power consumption remains unaffected by the memory planning in PAMA's Backend.

Instead, dynamic power consumption is influenced by the number of memory accesses, which are determined by the graph structure optimized in PAMA's Core stage (see Section 3.4), as well as the loop order and loop tiling managed by ZigZag. ZigZag provides annotations for the estimated number of accesses and corresponding power numbers for each node. This information allows PAMA to generate a comprehensive power estimation for the SoC, presented in Appendix B.

4.3.1. RRAM

We commence by discussing the allocation of ResNet18 tensors across the RRAM banks in MINOTAUR using the Min Overlap memory planning strategy as detailed in Section 3.5.2. The resulting allocation is illustrated in Figure 4.1, where each tensor is assigned the same random color as in the SRAM visualizations. Analogous to Figure 2.6, the banks are depicted with their lowest address at the bottom so that continuous access progresses upward through the bank. In the Min Overlap approach, all tensors are allocated utilizing the Quad bandwidth mode, causing each tensor displayed in Figure 4.1 to span four banks. This allocation proves to be suboptimal since not all tensors require the full bandwidth.

Therefore, we also showcase the allocation of the same ResNet18 tensors using the Bandwidth Aware memory planning strategy in Figure 4.2. This allocation is generated by taking into account the annotated bandwidth requirements from ZigZag, utilizing the Bandwidth Aware allocation algorithm described in Section 3.5.2. As observed, only a small fraction of the tensors necessitate the full bandwidth, resulting in a significantly more efficient allocation than the Min Overlap (or Min Size) allocation.

A comprehensive energy comparison of various RRAM allocation strategies across all models is depicted in Figure 4.3. Where needed, the RRAM was extended with additional banks to accommodate all parameters of the larger networks. While ResNet18 fits into the 12 banks of single MINOTAUR SoC, ResNet152, for example, required a total of 60 RRAM banks. Scaling the RRAM size can be done by simply increasing the number of RRAM banks in the hardware cost model.

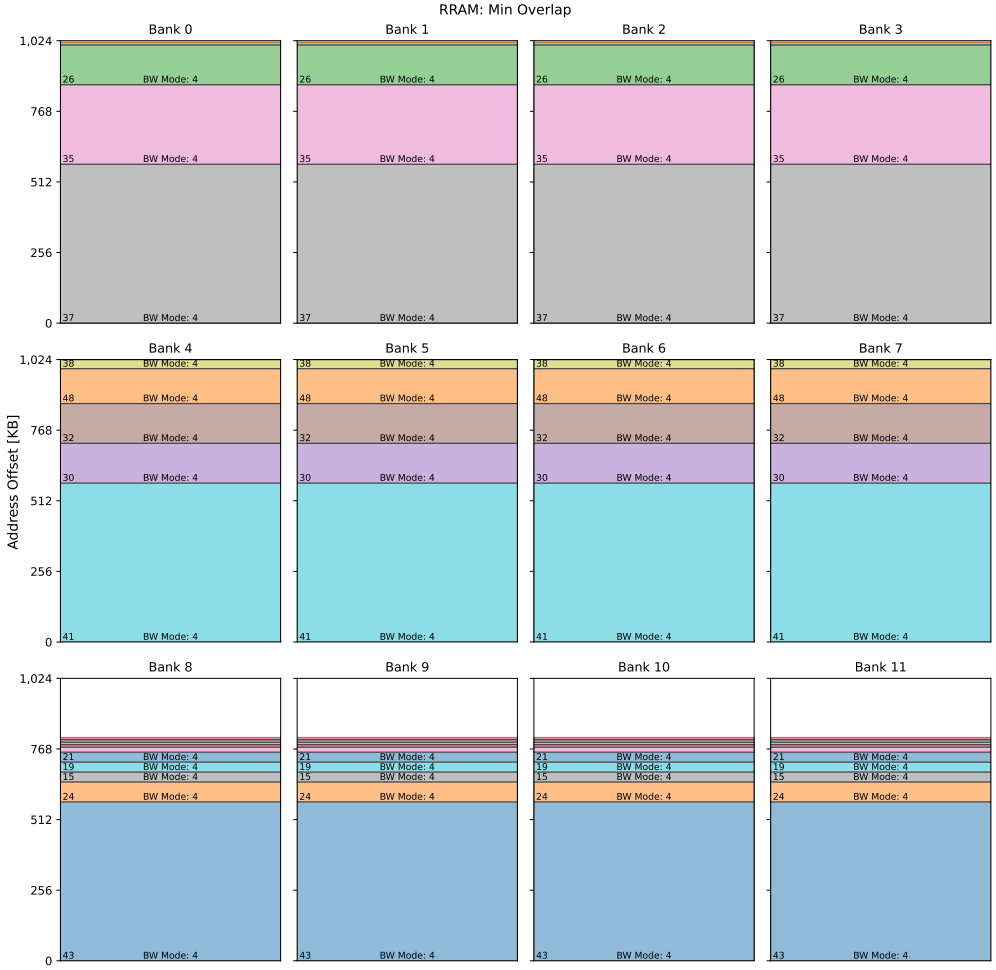

Figure 4.1.: RRAM allocation of ResNet18 parameter tensors using the Min Overlap memory planning strategy.

The absence of RRAM power gating results in the highest energy consumption, an effect that becomes more pronounced for larger networks with an increased number of parameters. This can be attributed to the fact that the total network parameter size grows more rapidly than the size of a single set of parameter tensors that need to be accessed simultaneously. Both the Min Size and Min Overlap strategies are able to reduce energy consumption by a significant margin, but the Bandwidth Aware strategy is able to further reduce the energy consumption by up to 70% compared to the Min Size strategy. The Bandwidth Aware strategy is able to identify parameter tensors that do not require the full bandwidth and can therefore be allocated in a more efficient manner.

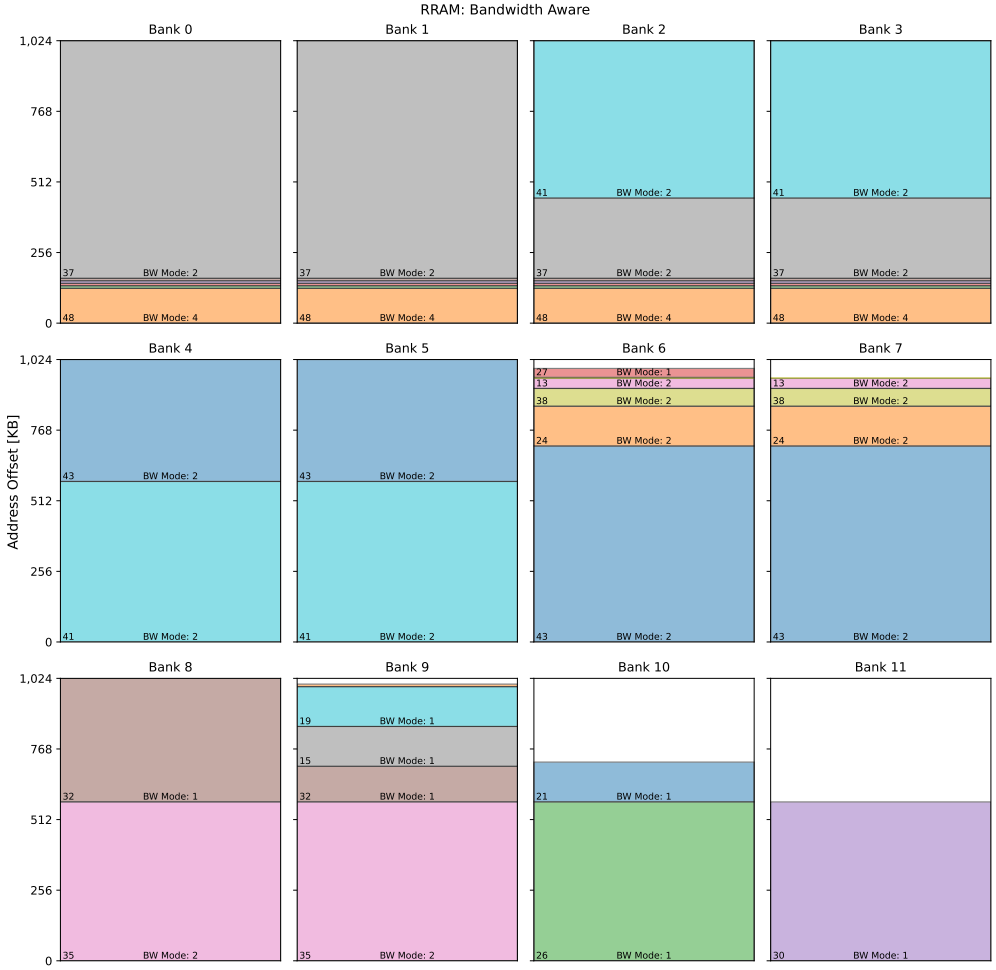

Figure 4.2.: RRAM allocation of ResNet18 parameter tensors using the Bandwidth Aware memory planning strategy.

An interesting exception to this trend is AlexNet, which consists of several large parameter tensors spanning multiple banks. Consequently, the Bandwidth Aware strategy is less effective for AlexNet compared to other models. Nonetheless, LLMs or SotA vision models with structures akin to ResNet and MobileBERT but with greater depth should be able to benefit significantly from power gating and, in particular, from the Bandwidth Aware allocation.

Across all presented models, the presented Bandwidth Aware power gating approach results in 84.68% energy savings compared to the no power gating, with up to 97.35% (37.79x) for larger models like ResNet152. It is important to note that the presented energy savings are achievable without any performance overhead, as the memory bandwidths are tailored to the specific requirements of each DNN layer.

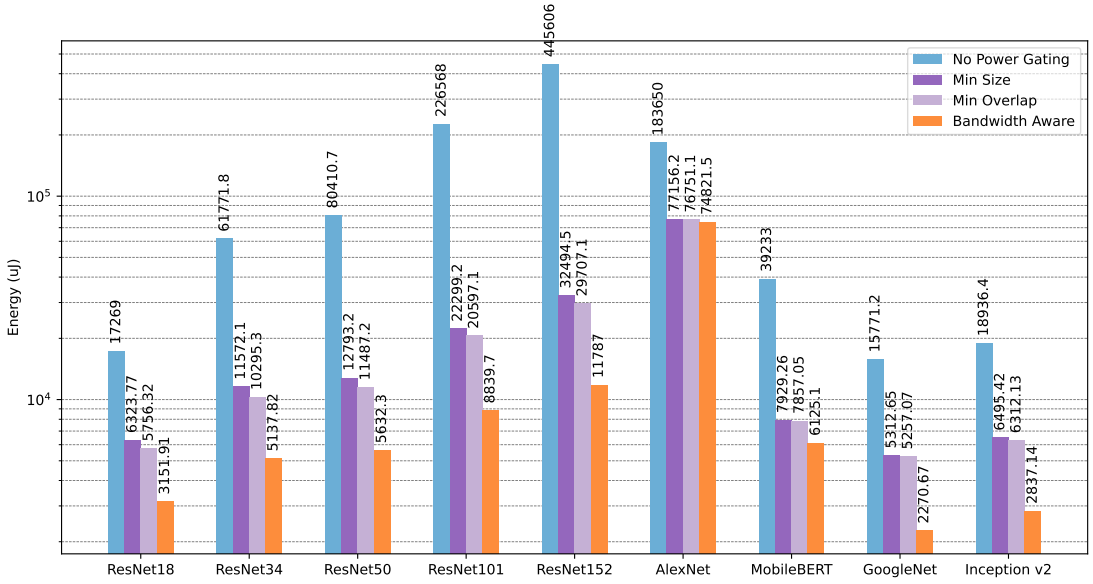

Figure 4.3.: Energy comparison of the different RRAM allocation strategies across various DNNs.

4.3.2. SRAM

Similar to RRAM, SRAM energy consumption also greatly benefits from power gating. This effect is demonstrated in Figure 4.4, where each model's SRAM allocation is generated using no power gating, power gating alone, and power gating in conjunction with retention mode. While the latter is consistently the most efficient, its relative benefit compared to power gating alone is not as pronounced. The average power savings across all models using power gating alone is approximately 69.11%, while the average power savings using power gating + retention is approximately 70.15% (+1.04%).

AlexNet is once again the outlier, as it does not benefit from retention at all. This is due to the graph structure of AlexNet being a *single chain* of layers, meaning that all parameters are accessed in every iteration. Consequently, the retention mode offers no advantage, as no parameter tensors need to be retained between iterations.

In general, the modest energy savings achieved through the retention mode can be attributed to its limited applicability, as it is only beneficial for parameter tensors that are accessed multiple times. However, in the majority of DNNs, this is the case for only a small fraction of tensors. Moreover, the greedy allocation strategy employed in this study fails to arrange the parameter tensors in a manner that maximizes the benefits incurred from

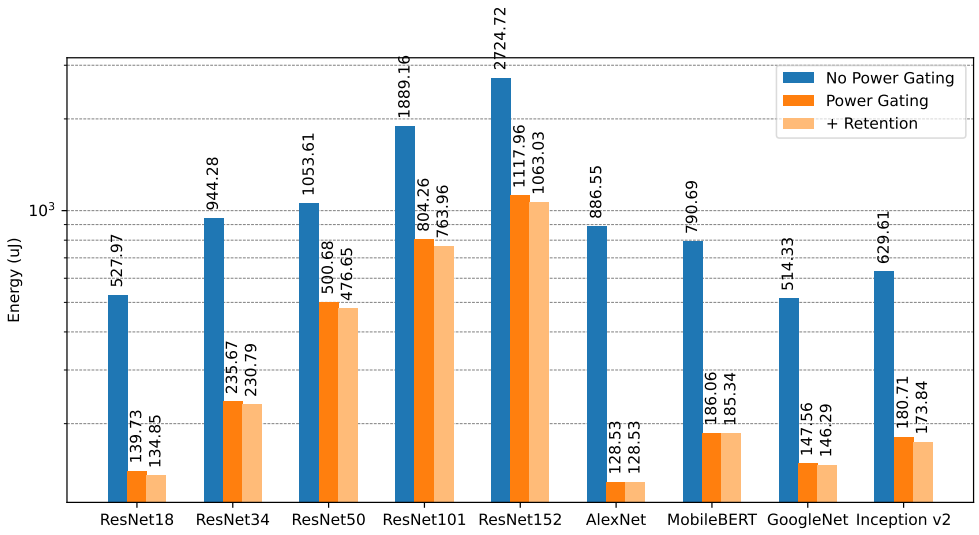

Figure 4.4.: Energy comparison of different SRAM power management approaches, including: no power gating, power gating alone, and power gating and retention. Generated using the Greedy by Size strategy.

retention. This issue is particularly evident in the case of MobileBERT, where the greedy allocation strategy assigns the largest parameter tensors to the first bank, leading to unnecessary power consumption (refer to Figure 4.6).

