POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Integrating Design Space Exploration in Modern Compilation Toolchains for Deep Learning

Supervisors

Prof. Daniele Jahier PAGLIARI

Dr. Alessio BURRELLO

Dr. Matteo RISSO

Dr. Francesco DAGHERO

Dr. Josse VAN DELM

Candidate

Mohamed Amine HAMDI

Academic Year 2022-2023

Abstract

In recent years, the rapid growth of Artificial Intelligence (AI) and the explosion of hardware devices with AI-specific features have led to a rising demand for tools and frameworks capable of translating Deep Learning models from high-level languages like Python into lower-level code optimized for a particular hardware target, often in C.

This thesis focuses on edge heterogeneous systems, which have limited computational capabilities, low memory, and prioritize energy efficiency.

The proliferation of diverse hardware platforms and programming ecosystems makes porting AI models to every device a non-trivial task. An ideal solution would be a universal tool that can translate high-level model representations, e.g., in Python, into low level code while accommodating various hardware constraints, programming languages, and interfaces. Unfortunately, achieving this goal without compromising performance is still a challenge. For example, the TVM compiler stack is a popular open-source toolchain for deploying networks on many devices, including CPUs, GPUs, or ARM and RISC-V-based Microcontrollers (MCUs) but falls short when generating code for heterogeneous Systems-on-Chip (SoCs) containing different accelerators.

An effective approach to address this challenge is TVM-BYOC (Bring Your Own Codegen), an open-source framework built on top of TVM, targeting AI accelerator producers. BYOC relieves programmers from building and maintaining a full compiler stack. Instead, users can reuse TVM components and plug in optimized kernels for specific accelerator-supported layers. Vendors can then focus solely on optimizing their own kernel library to fully leverage their hardware. HTVM, for instance, follows this approach and integrates TVM with DORY, an end-to-end automatic deployment tool, which supports the deployment on multi-level memory based MCU, thanks to the usage of tiling and the orchestration of the computational and memory management parts. HTVM generates C code directly and offers more flexibility than vendor-specific stacks manually tuned for the hardware.

This thesis builds upon this work by replacing DORY with a more flexible tool, ZigZag, in the TVM+BYOC flow. ZigZag is a Design Space Exploration tool that identifies optimal temporal mappings within a vast search space, given a definition

of the target accelerator and workload.
ZigZag's representation relies on loops and their ordering, and its internal memory allocator can produce unevenly mapped schedules, crucial for edge devices with limited memory. Unlike DORY, ZigZag, in its prior state, could not generate code directly. This limitation is addressed through two primary integration interfaces: the TVM-to-ZigZag interface, which exports the layer structure from TVM to ZigZag, and the ZigZag-to-TVM interface, a template for code generation that accounts for the order of loops and tiling information provided by ZigZag.
With experiments on DIANA, a heterogeneous platform comprising a RISC-V core, a digital AI accelerator, and an analog one, significant improvements were achieved compared to HTVM, a previous compilation toolchain already tailored for DIANA. More specifically, executing a set of 2D convolutional layers with varying hyper-parameters on the DIANA digital accelerator, I observed an average performance improvement of 67% was observed compared to the existing ZigZag model. Compared to HTVM, we obtained an average speed-up of 26%, with peak improvements reaching up to 56% on the same layers.
Instead, executing the benchmarks that are part of the TinyML suite, this thesis offers on average a 11% perfomance gain over HTVM. This improvement is achieved already with still limited support of the networks.

Acknowledgements

I would like to express my heartfelt gratitude to all of my supervisors for their unwavering support throughout the course of this thesis. I am immensely thankful to Prof. Daniele Jahier Pagliari, Dr. Alessio Burrello, Dr. Matteo Risso, Dr. Francesco Daghero, and Dr. Josse Van Delm. Their guidance and mentorship have been truly remarkable, both in terms of their exceptional technical expertise and their outstanding human qualities.
I also extend my gratitude for their enduring patience and invaluable insights that guided me at every stage of this thesis. Without their exceptional support, this rewarding research experience would not have been possible. Furthermore, I am appreciative of their provision of the necessary equipment and instrumentation, which played a pivotal role in successfully completing the experiments.
Lastly, I wish to express my deep gratitude to my family and friends. Their unwavering support and encouragement have been a cornerstone of my academic journey, and I am profoundly grateful for their presence through every step of this path.