Drawing upon a similar methodology as the one utilized in the RRAM investigation, we assess and compare various SRAM allocation strategies, as illustrated in Figure 4.5. In particular, we investigate four greedy allocation strategies, denoted by the blue-shaded bars, which are based on the algorithm depicted in Algorithm 4. Additionally, we explore two genetic algorithm-based approaches, represented by the orange-shaded bars. As observed, the algorithm-based allocation consistently surpasses the performance of greedy-based allocation strategies, even when considering relatively simple network structures such as AlexNet or ResNet.

This superior performance can be attributed to the genetic algorithm's ability to iteratively optimize the allocation in a holistic manner, effectively identifying the optimal allocation for each tensor within the context of the entire network's peak size and energy consumption. In contrast, greedy-based allocation strategies lack such a feedback mechanism, rendering them incapable of allocating the parameter tensors in a way that maximizes overall energy savings.

Figure 4.6 and Figure 4.7 emphasize how the Genetic Algorithm can enhance the energy consumption of the SRAM allocation by rearranging the tensors in a more optimal manner compared to the Greedy strategies. Figure 4.6 illustrates the SRAM allocation for a single

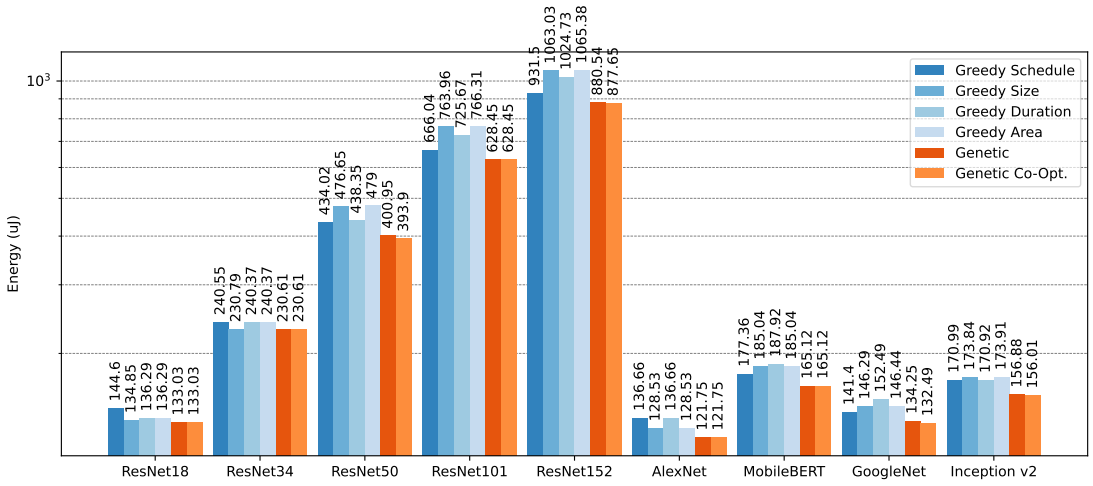

Figure 4.5.: Energy comparison of the different SRAM allocation strategies across all models.

MobileBERT encoder block. The Greedy by Size strategy allocates the *large* blue input tensor, which persists from the beginning to the end, first, resulting in its placement at the *bottom* of the 0th SRAM bank. While this approach of prioritizing large tensors generally yields satisfactory allocation performance, it fails to recognize that this tensor is not required for most of the computation. It remains idle for the majority of the time, only being utilized at the beginning and the end. This is where the Genetic Algorithm significantly improves the allocation. It determines that relocating the input tensor to the top of the stack, where it is allocated in its *own* SRAM bank, allows for the utilization of the retention mode to conserve energy. This simple rearrangement of the tensors results in an approximately 11% energy reduction without incurring any additional overhead.

Similarly, Figure 4.7 depicts the SRAM allocation of ResNet50 parameter tensors using the Greedy by Size strategy as well as the Genetic Algorithm. A scenario akin to the single MobileBERT encoder is observed. However, in this case, the situation is reversed. The Greedy by Size strategy allocates the *small* but *long* input tensor at the top across banks 7 and 8. This visibly leads to a considerable amount of wasted bank space (grey area) which must be kept powered even though it is unused. The Genetic Algorithm rearranges the tensors so that the input tensor is allocated at the bottom of the stack, resulting in significantly less grey area, i.e., less wasted power. Similar behavior is evident towards the end of the computation, where the smaller tensors are arranged more optimally with respect to the number of SRAM banks that need to be powered. The Genetic allocation achieves approximately 18% energy savings compared to the Greedy by Size strategy.

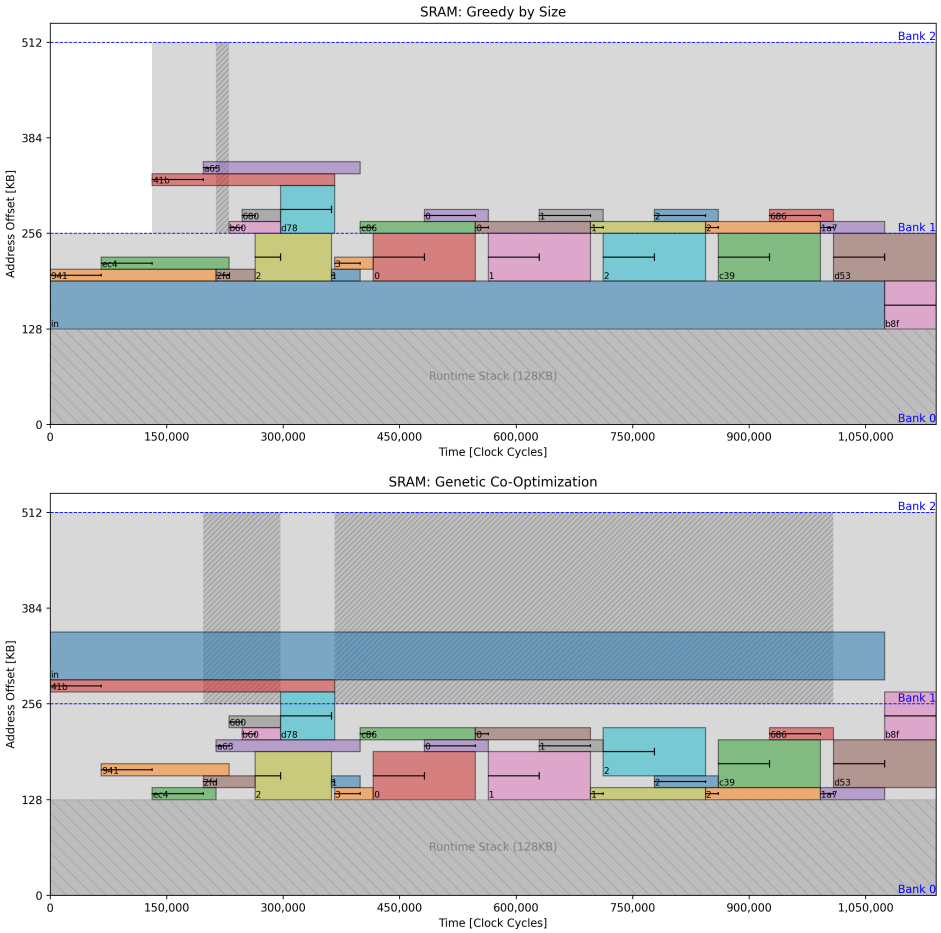

Figure 4.6.: Comparison of the Greedy by Size vs the Genetic Algorithm SRAM allocation for MobileBERT. Moving the input tensor to the top makes it possible to use the retention mode to save energy.

The complexity and varied optimization results of these two examples underscore the challenge of devising a single fixed heuristic capable of accommodating the constraints across all models. The Genetic Algorithm-based approach is capable of adapting to the diverse requirements of different models and consistently finds superior allocations for all tested models, as demonstrated in Figure 4.5.

Besides yielding more optimal allocations, the Genetic Algorithm also offers a Pareto front of design points with respect to energy consumption and peak SRAM usage. Four exemplary Pareto fronts are presented in Figure 4.8. As observed, both Genetic Algorithm approaches significantly outperform all four Greedy strategies. The more intricate a model architecture becomes, the greater the flexibility in tensor rearrangement possibilities and the higher the likelihood that the Genetic Algorithm will discover a superior allocation. While in the case of

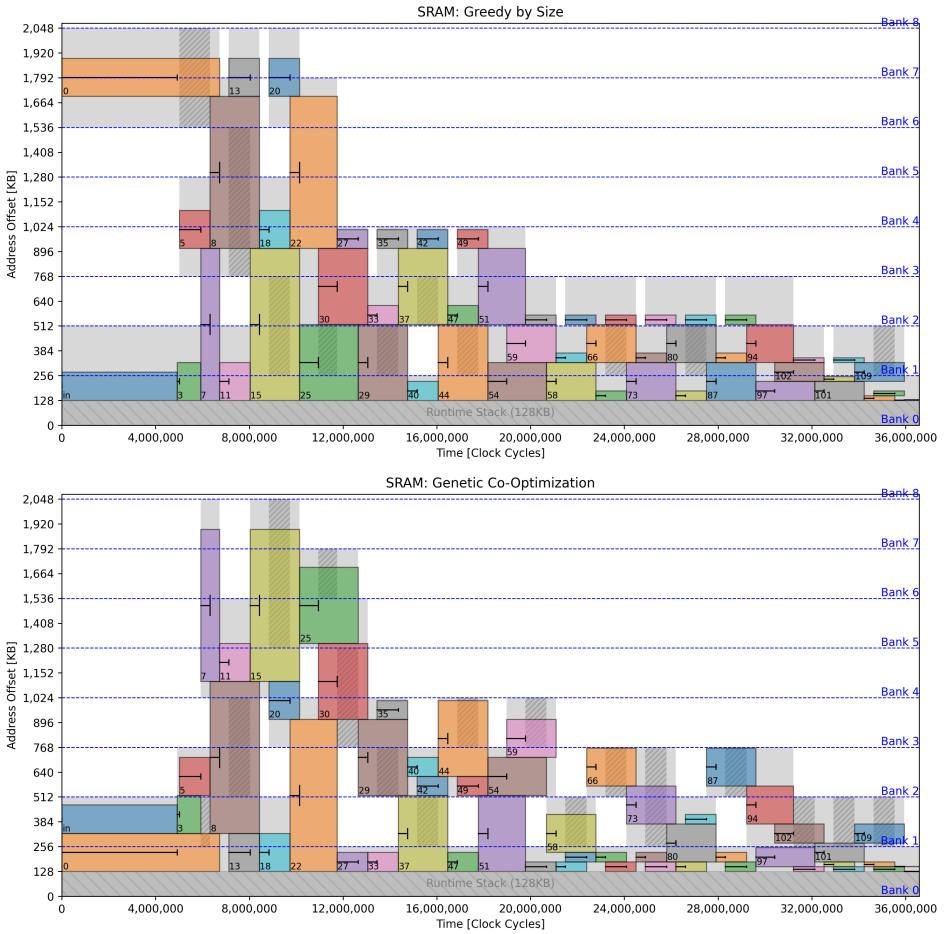

Figure 4.7.: Comparison of the Greedy by Size vs the Genetic Algorithm SRAM allocation for ResNet50.

MobileBERT, the Genetic Algorithm approaches determine a superior allocation in terms of both energy and peak SRAM usage, they are unable to provide more than a single Pareto-optimal design point. This can be attributed to the fact that rearranging the relatively simple tensor configuration depicted in Figure 4.6 does not result in more than a single optimal solution. However, for more complex models, such as Inception v2 and ResNet152, the Genetic Algorithm approaches are able to identify multiple Pareto-optimal design points, which are not possible to discover using the Greedy strategies.

Another intriguing observation is that the Genetic Co-Optimization does not always yield a better energy allocation compared to the *plain* Genetic Algorithm approach. This is evident in the upper-left Pareto front of the Inception v2 model and the lower-right of the ResNet152 model in Figure 4.8. While the Co-Optimization identifies the same or more peak SRAM

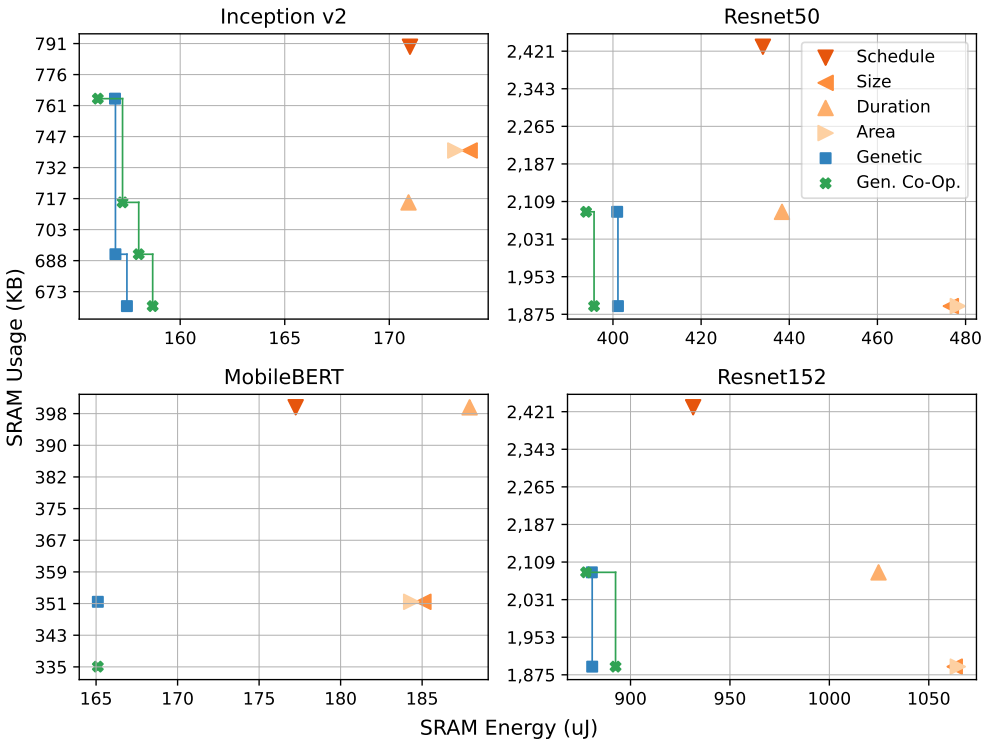

Figure 4.8.: Pareto front of the SRAM allocation of Inception v2, MobileBERT, ResNet50, and ResNet152.

usage points, it is unable to optimize them to the same extent with respect to energy. Given that the design space available to the Genetic Co-Optimization is a superset of that of the *plain* Genetic Algorithm, one would expect it to be able to discover these points as well.