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XIV

1 Introduction 1

2 Background 4
- 2.1 Deep Neural Networks 4
  - 2.1.1 Shallow networks: Perceptron 4
  - 2.1.2 The Basics of Neural Networks 5
  - 2.1.3 Convolutional Neural Networks 5
  - 2.1.4 Convolutional layers 6
- 2.2 Hardware Platforms for AI 9
  - 2.2.1 Edge devices 10
  - 2.2.2 AI accelerators to improve efficiency 11
  - 2.2.3 Heterogeneous platforms 13
- 2.3 Compilers 17
  - 2.3.1 Compilers structure 18
  - 2.3.2 AI compilers 20
  - 2.3.3 How does an AI Compiler work 21

3 Related Works 24
- 3.1 Compilers for Edge devices 24
  - 3.1.1 TVM 24
  - 3.1.2 DORY 25
- 3.2 Compilers for heterogeneous systems 26
  - 3.2.1 HTVM 26
- 3.3 ZigZag 27
  - 3.3.1 Mapping Engine 29

3.3.2 Cost Evaluation 31
3.4 Deployment on Edge Devices: TinyML leader board 32

4 Material & Methods 34
4.1 Objective 34
4.2 Description of the workflow 36
4.2.1 TVM-to-ZigZag: Workload generation 37
4.2.2 ZigZag-to-TVM: Code generation 41
4.3 Testing set-up 48
4.3.1 RTL simulations 48
4.3.2 DIANA setup 48
4.4 ZigZag modification 50
4.4.1 New Computational Model Engine 50
4.4.2 Computational aspect 50
4.4.3 Memory transfer aspect 53

5 Experimental Results 55
5.1 Single Fused Convolutional Layer 55
5.1.1 Comparison of TVM+ZigZag with the Baseline 56
5.1.2 Comparison of TVM+ZigZag with HTVM 58
5.1.3 Problems of TVM+ZigZag 60
5.1.4 TVM+ZigZag New vs Old Cost Models vs HTVM 67
5.2 MLPerf Tiny 69
5.2.1 ResNet 70
5.2.2 MobileNet 71
5.2.3 DAE 71

6 Conclusions and future works 73

Bibliograpy 75

List of Tables

2.1 2D convolutional layer parameters. 9

3.1 TinyML leader board including HTVM results. Latency is in ms. . 33

5.1 First Set of Layers (24KB of activation memory and 64KB of weight memory) 56
5.2 Second Set of Layers (64KB of activation memory and 64KB of weight memory) 56
5.3 Third Set of Layers (128KB of activation memory and 64KB of weight memory) 57
5.4 First Group of Layers (256KB of activation memory and 64KB of weight memory) used for analyzing and comparing simulation data with ZigZag's expected results 61
5.5 Second Group of Layers (24KB of activation memory and 64KB of weight memory) used for analyzing and comparing simulation data with ZigZag's expected results 61

List of Figures

2.1 Taken by [12], convolutional layer from a computational view. . . . 6
2.2 This table list the sheer difference in computational capability of hardware used in the cloud and the edge devices. 9
2.3 Example of a MAC unit 11
2.4 Weight stationary 14
2.5 Output stationary 14
2.6 Row stationary dataflow in a 1d convolutional layer example. 14
2.7 No local reuse 14
2.8 An example of a generic heterogeneous platform 15
2.9 DIANA architecture that includes two different accelerator cores. . 16
2.10 Taken by [17], this flow-chart mark the separation from the back-end to the fron-end of a compiler. 18
2.11 Structure of AI compilers targeting general purpose hardware. . . . 22
2.12 AI compiler for MCUs structure. 23

3.1 Figure taken by [5], which introduces HTVM and explain how to fuse DORY inside TVM throughout the BYOC infrastructure. . . . 27
3.2 Taken by [7]. Workload representation on ZigZag. 28
3.3 Taken by [7]. Mapping representation on ZigZag. 29

4.1 Workflow of the compilation process. 37
4.2 This function is a utility used to build a 2d convolution that is supported by the digital accelerator core of DIANA. 38
4.3 Definition and initialization function of the object that is responsible for the generation of the workload and then also the next steps . . . 39
4.4 These functions are used to follow through the operations that are part of a fused layer, and will use a router to call a function dedicated to extrapolate every information useful for that operator. 40
4.5 This function extrapolates all the useful information for a 2d convolution. 41