The primary reason behind this limitation likely lies in the fact that the much larger search space of different schedules encompasses only a small subset of valid schedules. Figure 4.9 offers a visual representation of the percentage of invalid individuals in the population throughout the genetic algorithm, which are *invalid* with respect to the two checks depicted in Figure 3.7. An individual, or a specific set of schedule and allocation permutations, is considered invalid if it either violates the graph dependencies or exceeds the SRAM capacity. In the case of the *plain* Genetic Algorithm with no SRAM size constraints (red), all individuals are valid. Upon introducing an SRAM size constraint (blue), the percentage of invalid individuals increases for both the Inception and ResNet models to approximately 30% and 40%, respectively. This increase results from the SRAM size constraint limiting the number of possible schedules. When allowing the Genetic Co-Optimization to modify the schedule as well (green and orange), the percentage of invalid individuals surges to nearly 85% for the Inception v2 model and 80% for ResNet50. Consequently, only about 15% to 20% of the individuals are retained for the next generation in each genetic algorithm itera-

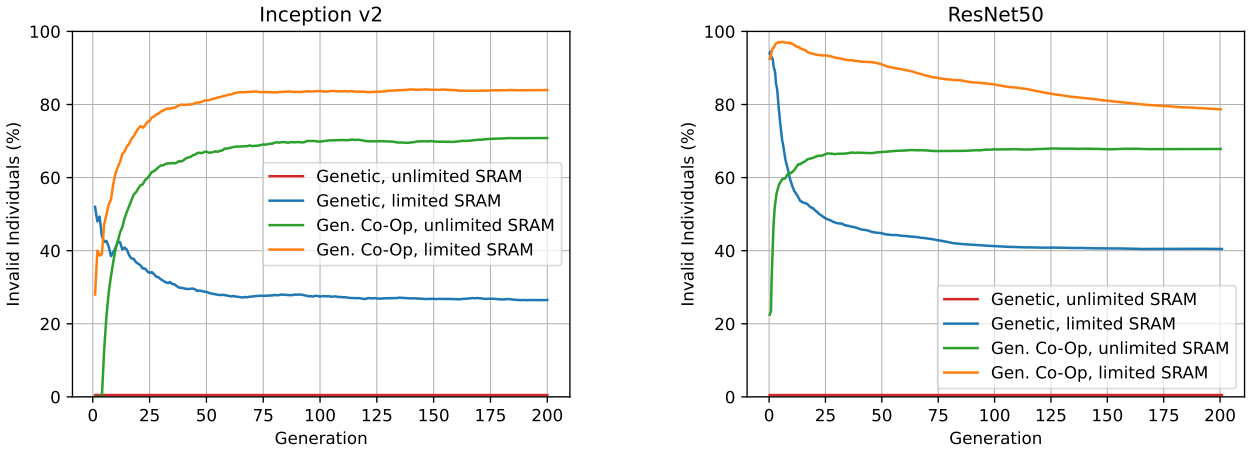

Figure 4.9.: Percentage of invalid individuals in the population over the course of 200 generations of the Genetic Algorithm and the Genetic Co-Optimization approach.

tion. This significantly curtails the effectiveness and, more importantly, the efficiency of the Genetic Co-Optimization approach, necessitating a much larger number of generations to identify a comparably good solution as the *plain* Genetic Algorithm approach.

A potential mitigation strategy for this issue could involve permitting the Genetic Co-Optimization to generate only valid schedules. As observed above, intrinsically forcing the Genetic Co-Optimization to do so through penalization seems ineffective. Instead, an approach similar to the one used in allocation permutation may be required, where all possible permutations result in valid assignments (in the case of unlimited SRAM size). The Genetic Algorithm generates a permutation, and a second intermediate algorithm then creates valid assignments based on this permutation order, as demonstrated in Algorithm 4. A similar method could be employed to ensure that all generated permutations yield valid schedules. One potential approach to facilitate this could involve creating a set of all valid schedule permutations and allowing the Genetic Co-Optimization to use its permutation to index into this *valid subset* of all possible schedules. While this might be an interesting approach, it remains to be seen whether it can actually discover a better solution than the *plain* Genetic Algorithm approach. As there is no readily available algorithm to map a permutation to a valid schedule designing and implementing this approach is left for future work.

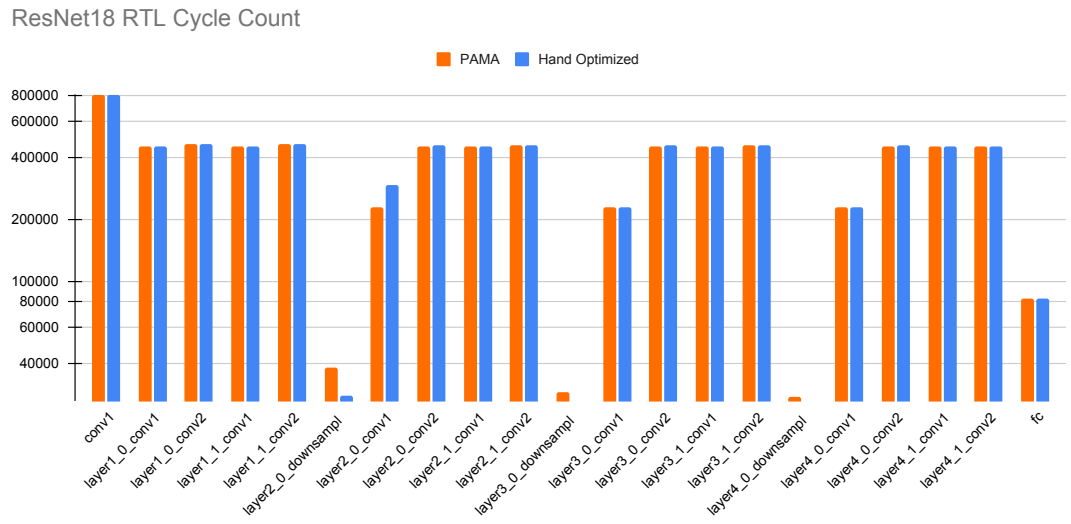

Figure 4.10.: RTL cycle count of the ResNet18 model running on the MINOTAUR SoC. Comparison of hand-optimized and PAMA-generated code.

4.4. Performance

This section will showcase the single-chip performance of the PAMA compiler when targeting the MINOTAUR SoC. We will compare PAMA's performance against hand-optimized schedules. The performance metrics employed will be the RTL cycle count for each layer of both ResNet18 and a single MobileBERT encoder block.

4.4.1. ResNet18

Figure 4.10 depicts the cycle count of each fused ResNet18 layer optimized by PAMA using its graph optimization and operator fusion mechanism, as presented in Chapter 3, compared to a hand-optimized implementation created by a domain expert. The main factor for determining the cycle count for each layer is not PAMA itself but rather ZigZag, which is responsible for finding the optimal loop order and loop tiling.

As can be seen, PAMA is able to identify and match the same patterns identified by the domain expert, resulting in the same number of bars in the figure. In terms of the cycle count, ZigZag is able to match the hand-optimized implementation for all layers, except for the down-sampling layers. This is most likely due to the fact that the MINOTAUR cost model used by ZigZag is not yet capable of accurately predicting the performance of the

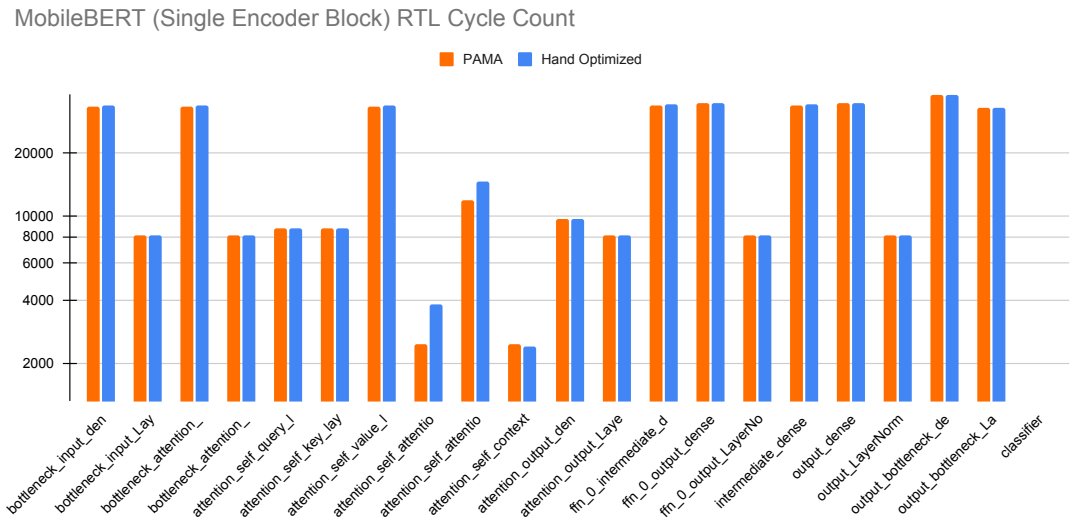

Figure 4.11.: RTL cycle count of the MobileBERT model running on the MINOTAUR SoC. Comparison of hand-optimized and PAMA-generated code.

1x1 convolutional layers used in the down-sampling layers. This is a known issue and will be addressed in future work. Furthermore, PAMA is able to beat the hand-optimized implementation for the first layer of the second residual block (`layer2_0_conv1`).

4.4.2. MobileBERT

Similar to ResNet18, the results of MobileBERT, depicted in Figure 4.11 show that PAMA and ZigZag are again able to match or even exceed the hand-optimized implementation for most of the layers. In particular, the first two layers of the self-attention block show that ZigZag's loop tiling is able to outperform the domain expert. This is a significant step towards the goal of automated model optimization for SoC architectures, making it possible to optimize extremely large models for a given SoC without the need for domain expertise.

Nevertheless, it must be noted that the presented results require an accurate description of the SoC architecture, both in terms of the supported patterns and the cost model. However, this upfront effort is only required once and can then be reused to map a large set of models.

4.5. PAMA CLI

One of the primary benefits of the PAMA compiler lies in its ability to automatically generate most of the performance metrics and visualizations discussed earlier during the compilation and mapping process. This capability enables users to assess a model's performance on a specified SoC without executing the model on the target simulator or hardware. This information is also conveniently supplied in text form when using PAMA's CLI (Command Line Interface). For instance, when the PAMA compiler is run on ResNet18, the following output is produced:

Listing 4.1: PAMA CLI on ResNet18

```
(venv) fabian@desk$ python3 run_pama.py --model_path models/onnx/resnet18.onnx
INFO: --- Preprocessor running on 'models/onnx/resnet18.onnx' ---
INFO: --- Preprocessor done ---
INFO: --- ZigZag running on 'models/onnx/resnet18.onnx' ---
INFO: --- ZigZag done ---
INFO: --- PAMA running on 'models/onnx/resnet18.onnx' ---
INFO: Estimated runtime: 916.61ms (18,332,222 cycles)
INFO: Required SRAM: 733,184B (34.96\%)
INFO: Required RRAM: 11,694,656B (92.94\%)
INFO: Energy SRAM: 134.85uJ (1.34x opt)
INFO: Energy RRAM: 3,151.91uJ (1.02x opt)
INFO: --- PAMA done ---
```

As illustrated, PAMA initially executes the preprocessor, subsequently invokes ZigZag for loop tiling and loop order optimization, and ultimately runs PAMA itself. Upon completion, the PAMA compiler presents several performance-related metrics to the user. The first three metrics include the estimated runtime, as well as the required SRAM and RRAM. The estimated runtime represents the model's total runtime in clock cycles, whereas the required SRAM and RRAM denote the peak tensor usage and necessary parameter storage, respectively. Additionally, the percentage of required SRAM and RRAM is provided. The final two metrics encompass the model's energy consumption with respect to SRAM and RRAM and the ratio of energy consumption to the optimal energy consumption. In the context of SRAM, the optimal consumption is defined as the lowest energy consumption achievable through the tightest possible packing of all tensors. For RRAM, allocation is considered optimal if no *overflow* extends beyond a single super bank[1]. As can be seen, PAMA's mapping achieves 1.34x of the optimal SRAM and 1.02x of the optimal RRAM energy consumption.

[1] Note, that one super bank can contain one, two, or four banks, depending on the bandwidth mode.

4.6. Conclusion

The experimental findings presented in this work reveal that PAMA is adept at automatically optimizing an extensive range of modern models for a given SoC architecture. In terms of peak memory utilization and energy consumption, PAMA's innovative Genetic Algorithm and Bandwidth Aware allocation techniques considerably surpass SotA methods. With respect to runtime, PAMA's integration with ZigZag yields highly optimized kernels that can match or even exceed the performance of manually optimized schedules. Furthermore, PAMA's CLI can automatically produce vital performance metrics, facilitating the evaluation of a model's performance on a specified SoC without the need for time-consuming simulators or execution on actual hardware.

Besides the results delineated here, Appendix A offers an in-depth examination of the genetic algorithm's performance, particularly the performance across multiple generations. In addition, Appendix C displays supplementary artifacts generated by PAMA, encompassing the IR at various stages of the compilation process and the resultant code.