4.6 Accelerator definition, that is built through the ZigZag classes, it includes the MAC array and the memory hierarchy. 42
4.7 Object that includes for each layer type the associated optimal spatial mapping and other useful information like the mapping between the memory operands and the layer operands. 43
4.8 Object that includes for each layer type the associated optimal spatial mapping and other useful information like the mapping between the memory operands and the layer operands. 43
4.9 The figure pictures the process that is pursued to call the ZigZag API and transform its results. 44
4.10 List containing structured for loops used by the template. 45
4.11 Code interpreted by Mako to generate a dimension structure. 46
4.12 Output generated by Mako for a dimension structure, in this case the output one. 46
4.13 DIANA board connected properly with an USB-C cable 49
4.14 Accelerator state values, focusing on the convolutional ones. 52
4.15 Simulation trace focusing over the state of the accelerator while computing a layer where the weights have a width of 1. 52
4.16 Simulation trace focusing over the state of the accelerator while computing a layer where the weights have a width of 7. 53
4.17 Code used to calculate properly the number of cycles needed for the computation of a 2D convolutional layer over DIANA. 53
4.18 Code used to calculate properly the number of transfers that are actually executed to move a block, and the overhead brought by them. 54

5.1 Results of the inference for the layers of the first set (Tab. 5.1), comparing the baseline configuration with TVM+ZigZag. 57
5.2 Results of the inference for the layers of the second set (Tab. 5.2), comparing the baseline configuration with TVM+ZigZag. 58
5.3 Results of the inference for the layers of the third set (Tab. 5.3), comparing the baseline configuration with TVM+ZigZag. 58
5.4 Results of the inference of the layers of the first set (Tab. 5.1), comparing the HTVM configuration with TVM+ZigZag. 59
5.5 Results of the inference of the layers of the second set (Tab. 5.2), comparing the HTVM configuration with TVM+ZigZag. 59
5.6 Results of the inference of the layers of the third set (Tab. 5.3), comparing the HTVM configuration with TVM+ZigZag. 60
5.7 Results of the computational cost for the inference of the layers in the first group (Tab. 5.4) compared with ZigZag's evaluation. . . . 62
5.8 Results of the computational cost for the inference of the layers in the second group (Tab. 5.5) compared with ZigZag's evaluation. . . 62

5.9 Comparison between the evaluation of the memory onloading cost fullfilled by ZigZag cost model and the actual latency that it takes for the layers that are part of the first group(Tab. 5.4). 63
5.10 Comparison between the evaluation of the weight transfer cost fullfilled by ZigZag cost model and the actual latency that it takes for the layers that are part of the first group(Tab. 5.4). 64
5.11 Comparison between the evaluation of the inpu transfer cost fullfilled by ZigZag cost model and the actual latency that it takes for the layers that are part of the first group(Tab. 5.4). 64
5.12 Comparison between the evaluation of the memory onloading cost fullfilled by ZigZag cost model and the actual latency that it takes for the layers that are part of the second group(Tab. 5.5). 64
5.13 Results of the input transfer for L2 of the second group(Tab. 5.5) browken down over actual transfer and overhead. 65
5.14 Comparison between the memory offloading cost evaluation by ZigZag's cost model and the actual latency for the layers in the first group (Tab. 5.4). 65
5.15 Comparison between the memory offloading cost evaluation by ZigZag's cost model and the actual latency for the layers in the second group (Tab. 5.5). 65
5.16 Comparison between the evaluation fulfilled by ZigZag's updated cost model and the actual results for set A (Tab. 5.1). 66
5.17 Comparison between the evaluation fulfilled by ZigZag's updated cost model and the actual results for set B (Tab. 5.2). 66
5.18 Comparison between the evaluation fulfilled by ZigZag's updated cost model and the actual results for set C (Tab. 5.3). 66
5.19 Results and comparison of the inference of layers in set A (Tab. 5.1). 67
5.20 Results and comparison of the inference of layers in set B (Tab. 5.2). 67
5.21 Scheduling produced by the off-the-shelf version of ZigZag for L1 of set A (Tab. 5.1). 68
5.22 Scheduling produced by the new and upgraded version of ZigZag for L1 of set A (Tab. 5.1). 68
5.23 Results and comparison of the inference of layers of the first set (Tab. 5.1) compiled with TVM+ZigZag and HTVM. 68
5.24 Results and comparison of the inference of layers of the second set (Tab. 5.2) compiled with TVM+ZigZag and HTVM. 68
5.25 Results and comparison of the inference of layers of the third set (Tab. 5.3) compiled with TVM+ZigZag and HTVM. 68
5.26 Scheduling produced by HTVM for L3 of set A (Tab. 5.1). 69
5.27 Scheduling produced by TVM+ZigZag for L3 of set A (Tab. 5.1). . 69

5.28 Results of the inference of ResNet with three configurations of TVM+ZigZag, each with different LPF limit values, and HTVM. . . 70
5.29 Results of the inference of MobileNet with three configurations of TVM+ZigZag, each with different LPF limit values, and HTVM. . . 71
5.30 Results of the inference of the DAE network with three configurations of TVM+ZigZag, each with different LPF limit values, and HTVM. 72