5. Conclusion and Outlook

5.1. Conclusion

This thesis introduced PAMA, a power-aware deep learning compiler for ultra-low-power edge devices with stringent resource constraints. By integrating with ZigZag, a SotA DNN mapping framework, and employing novel memory planning algorithms, PAMA has demonstrated its ability to optimize deep learning model execution on MINOTAUR, a power-optimized deep learning accelerator developed at Stanford University. Using PAMA, we have shown that up to 97.35% of static on-chip memory energy can be conserved while preserving the same performance and accuracy as hand-optimized implementations.

Chapter 2 provided an overview of the MINOTAUR deep learning accelerator SoC, elucidating its high-level architecture and core design principles. MINOTAUR features a highly optimized systolic array-based accelerator, along with a novel on-chip memory architecture that supports fine-grained power gating and variable bandwidth access modes. A comprehensive understanding of the hardware architecture proves essential for developing a compiler capable of effectively targeting the advanced features of the MINOTAUR SoC.

Chapter 3 presented PAMA, a power-aware deep learning compiler for ultra-low-power edge devices. Chapter 3 began by discussing previous work on deep learning compilers and the motivation for developing a new compiler. It then detailed PAMA's design, including the compiler architecture, memory planning algorithms, and its integration with external mapping tools like ZigZag. Chapter 3 concluded by presenting the SotA memory planning algorithms developed for PAMA, which can optimize the execution of deep learning models on deep learning accelerators while simultaneously considering both energy and memory constraints.

Chapter 4 evaluated PAMA through experiments on a variety of modern deep learning models, such as CNNs and Transformer networks. The results demonstrated that the proposed compiler can effectively reduce power consumption while concurrently matching or even surpassing the performance of hand-optimized implementations. The results underscore the efficacy of the proposed memory planning algorithms, which yield substantial power savings while maintaining performance and accuracy.

5.2. Outlook and Future Work

The results presented in this work primarily concentrate on evaluating PAMA within a single-chip configuration. However, the growing size of deep learning models and the increasing demand for larger memory resources call for the utilization of multiple interconnected chips to support these larger models. In this regard, several MINOTAUR SoCs can be interconnected to establish a multi-chip system, capable of scaling with minimal overhead to deploy deep learning models of arbitrary sizes. Ongoing collaborations among research groups at Stanford University, KU Leuven, and TU Munich are exploring the use of MINOTAUR, Stream [32], and PAMA to facilitate the deployment of progressively larger deep learning models on multi-chip configurations. A preliminary outline of the proposed multi-chip system is depicted in fig. 5.1, emphasizing the integration of PAMA, ZigZag, and Stream on the software side and an automated HLS/RTL design flow on the hardware side. This proposed multi-chip system is designed to create the *illusion* of a single-chip system, offering a lower cost and greater flexibility compared to a single-chip solution. The outcomes of this collaboration will be detailed in future work.

In addition to the multi-chip system, there are other directions within the scope of PAMA worth exploring to further enhance its capabilities:

The genetic Co-Optimization-based memory planning introduced here could be refined by reducing the search space and permitting only the selection of valid schedules during the evolutionary process. This might potentially accelerate convergence and enable a more aggressive co-optimization of schedule and memory address selection.

Moreover, ongoing research is concentrating on supporting on-device training using a *rolling window* approach to accommodate the limited memory resources of edge devices. This necessitates careful consideration of scaling and quantization factors to ensure that models can be trained and updated using the limited precision provided by the 8-bit Posit data type. This enhancement would render the proposed compiler more versatile, allowing models to be trained and updated on edge devices themselves, rather than depending on a central server. Such an approach could enable applications demanding real-time adaptation to new data, such as personalized speech recognition, and reduce the reliance on connectivity between devices and the central server.

Integrating the system with sensors and executing actual end-to-end applications on the physical MINOTAUR SoC (once returned from fabrication) is another area of forthcoming work. This integration will facilitate the evaluation of PAMA in a real-world context and offer a deeper understanding of the challenges that need to be addressed to enable the deployment of deep learning models on edge devices.

In summary, the proposed power-aware deep learning compiler constitutes a significant step forward in enabling the deployment of deep learning models across a wider range of edge devices. PAMA effectively optimizes the execution of deep learning models on ultra-low-power edge devices, taking into account the power and memory constraints of the target device. The memory planning strategies facilitate efficient resource utilization on both conventional SRAM and innovative non-volatile RRAM. The future research directions outlined in this section, if realized, would further enhance the performance and capabilities of the compiler, bringing the power of deep learning closer to the devices that permeate our daily lives.

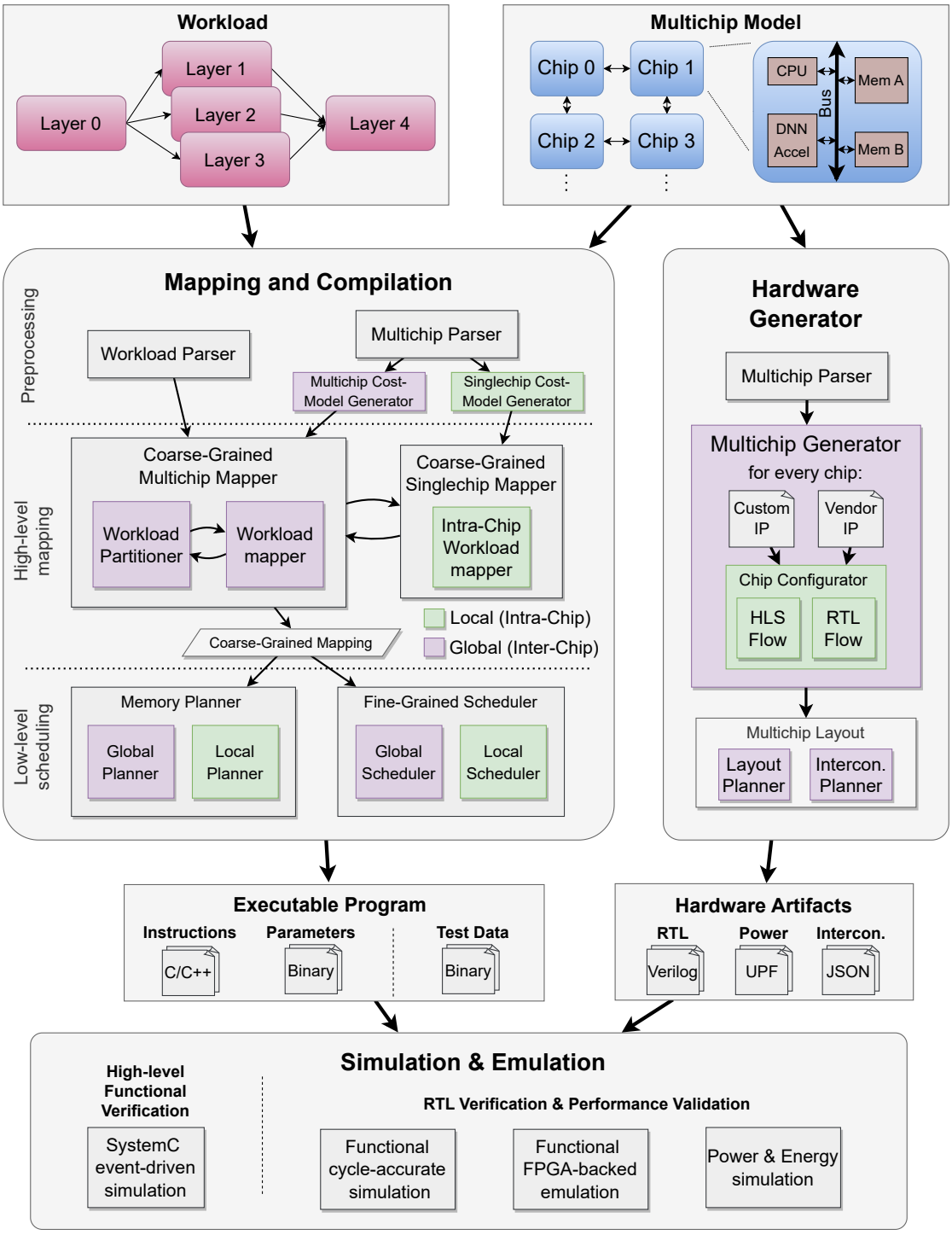

Figure 5.1.: Holistic Multi-Chip Compilation Flow incorporating PAMA, ZigZag, and Stream on the software side, and an automated HLS/RTL design flow on the hardware side.

A. Genetic Algorithm

This section provides a more detailed overview of the genetic algorithm employed in PAMA, encompassing additional information with regards to the fitness function, crossover and mutation operators, and selection method. Additionally, it elucidates the various parameters of the genetic algorithm and their influence on the search process. In addition to the information provided here, the reader is encouraged to refer to the PAMA source code.

A.1. Genetic Operators

The genetic algorithm utilized in PAMA is built upon the DEAP library [77]. DEAP serves as an evolutionary computation framework for rapid prototyping, providing a suite of data structures and tools necessary for the implementation of evolutionary computation techniques, such as genetic algorithms. Genetic algorithms are heuristic search algorithms inspired by the process of natural selection, wherein the fittest individuals are chosen for reproduction, while the less fit individuals are discarded.

PAMA employs genetic algorithms for the process of SRAM planning with the aim of optimizing the placement of activation tensors within the memory space. An example allocation of tensors for MobileBERT is presented in Figure 4.6. In contrast to greedy algorithms, genetic algorithms can discover a more optimal allocation of the activation tensors, thereby reducing energy consumption.

To guide this search, several data structures and functions must be implemented. This section offers supplementary background information on their implementation. For a visual representation of the genetic algorithm, refer to Figure 3.7.

A.1.1. Individual

As illustrated in Figure 3.7, the genetic algorithm is based on a population of individuals. These individuals are represented by a Python class, named `MappingConfiguration()`. It comprises two permutations, implemented as simple Python lists. One permutation corre-

sponds to the schedule of the nodes in the graph, denoted as `schedule_perm`, and the other represents the SRAM allocation order of usage records, referred to as `usage_record_perm`. The schedule of the nodes in the graph is used to determine the order in which the nodes are executed. The SRAM allocation order of usage records is used to determine the order in which the activation tensors are allocated in the SRAM using the greedy SRAM allocation algorithm illustrated in Algorithm 4.

The schedule permutation `schedule_perm` is initialized with the topological order of the nodes in the graph, ensuring that the algorithm begins with a valid permutation. In contrast, the usage record permutation `usage_record_perm` is initialized with a random permutation of the usage records. Since any usage record permutation is valid, the algorithm can commence with a random permutation without the risk of invalidating all individuals in the population.

These two permutations completely characterize the individual and are employed to ascertain the fitness of the individual. The fitness of an individual is determined by the fitness function, which is discussed in the next section.

A.1.2. Fitness Function

The fitness function in PAMA's genetic algorithm is employed to evaluate the fitness of an individual and decide whether it will be chosen for reproduction. In PAMA, this function is named `evaluate_individual()`. The fitness of an individual represents the measure of its performance within the search space.

To enable parallelization of the algorithm across multiple CPU cores, the fitness function must remain *pickable*[1]. However, passing all required input variables as arguments to the fitness function would render it non-pickable[2]. Consequently, the fitness function retrieves the necessary input variables from the global namespace. This is achieved by using the `global` keyword, which permits the function to access global variables. The only exception is the `individual` parameter. As such, `evaluate_individual()` accepts an individual as input, obtains the remaining required input variables from the global namespace, and returns a tuple containing the individual's fitness. The fitness of an individual is determined as follows:

[1] In Python, a function being *pickable* implies that it can be serialized using the *pickle* module, permitting it to be saved to a file or transmitted over a network. This is essential when utilizing multiprocessing, as pickling facilitates the distribution of functions and their arguments among multiple processes.

[2] Stream's individuals, for example, are not pickable, rendering it impossible to parallelize the genetic workload allocation in Stream's multi-chip mapper.

1. Retrieve global variables: the graph `G`, MINOTAUR multi-chip model `multichip`, base schedule `base_schedule` (the topological schedule), and SRAM configuration `sram_config` (extracted from the multi-chip model for convenience).
2. Extract schedule and usage record permutations from the input individual.
3. Generate the graph's schedule by sorting the base schedule in accordance with the individual's `schedule_perm`.
4. If the resulting schedule is invalid, return an invalid fitness score (i.e., a fitness score of infinity).
5. Employ the schedule to ascertain the execution times of the nodes in the graph.
6. Produce usage records according to the individual's schedule permutation.
7. Sort the usage records based on the individual's `usage_record_perm`.
8. Perform the SRAM planning using the sorted usage records.
9. If the peak SRAM size surpasses the permissible SRAM size, the function returns an invalid fitness score.
10. Estimate the SRAM energy consumption utilizing the generated schedule and the SRAM configuration.
11. Lastly, return the fitness score as a tuple containing the estimated SRAM energy, peak SRAM size, and Hamming distance between the schedule permutation and the default schedule permutation.

Based on the returned fitness score, the genetic algorithm determines whether the individual is suitable for reproduction. In addition to the two apparent values, the fitness score tuple also incorporates the Hamming distance between the individual's schedule permutation and the default schedule permutation. This is done to reward individuals possessing a schedule permutation distinct from the default schedule permutation. This approach encourages the algorithm to explore the search space of potential schedules and aids in averting the search from becoming trapped in a local optimum.

A.1.3. Evolutionary Algorithm

DEAP offers a variety of ready-to-use evolutionary algorithms that necessitate minimal modification. In PAMA, the genetic algorithm is built upon the `eaMuPlusLambda()` algorithm. The name originates from the algorithm's utilization of a population of size `mu` and the generation of `lambda` offspring per generation, selecting the best `mu` individuals from both the population and offspring for the subsequent generation (hence the name `mu` plus `lambda`).

Although `lambda` can generally be chosen arbitrarily, in PAMA, it is set to twice the population size `mu`. For a comprehensive list of evolutionary algorithms provided by DEAP, refer to `https://deap.readthedocs.io/en/master/api/algo.html`.

A.1.4. Crossover and Mutation

The genetic algorithm employs two operators to generate new individuals from existing ones: crossover and mutation. Crossover is utilized to combine the genetic material of two individuals, whereas mutation serves to modify an individual's genetic material.

PAMA employs the `cxOrdered()` crossover operator, which combines the two schedule permutations and two usage record permutations of two distinct individuals, respectively. This operator ensures that the resulting permutation is guaranteed to be free of duplicates by combining the genetic material of two individuals in a specific manner.

The mutation operator used in PAMA is `mutShuffleIndexes()`, which randomly shuffles the indexes of the schedule permutation and the usage record permutation. The `indpb` parameter dictates the probability of a mutation occurring and is set to 0.05 by default. Consequently, there is a 5% chance that a mutation will transpire for each index in both the schedule permutation and usage record permutation.

A.2. Genetic Algorithm Parameters

The genetic algorithm and evolutionary process are steered by several parameters. These parameters must be meticulously tuned to ensure the algorithm's ability to find an optimal solution. In this section, the genetic algorithm parameters are discussed.

The two most crucial parameters are the population size and the number of generations. The population size determines the number of individuals in the population, while the number of generations indicates the iterations the algorithm will execute before stopping. Since it is unknown what allocations can be discovered, PAMA's genetic algorithm does not utilize a stopping criterion by default, but instead iterates through all specified generations before halting.

The population size and number of generations are intimately related. A larger population size results in an expanded search space, allowing the algorithm to identify better solutions. However, this also leads to increased runtime and memory footprint. The population size's influence on the algorithm's convergence is illustrated in Figure A.1. As demonstrated, the algorithm converges more rapidly with a larger population size. This is particularly true

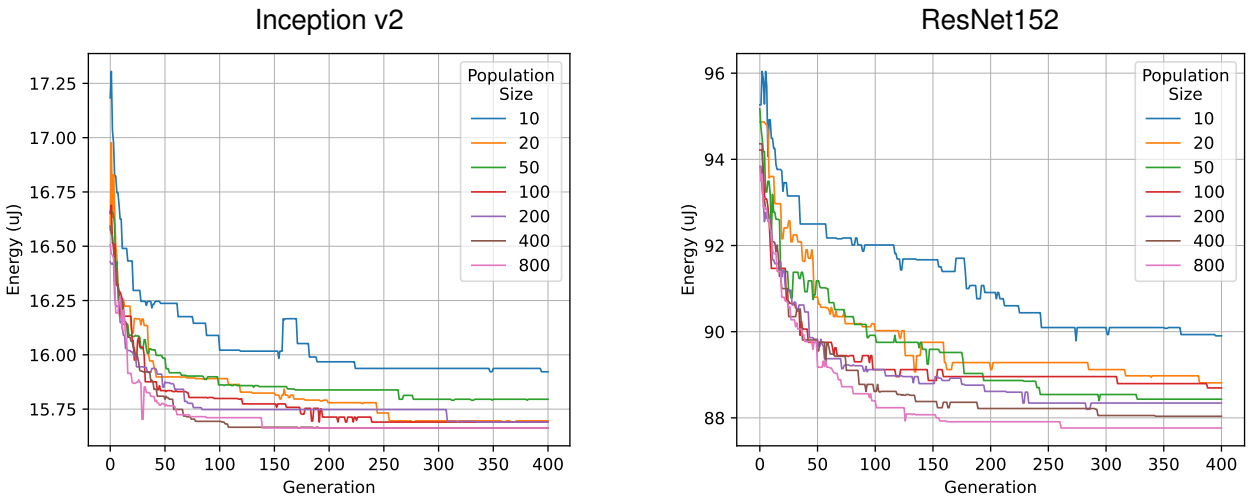

Figure A.1.: Energy consumption of the best individual in the population over the course of 400 generations using different population sizes.

for larger and more complex network architectures, such as Inception v2. In comparison, ResNet152 converges more swiftly with a smaller population size due to its simpler network architecture and smaller search space. There is no definitive rule of thumb for the optimal population size, which is often determined through trial and error. Similarly, determining the ideal number of generations is unclear. However, the number of generations should be sufficiently large to ensure the algorithm has ample opportunity to explore the search space.

In addition to the population size and the number of generations, the genetic algorithm encompasses several other parameters. These parameters serve to control the evolutionary process at each generation. Two pivotal parameters are the probability of mutation and the probability of crossover. Ideally, their sum should equal 1. In PAMA, these two parameters were determined, once again through trial and error, to be 0.6 and 0.3, respectively. Setting these values too low will render the evolutionary process sluggish and inert, whereas setting them too high will cause the search process to become unstable, preventing the algorithm from converging to an optimal solution.

B. Integration with External Mapping Tools

PAMA can integrate with external DNN mapping and exploration tools. In this chapter, we will discuss two such tools that are currently compatible with PAMA. The first tool is ZigZag [27], which focuses on mapping deep learning workloads onto single-chip accelerators. The second tool, Stream [32], provides a comprehensive framework for mapping deep learning workloads to multi-chip systems.

Both ZigZag and Stream are implemented in Python, are open-source, and can be accessed through their respective GitHub repositories [62, 87]. Interested readers are encouraged to delve into these tools and explore their source code. The author would like to express gratitude to Arne Symons and his colleagues at KU Leuven for their exceptional work in developing these tools, ensuring ease of use, and providing comprehensive documentation.

B.1. ZigZag

ZigZag [27] is a DSE framework specifically tailored for embedded deep learning accelerators. It is designed to explore and optimize both the architecture and workload mapping on these architectures. ZigZag primarily aims to enhance the performance and energy efficiency of single-core/single-chip deep learning systems by minimizing data movement overhead and maximizing accelerator utilization. Unlike other DSE frameworks, ZigZag can search over uneven mapping opportunities using intelligent mapping search strategies.

The original ZigZag paper [27] proposed several heuristic mapping strategies, which were later extended by Symons et al. [71] to provide a more comprehensive search over the space of potential temporal mappings.

ZigZag's innovative Memory-Centric Design Space Representation decouples operands (weights, inputs, and outputs), memory hierarchy, and mapping scenarios, thus opening up a novel mapping space for DSE. As a result, ZigZag can identify superior design points compared to other frameworks.

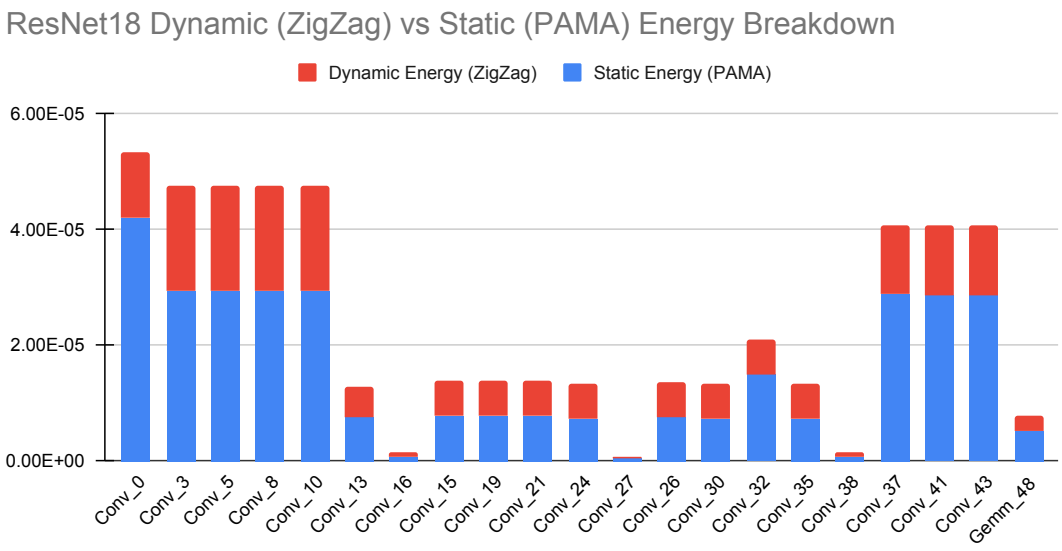

Figure B.1.: Energy breakdown between static energy consumption, measured by PAMA, and dynamic energy, measured by ZigZag of ResNet18 on MINOTAUR.

Similar to PAMA, ZigZag offers an ONNX-compliant Frontend that is continuously being refined. The framework also provides an easy-to-use Python API (Application Programming Interface) and the ability to annotate ONNX models with ZigZag's mapping information. This feature allows users to integrate ZigZag into their own workflows.

Another critical feature of ZigZag is the Loop Relevance Principle, built upon the Memory-Centric Design Space Representation. This enables the framework to systematically extract essential information, such as memory accesses and required memory bandwidth, which the Hardware Cost Estimator utilizes to determine the system's energy and performance values. It is these values that are annotated to the ONNX model and used by PAMA for further compilation.

In the context of this work, ZigZag was further extended to extract hardware information from the MINOTAUR cost model and convert it into its own Hardware Cost Estimator. This enables ZigZag and PAMA to share a cost model, which is crucial for incorporating ZigZag into PAMA.

Figure B.1 and fig. B.2 depict the energy breakdown between static energy consumption, as measured by PAMA, and dynamic energy, as measured by ZigZag, for ResNet18 and MobileBERT on MINOTAUR. These values were generated using an earlier version of the MINOTAUR cost model. The current version, which offers greater accuracy, will be utilized

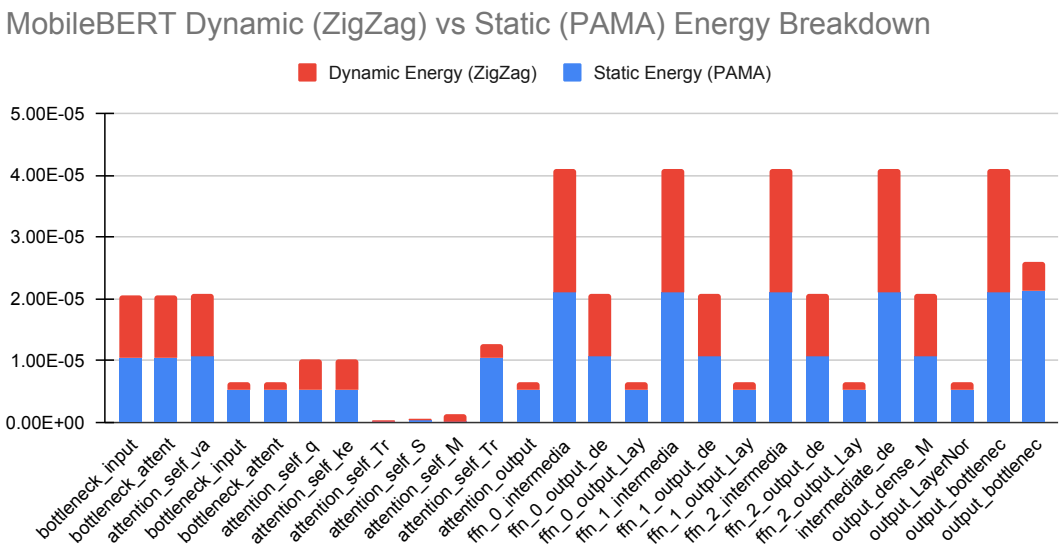

Figure B.2.: Energy breakdown between static energy consumption, measured by PAMA, and dynamic energy, measured by ZigZag of MobileBERT on MINOTAUR.

in future work. Nevertheless, the energy breakdown between static and dynamic energy consumption serves as a good indicator of the collaborative data sharing between ZigZag and PAMA.

B.2. Stream

Stream is a state-of-the-art, open-source framework designed for the exploration of fine-grained scheduling of DNNs on multi-chip hardware accelerator systems [32, 88]. The framework has been recently updated to support integration with the MINOTAUR multi-chip model and the PAMA compiler, thus enabling multi-chip DNN mapping and scheduling configurations. Stream encompasses a fast fine-grained data dependency generator, a genetic algorithm-based layer-core allocator, and a heuristic-driven multi-chip scheduler. The efficacy of Stream has been validated through comparisons with several cutting-edge hardware accelerator implementations.

In addition to multi-chip scheduling, Stream is also capable of employing a depth-first layer-fused scheduling approach. In contrast to PAMA's layer fusion, Stream splits layers and combines them into a depth-first execution pattern, resulting in up to 30x energy-delay product (EDP) reduction compared to traditional layer-by-layer scheduling. This depth-first

scheduling approach further enables Stream to reduce the memory requirements in of both SRAM and RRAM, making it possible to schedule even the largest of DNN workloads on resource-constrained multi-chip systems.

Although Stream's integration with PAMA is still under development, preliminary findings indicate its potential to identify advantageous multi-chip mapping and scheduling configurations for an array of DNN workloads. For further information, readers are encouraged to delve into Stream's source code and accompanying documentation.

C. PAMA Artifacts

This section offers a comprehensive overview of the artifacts produced by PAMA. PAMA's CLI allows users to compile ONNX models and generate target code for deep learning accelerators, such as MINOTAUR.

C.1. Graph Transformations

A clear understanding of the graph operations utilized in PAMA can be achieved by examining the ONNX graph and PAMA's IR DAG at various stages throughout the compilation process. The ONNX model visualizations are generated using Netron [89], whereas the IR DAGs are produced with the aid of PAMA's internal graph visualization utility, which leverages functions provided by NetworkX [72].

C.1.1. ResNet18 Example

We begin by highlighting the graph transformations applied to the ResNet18 model and its corresponding ONNX representation. The ONNX graph is depicted on the left side of Figure C.1. Subsequently, the graph undergoes annotation by ZigZag. The resulting annotated graph is displayed on the right side of Figure C.1. This annotated graph is then converted into PAMA's IR DAG using PAMA's parser, as illustrated in Figure C.2. The IR DAG undergoes optimization via the DAG optimization techniques discussed earlier, followed by pattern matching through the PME. The outcome is visualized in Figure C.3. This graph serves as the foundation for scheduling, memory planning, and code generation.

C.1.2. MobileBERT Example

In a manner akin to the ResNet18 example, we elucidate the graph transformations applied to the MobileBERT model and its ONNX representation. The ONNX graph, prior to preprocessing and frontend optimizations, is presented in Figure C.4. Following preprocessing and frontend optimizations, the ONNX graph is depicted on the left side of Figure C.5.

Subsequently, the graph undergoes annotation via the ZigZag framework. The resulting annotated graph is displayed on the right side of Figure C.5. This annotated graph is then converted into PAMA's IR DAG, as illustrated in Figure C.6. The IR DAG undergoes optimization through the DAG optimization techniques previously discussed, followed by pattern matching. The outcome is visualized in Figure C.7.

C.2. Generated Code

Utilizing PAMA's C-Struct Backend, C++ code is generated that can be compiled with a RISC-V GCC toolchain and executed on the MINOTAUR SoC. For instance, the generated code corresponding to the second fused layer in ResNet18 is presented in Listing C.1. This C-Struct encompasses all the essential information required for the layer's execution on the SoC, including memory addresses, loop order and tilings, as well as various flags that govern the layer's execution process.

Besides the C-Struct, PAMA generates a collection of data structures during the compilation process that serve to reference other layer artifacts, such as input, weight, and output tensors, particularly during unit testing. A short snippet of these data structures can be found in Listing C.2. For further details, readers with a keen interest are encouraged to explore the CLI tool documentation.

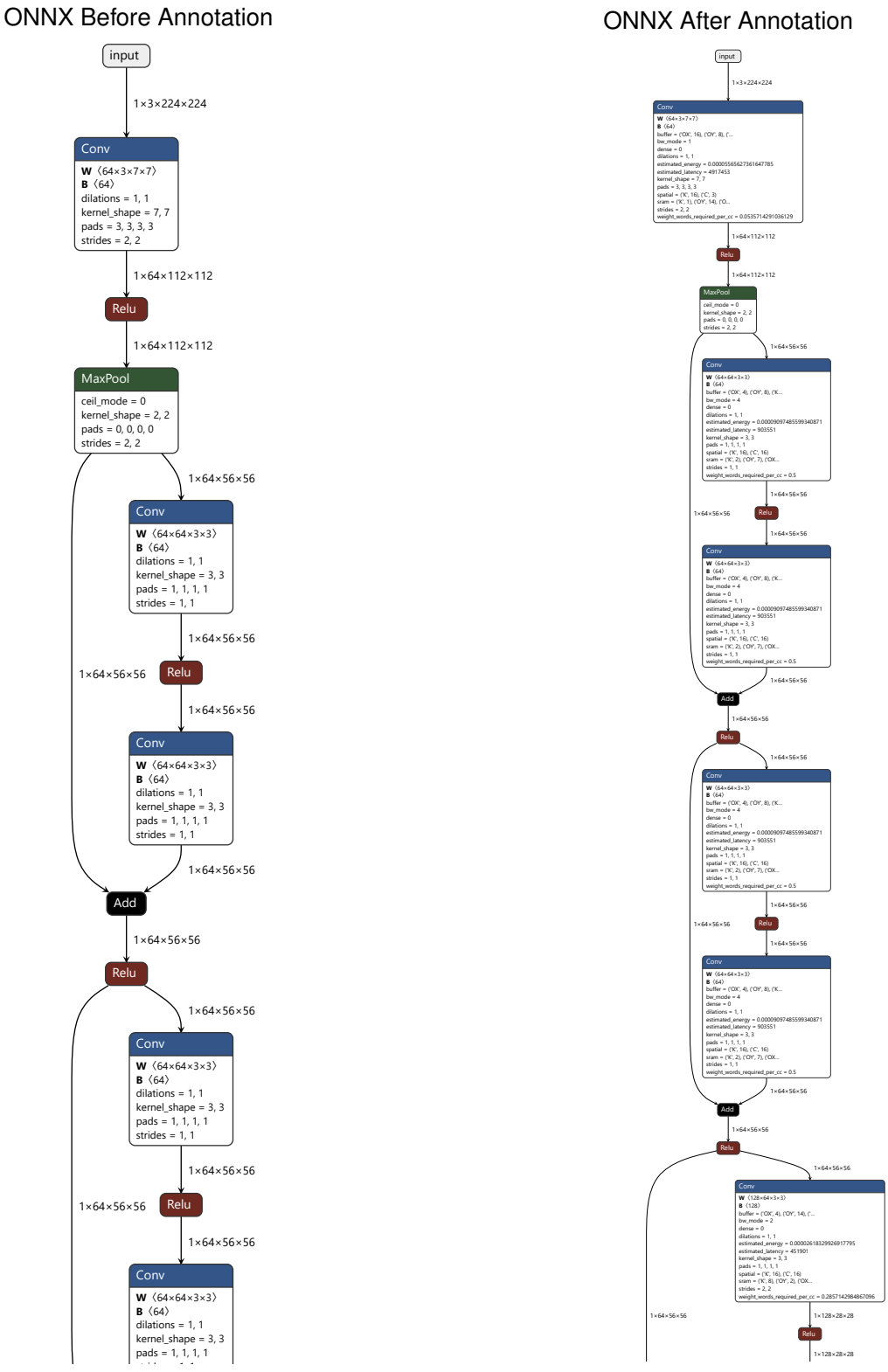

Figure C.1.: ResNet18 ONNX graph before and after ZigZag annotation.

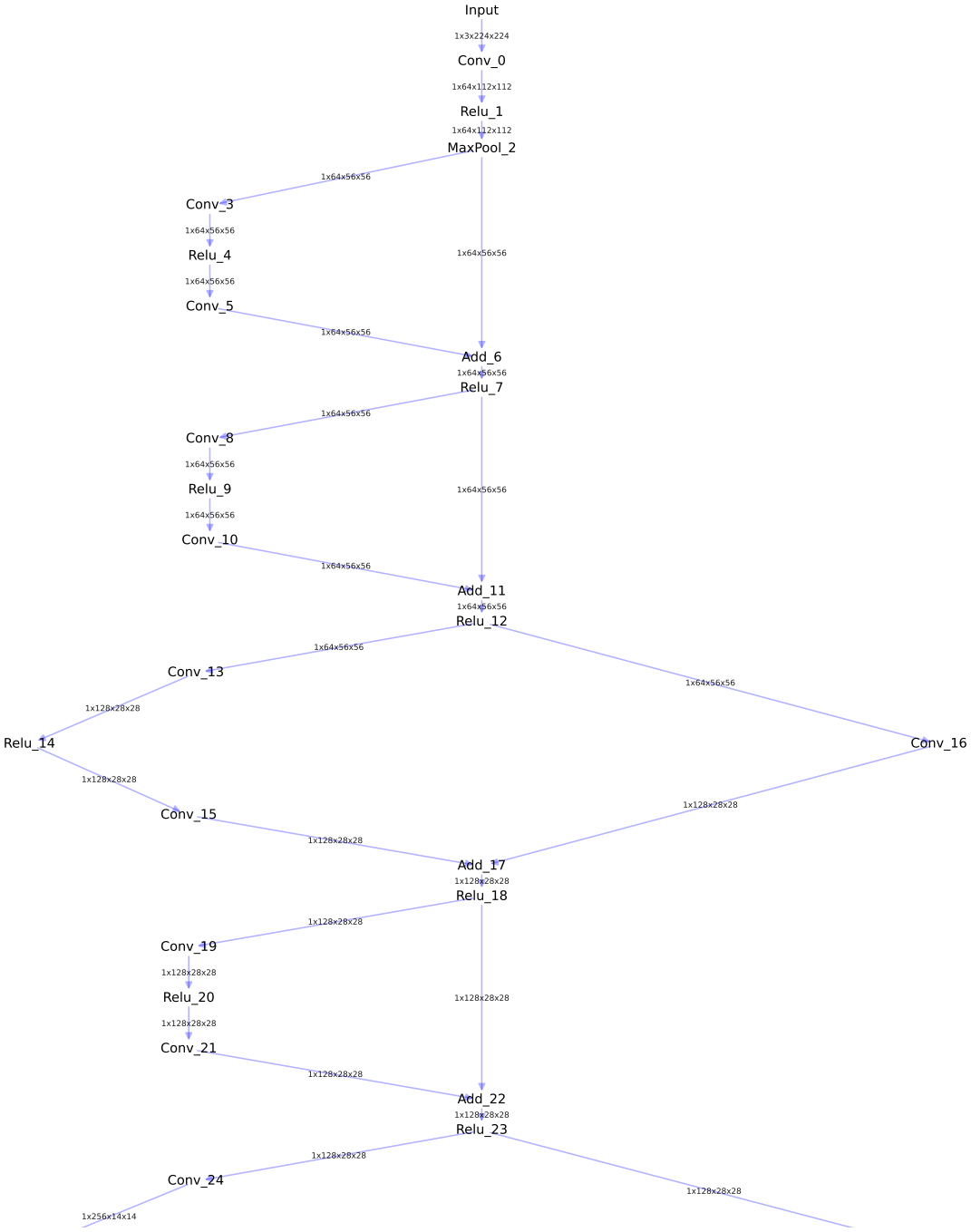

Figure C.2.: ResNet18 PAMA IR DAG before pattern matching.

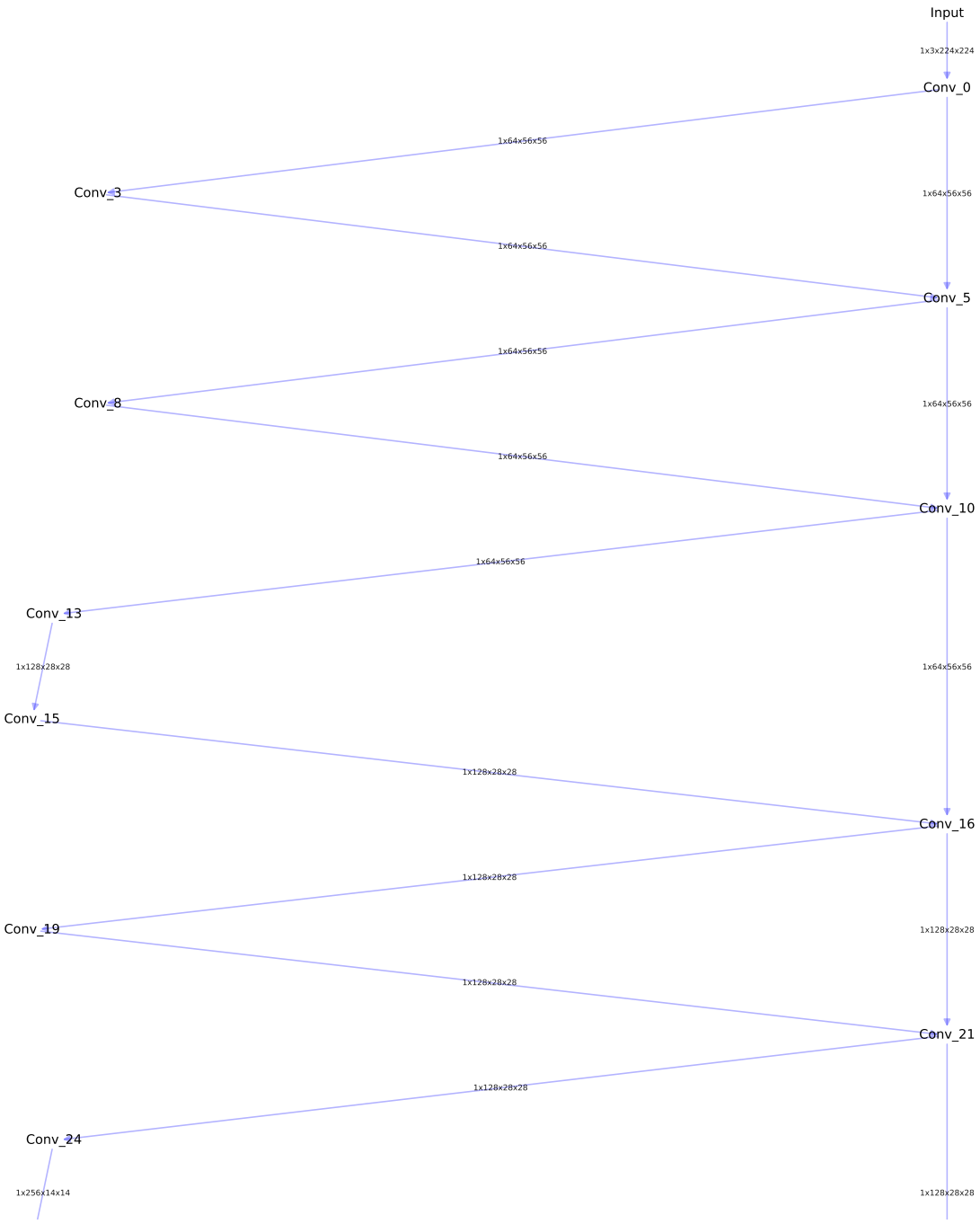

Figure C.3.: ResNet18 PAMA IR DAG after pattern matching.

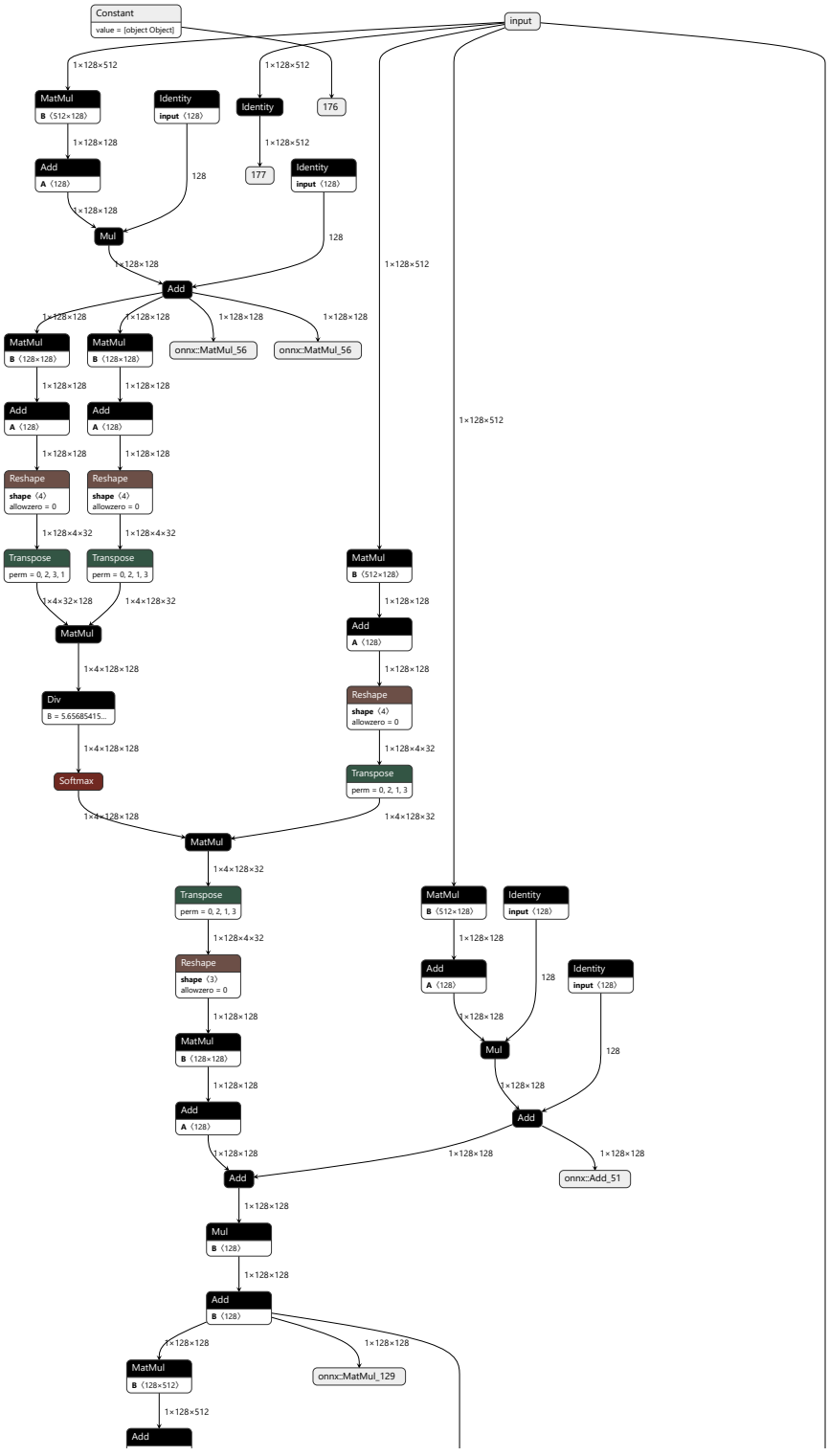

Figure C.4.: MobileBERT ONNX graph before preprocessing and frontend optimizations.

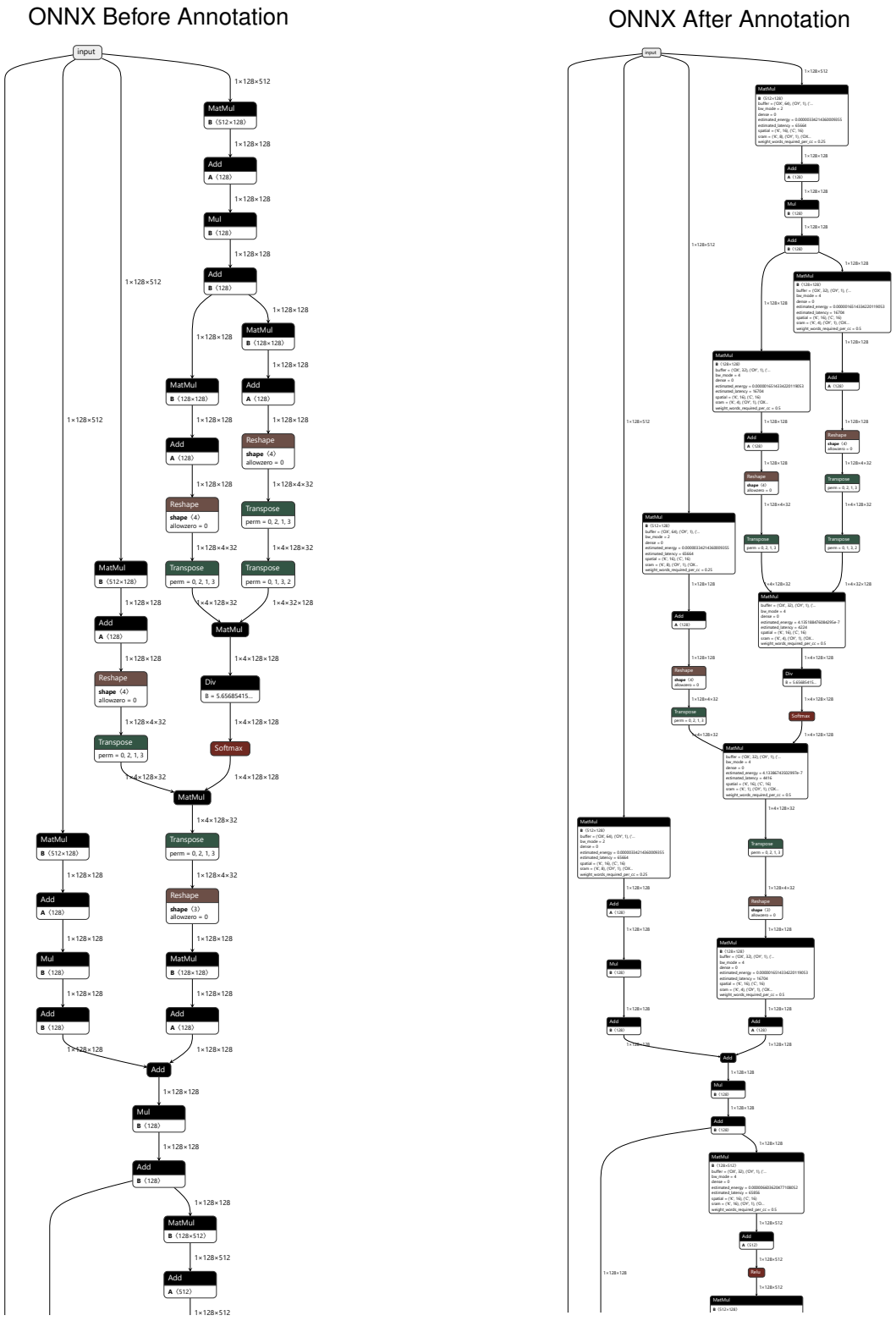

Figure C.5.: MobileBERT ONNX graph before and after ZigZag annotation.

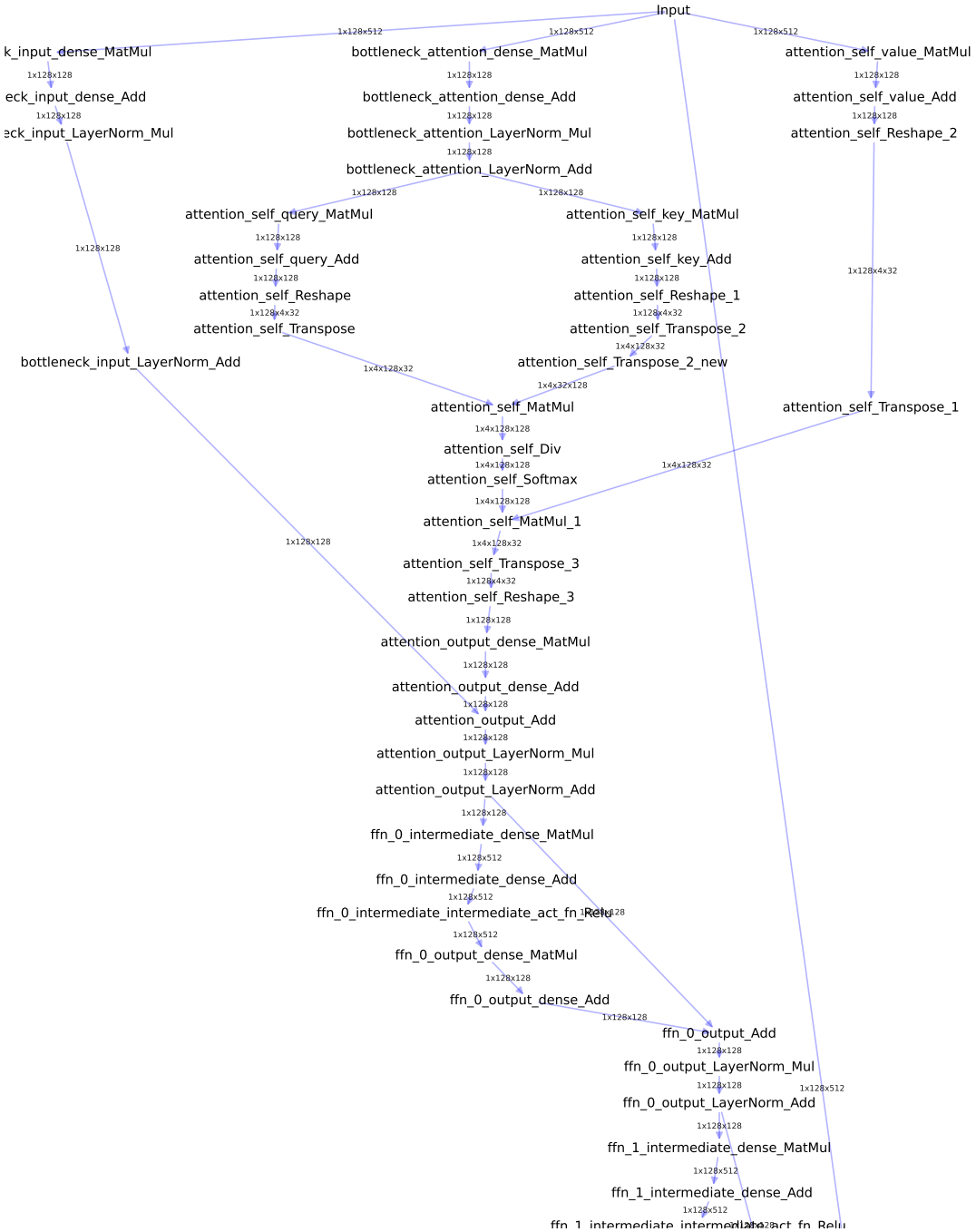

Figure C.6.: MobileBERT PAMA IR DAG before pattern matching.

Figure C.7.: MobileBERT PAMA IR DAG after pattern matching.

```
const SimplifiedParams Conv_3_params{
    .INPUT_OFFSET = 131072,
    .WEIGHT_OFFSET = 518144,
    .OUTPUT_OFFSET = 331776,
    .WEIGHT_TRANSPOSE = false,
    .loops = {{14, 7, 2, 1, 1, 1}, {4, 3, 3, 2, 8, 4}},
    .inputXLoopIndex = {0, 5},
    .inputYLoopIndex = {1, 4},
    .reductionLoopIndex = {3, 0},
    .weightLoopIndex = {2, 3},
    .fxIndex = 2,
    .fyIndex = 1,
    .weightReuseIndex = {4, 5},
    .STRIDE = 1,
    .REPLICATION = false,
    .RELU = true,
    .BIAS = true,
    .BIAS_OFFSET = 555008,
    .RESIDUAL = false,
    .RESIDUAL_OFFSET = -1,
    .MAXPOOL = false,
    .AVGPOOL = false,
    .WEIGHT = true,
    .STORE_IN_ACC = false,
    .ACC_FROM_ACC = false,
    .SOFTMAX = false,
    .ATTENTION_SCALING = false,
    .FC = false,
    .NO_NORM = false,
    .SOFTMAX_GRAD = false,
    .FC_GRAD = false,
    .NO_NORM_GRAD = false,
    .RELU_GRAD = false,
    .BIAS_GRAD = false,
    .CROSS_ENTROPY_GRAD = false,
    .MSE_GRAD = false,
    .BCE_WITH_LOGITS_GRAD = false,
    .INPUT_TRANSPOSE = false,
    .CONCAT_INPUT = false,
    .CONCAT_WEIGHT = false,
    .SPLIT_OUTPUT = false,
    .GRAD_CLIPPING = false,
    .GRAD_CLIPPING_UNIT_TEST = false,
    .WEIGHT_SPLITTING = false,
    .WEIGHT_RESIDUAL_OFFSET = -1,
    .learningRate = 0.0,
    .ACC_T_INPUT = false,
    .ACC_T_WEIGHT = false,
    .ACC_T_OUTPUT = false,
    .ACC_T_RESIDUAL = false,
    .outputExpBias = 0,
    .WEIGHT_UPDATE = false,
    .ERROR_FEEDBACK = false,
    .depthwise = false,
    .sram_banks = {ON, ON, ON, OFF, OFF, OFF, OFF, OFF},
    .rram_banks = {ON, ON, ON, ON, OFF, OFF, OFF, OFF, OFF, OFF, OFF, OFF},
    .bandwidth_mode = QUAD,
    .bank_offsets = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    .name = "Conv_3",
};
```

Listing C.1: Generated C-Struct for 2nd ResNet18 Layer.

```
const std::map<std::string,  Files> filesCodeGen{
    {"Conv_0", {"input",  "Conv_0_weight", "Conv_0_bias", "input_8"}},
    {"Conv_3", {"input_8",  "Conv_3_weight", "Conv_3_bias", "onnx::Conv_129"}},
    {"Conv_5", {"onnx::Conv_129", "Conv_5_weight", "Conv_5_bias", "input_24", "input_8"}},
    {"Conv_8", {"input_24",  "Conv_8_weight", "Conv_8_bias", "onnx::Conv_136"}},
    {"Conv_10", {"onnx::Conv_136", "Conv_10_weight", "Conv_10_bias", "input_40", "input_24"}},
    {"Conv_13", {"input_40", "Conv_13_weight", "Conv_13_bias", "onnx::Conv_143"}},
    {"Conv_15", {"onnx::Conv_143", "Conv_15_weight", "Conv_15_bias", "onnx::Add_210"}},
    {"Conv_16", {"input_40", "Conv_16_weight", "Conv_16_bias", "input_60", "onnx::Add_210"}},
    {"Conv_19", {"input_60", "Conv_19_weight", "Conv_19_bias", "onnx::Conv_152"}},
    {"Conv_21", {"onnx::Conv_152", "Conv_21_weight", "Conv_21_bias", "input_76", "input_60"}},
    {"Conv_24", {"input_76", "Conv_24_weight", "Conv_24_bias", "onnx::Conv_159"}},
    {"Conv_26", {"onnx::Conv_159", "Conv_26_weight", "Conv_26_bias", "onnx::Add_225"}},
    {"Conv_27", {"input_76", "Conv_27_weight", "Conv_27_bias", "input_96", "onnx::Add_225"}},
    {"Conv_30", {"input_96", "Conv_30_weight", "Conv_30_bias", "onnx::Conv_168"}},
    {"Conv_32", {"onnx::Conv_168", "Conv_32_weight", "Conv_32_bias", "input_112", "input_96"}},
    {"Conv_35", {"input_112", "Conv_35_weight", "Conv_35_bias", "onnx::Conv_175"}},
    {"Conv_37", {"onnx::Conv_175", "Conv_37_weight", "Conv_37_bias", "onnx::Add_240"}},
    {"Conv_38", {"input_112", "Conv_38_weight", "Conv_38_bias", "input_132", "onnx::Add_240"}},
    {"Conv_41", {"input_132", "Conv_41_weight", "Conv_41_bias", "onnx::Conv_184"}},
    {"Conv_43", {"onnx::Conv_184", "Conv_43_weight", "Conv_43_bias", "onnx::Gemm_190", "input_132"}},
    {"Gemm_48", {"onnx::Gemm_190", "Gemm_48_weight", "Gemm_48_bias", "output"}},
};

const std::vector<std:: string> orderCodeGen{
    "Conv_0",
    "Conv_3",
    "Conv_5",
    "Conv_8",
    "Conv_10",
    "Conv_13",
    "Conv_15",
    "Conv_16",
    "Conv_19",
    "Conv_21",
    "Conv_24",
    "Conv_26",
    "Conv_27",
    "Conv_30",
    "Conv_32",
    "Conv_35",
    "Conv_37",
    "Conv_38",
    "Conv_41",
    "Conv_43",
    "Gemm_48",
};
```

Listing C.2: Subset of Generated ResNet18 Artifacts.

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[2] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[3] Zhuang Liu et al. “A convnet for the 2020s”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 11976–11986.

[4] Joseph Redmon et al. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[5] Shaoqing Ren et al. “Faster r-cnn: Towards real-time object detection with region proposal networks”. In: *Advances in neural information processing systems* 28 (2015).

[6] Zhengxia Zou et al. “Object detection in 20 years: A survey”. In: *arXiv preprint arXiv:1905.05055* (2019).

[7] Dario Amodei et al. “Deep speech 2: End-to-end speech recognition in english and mandarin”. In: *International conference on machine learning*. PMLR. 2016, pp. 173–182.

[8] Li Deng et al. “Recent advances in deep learning for speech research at Microsoft”. In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 8604–8608.

[9] Ali Bou Nassif et al. “Speech recognition using deep neural networks: A systematic review”. In: *IEEE access* 7 (2019), pp. 19143–19165.

[10] Xipeng Qiu et al. “Pre-trained models for natural language processing: A survey”. In: *Science China Technological Sciences* 63.10 (2020), pp. 1872–1897.

[11] Thomas Wolf et al. “Transformers: State-of-the-art natural language processing”. In: *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 2020, pp. 38–45.

[12] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[14] Matthew Tancik et al. “Block-nerf: Scalable large scene neural view synthesis”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 8248–8258.

[15] Sorin Grigorescu et al. “A survey of deep learning techniques for autonomous driving”. In: *Journal of Field Robotics* 37.3 (2020), pp. 362–386.

[16] Sourav Garg et al. “Semantics for robotic mapping, perception and interaction: A survey”. In: *Foundations and Trends® in Robotics* 8.1–2 (2020), pp. 1–224.

[17] Yulan Guo et al. “Deep learning for 3d point clouds: A survey”. In: *IEEE transactions on pattern analysis and machine intelligence* 43.12 (2020), pp. 4338–4364.

[18] Dayong Wang et al. “Deep learning for identifying metastatic breast cancer”. In: *arXiv preprint arXiv:1606.05718* (2016).

[19] Jianfei Yang et al. “MetaFi: Device-free pose estimation via commodity WiFi for Metaverse avatar simulation”. In: *arXiv preprint arXiv:2208.10414* (2022).

[20] Zhaoshuo Li et al. “Temporally consistent online depth estimation in dynamic scenes”. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2023, pp. 3018–3027.

[21] Abdullah Mujahid et al. “Real-time hand gesture recognition based on deep learning YOLOv3 model”. In: *Applied Sciences* 11.9 (2021), p. 4164.

[22] Goutam Bhat et al. “Deep burst super-resolution”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 9209–9218.

[23] Vivienne Sze et al. “Efficient processing of deep neural networks: A tutorial and survey”. In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[24] Ripan Kumar Kundu, Akhlaqur Rahman, and Shuva Paul. “A study on sensor system latency in vr motion sickness”. In: *Journal of Sensor and Actuator Networks* 10.3 (2021), p. 53.

[25] Yuyi Mao et al. “A survey on mobile edge computing: The communication perspective”. In: *IEEE communications surveys & tutorials* 19.4 (2017), pp. 2322–2358.

[26] Tianqi Chen et al. “TVM: An automated end-to-end optimizing compiler for deep learning”. In: *arXiv preprint arXiv:1802.04799* (2018).

[27] Linyan Mei et al. “Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators”. In: *IEEE Transactions on Computers* 70.8 (2021), pp. 1160–1174.

[28] Mark Horowitz. “1.1 computing’s energy problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE. 2014, pp. 10–14.

[29] Massimo Giordano et al. “CHIMERA: A 0.92 TOPS, 2.2 TOPS/W edge AI accelerator with 2 MByte on-chip foundry resistive RAM for efficient training and inference”. In: *2021 Symposium on VLSI Circuits*. IEEE. 2021, pp. 1–2.

[30] Robert M Radway et al. “Illusion of large on-chip memory by networked computing chips for neural network inference”. In: *Nature Electronics* 4.1 (2021), pp. 71–80.

[31] Lita Yang et al. “Three-Dimensional Stacked Neural Network Accelerator Architectures for AR/VR Applications”. In: *IEEE Micro* 42.6 (2022), pp. 116–124.

[32] Arne Symons et al. “Towards Heterogeneous Multi-core Accelerators Exploiting Fine-grained Scheduling of Layer-Fused Deep Neural Networks”. In: *arXiv preprint arXiv:2212.10612* (2022).

[33] Bert Moons et al. “14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi”. In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE. 2017, pp. 246–247.

[34] Paul Palomero Bernardo et al. “UltraTrail: A configurable ultralow-power TC-ResNet AI accelerator for efficient keyword spotting”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 4240–4251.

[35] Zhehong Wang et al. “An all-weights-on-chip dnn accelerator in 22nm ull featuring 24$\times$ 1 mb erram”. In: *2020 IEEE Symposium on VLSI Circuits*. IEEE. 2020, pp. 1–2.

[36] Angelo Garofalo et al. “DARKSIDE: A Heterogeneous RISC-V Compute Cluster for Extreme-Edge On-Chip DNN Inference and Training”. In: *IEEE Open Journal of the Solid-State Circuits Society* 2 (2022), pp. 231–243.

[37] Vikram Jain et al. “TinyVers: A Tiny Versatile System-on-Chip With State-Retentive eMRAM for ML Inference at the Extreme Edge”. In: *IEEE Journal of Solid-State Circuits* (2023).

[38] Davide Rossi et al. “Vega: A ten-core SoC for IoT endnodes with DNN acceleration and cognitive wake-up from MRAM-based state-retentive sleep mode”. In: *IEEE Journal of Solid-State Circuits* 57.1 (2021), pp. 127–139.

[39] Kartik Prabhu et al. “CHIMERA: A 0.92-TOPS, 2.2-TOPS/W edge AI accelerator with 2-MByte on-chip foundry resistive RAM for efficient training and inference”. In: *IEEE Journal of Solid-State Circuits* 57.4 (2022), pp. 1013–1026.

[40] John Gustafson. “Posit arithmetic”. In: *Mathematica Notebook describing the posit number system* (2017).

[41] John L Gustafson and Isaac T Yonemoto. “Beating floating point at its own game: Posit arithmetic”. In: *Supercomputing frontiers and innovations* 4.2 (2017), pp. 71–86.

[42] Mark Sandler et al. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[43] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[44] Hsiang Tsung Kung and Charles E Leiserson. “Systolic arrays (for VLSI)”. In: *Sparse Matrix Proceedings 1978*. Vol. 1. Society for industrial and applied mathematics Philadelphia, PA, USA. 1979, pp. 256–282.

[45] Xuan Yang et al. “Interstellar: Using halide’s scheduling language to analyze dnn accelerators”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 369–383.

[46] Zhiqing Sun et al. “Mobilebert: a compact task-agnostic bert for resource-limited devices”. In: *arXiv preprint arXiv:2004.02984* (2020).

[47] Zachariah Carmichael et al. “Deep positron: A deep neural network using the posit number system”. In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1421–1426.

[48] Jawar Singh, Saraju P Mohanty, and Dhiraj K Pradhan. *Robust SRAM designs and analysis*. Springer Science & Business Media, 2012.

[49] Robert David et al. “Tensorflow lite micro: Embedded machine learning for tinyml systems”. In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 800–811.

[50] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[51] Tianqi Chen et al. “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems”. In: *arXiv preprint arXiv:1512.01274* (2015).

[52] Jared Roesch et al. “Relay: A new ir for machine learning frameworks”. In: *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*. 2018, pp. 58–68.

[53] Tianqi Chen et al. “Learning to optimize tensor programs”. In: *Advances in Neural Information Processing Systems* 31 (2018).

[54] MJKlaiber and Cgerum. *[RFC] uma: Universal modular accelerator interface*. Feb. 2022. URL: https://discuss.tvm.apache.org/t/rfc-uma-universal-modular-accelerator-interface/12039.

[55] Nadav Rotem et al. “Glow: Graph lowering compiler techniques for neural networks”. In: *arXiv preprint arXiv:1805.00907* (2018).

[56] PyTorch. *Glow: Machine Learning Compiler and Execution Engine for Hardware Accelerators*. https://github.com/pytorch/glow. Accessed: March 12, 2023. 2021.

[57] Google. *TensorFlow Lite: Deploy machine learning models on mobile and edge devices*. https://www.tensorflow.org/lite. Accessed: March 12, 2023. 2023.

[58] Hyoukjun Kwon et al. “Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings”. In: *IEEE micro* 40.3 (2020), pp. 20–29.

[59] Hyoukjun Kwon et al. “Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 754–768.

[60] Angshuman Parashar et al. “Timeloop: A systematic approach to dnn accelerator evaluation”. In: *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2019, pp. 304–315.

[61] Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. “Accelergy: An architecture-level energy estimation methodology for accelerator designs”. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2019, pp. 1–8.

[62] Linyan Mei et al. *HW Architecture-Mapping Design Space Exploration Framework for Deep Learning Accelerators*. `https://github.com/ZigZag-Project/zigzag`. 2022.

[63] A. Symons et al. *ZigZag 2.0.0 Framework Documentation*. Accessed: 2023-03-15. 2023. URL: `https://zigzag-project.github.io/zigzag/publications.html`.

[64] NVIDIA. *ONNX GraphSurgeon*. GitHub repository. Available at: `https://github.com/NVIDIA/TensorRT/tree/master/tools/onnx-graphsurgeon`. 2021.

[65] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[66] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. TR-2009-179. University of Toronto, 2009. URL: `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`.

[67] Alex Wang et al. “GLUE: A multi-task benchmark and analysis platform for natural language understanding”. In: *arXiv preprint arXiv:1804.07461* (2018).

[68] Pranav Rajpurkar et al. “Squad: 100,000+ questions for machine comprehension of text”. In: *arXiv preprint arXiv:1606.05250* (2016).

[69] *Edge TPU - run inference at the edge*. Google. URL: `https://cloud.google.com/edge-tpu`.

[70] Mingzhen Li et al. “The deep learning compiler: A comprehensive survey”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2020), pp. 708–727.

[71] Arne Symons, Linyan Mei, and Marian Verhelst. “Loma: Fast auto-scheduling on dnn accelerators through loop-order-based memory allocation”. In: *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE. 2021, pp. 1–4.

[72] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[73] Zhihao Jia et al. “TASO: optimizing deep learning computation with automatic generation of graph substitutions”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 47–62.

[74] Zhihao Jia et al. “Optimizing DNN computation with relaxed graph substitutions”. In: *Proceedings of Machine Learning and Systems* 1 (2019), pp. 27–39.

[75] Byung Hoon Ahn et al. “Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices”. In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 44–57.

[76] Yury Pisarchyk and Juhyun Lee. “Efficient memory management for deep neural net inference”. In: *arXiv preprint arXiv:2001.03288* (2020).

[77] Félix-Antoine Fortin et al. “DEAP: Evolutionary Algorithms Made Easy”. In: *Journal of Machine Learning Research* 13 (July 2012), pp. 2171–2175.

[78] Google. *Protocol Buffers*. `https://github.com/protocolbuffers/protobuf`. 2008. URL: `https://protobuf.dev/`.

[79] Synopsys. *Synopsys VCS Functional Verification Solution*. Accessed: 2023-03-12. 2023. URL: `https://www.synopsys.com/verification/simulation/vcs.html`.

[80] Richard Socher et al. “Recursive deep models for semantic compositionality over a sentiment treebank”. In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1631–1642.

[81] Dor Bank, Noam Koenigstein, and Raja Giryes. “Autoencoders”. In: *arXiv preprint arXiv:2003.05991* (2020).

[82] Ilya O Tolstikhin et al. “Mlp-mixer: An all-mlp architecture for vision”. In: *Advances in neural information processing systems* 34 (2021), pp. 24261–24272.

[83] TorchVision maintainers and contributors. *TorchVision: PyTorch’s Computer Vision library*. Version 1.2.0. Nov. 6, 2016. URL: `https://github.com/pytorch/vision`.

[84] Thomas Wolf et al. “Huggingface’s transformers: State-of-the-art natural language processing”. In: *arXiv preprint arXiv:1910.03771* (2019).

[85] Christian Szegedy et al. “Rethinking the inception architecture for computer vision”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[86] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[87] Linyan Mei, Arne Symons, and Guilherme paim. *Stream: Towards Heterogeneous Multi-core Accelerators Exploiting Fine-grained Scheduling of Layer-Fused Deep Neural Networks*. `https://github.com/ZigZag-Project/stream`. 2022.

[88] Linyan Mei et al. “DeFiNES: Enabling Fast Exploration of the Depth-first Scheduling Space for DNN Accelerators through Analytical Modeling”. In: *arXiv preprint arXiv:2212.05344* (2022).

[89] Lutz Roeder. *Netron, Visualizer for neural network, deep learning, and machine learning models*. Version 1.2.0. Accessed: March 24, 2023. 2017. DOI: `10.5281/zenodo.5854962`. URL: `https://github.com/lutzroeder/netron`.