Acronyms

DL
: Deep Learning

DNNs
: Deep Neural Networks

AI
: Artificial Intelligence

SoC
: System on Chip

GPU
: Graphic processing unit

GPGPU
: General Purpose Graphic Processing Unit

TPU
: Tensor Processing Units

MAC
: Multiply and accumulate

BYOC
: Bring Your Own Codegen

ReLU
: Rectified Linear Unit

MLP

Multi-layer Perceptron

CNNs

Convolutional Neural Networks

MCU

Microcontroller unit

CPU

Central processing unit

RAM

Random Access Memory

IoT

Internet of Things

RISC

Reduced instruction set computer

ALU

Arithmetic linear unit

Chapter 1

Introduction

Artificial Intelligence (AI) is a constantly developing field, where research and development are defining new ways to think to computation, with new algorithms that help both industries, human beings and also their health. The applications are several, from object classification and detection to generative ones that can help artists gather new inspiration and ideas, like DALL-E [1] or DeepDream [2]. As a direct consequence, many novel architectures, including dedicated AI accelerators, have been proposed and developed. These accelerators provide significantly better performance both in terms of latency and energy consumption over general-purpose hardware for executing AI tasks.
However, this hardware growth has not been matched by corresponding developments in software. Because of the current landscape with complex architectures the deployment of a Deep Neural Network model complexity has also increased. This is even more true in the deployment of networks including new up and coming operators, which have yet to be supported by the dedicated hardware.
The difficulty of the deployment is affecting heavily the compiler side of the task, where Deep Learning frameworks currently rely on vendor-specific libraries. Each vendor is required to build up a full compiler stack which takes advantage of the dedicated hardware in an optimal way. These vendor's libraries require lots of manual effort to be developed and end up being too specialized and not portable enough to new architectures even from the same vendor. Using the vendor's libraries therefore requires some degree of manual tuning. These vendors must also make tough decision on the support level of new operators, where there may not be an optimized implementation. An ideal solution would involve a global framework that can be easily adopted by any vendor; however, currently, targeting portability implies many challenges related to performance.
An interesting approach to address this issue is the one proposed by the TVM [3] compiler through the BYOC [4] (Bring Your Own Codegen) extension, a customizable add-on in an open-source and general-purpose compiler, that allows to specify

custom optimized code to deploy specific layers. Its purpose is to help programmers concentrate on low-level optimizations while reusing most of the components required for a deep learning compiler and exploiting TVM implementations, like the set of graph level optimization, which transforms the computational graph into an equivalent one. Other components that are taken care of by the framework are the graph partitioning system, which assign each layer, or group of layers, to the best hardware available based on a cost system. Finally to wrap everything together BYOC offers also an execution engine that executes the graph sequentially.
This thesis focuses is on the development of a new BYOC that targets heterogeneous systems. An example of a TVM+BYOC implementation is HTVM, a tool extends the capabilities of TVM by targeting edge heterogenous devices. HTVM [5] exploits TVM for the compilation of layers unsupported by the targeted architecture, while for the supported one it employs DORY [6], an end-to-end deployment tool for Microcontrollers (MCUs), for the tiling and code generation. However due to DORY limitations, this thesis aims to exploit a more general tool for this purpose. The first limitation of DORY concerns the restricted search space to map the layers to the hardware, in terms of tensors allocation and tiling dimensions, limited by the tool to facilitate the search. While the second one refers to the required human knowledge to define a priori a set of hyperparameters, such as the execution loops order for a convolution.
To overcome these limitations this thesis extends the core of HTVM integrating ZigZag to the workflow. ZigZag [7] is a Design Space Exploration (DSE) tool that identifies optimal schedule within a vast search space, given a definition of the target accelerator and workload. The schedule is an ordering and mapping of the for loops that are required and are involved in the computation. The mapping can be even or uneven, where an even mapping requires all the loops to be associated to the same memory level for each operand. While an uneven mapping doesn't have these constraints and leaves freedom to the memory allocation process.
DORY, unlike ZigZag, doesn't support 2 key features. The first one is loop ordering, therefore DORY relies on a fixed ordering for each targeted hardware, which can imply a drop in optimal performances. The second feature that DORY lack support for is uneven mapping. Unevenly mapped schedules can optimize the hardware hardware by utilizing the memory instances in a clever way. This feature is crucial for edge devices with limited memory.
The main contributions achieved throughout this thesis are as follows:

- a well detailed overview of the current landscape and methodologies used by the state-of-the-art for the compilation processes on both edge devices and heterogeneous ones.
- the integration of the end-to-end open source DNN compiler TVM with the novel DSE tool ZigZag following the BYOC(Bring Your Own Codegen) pattern,

making it possible to infer layers of DNNs with DIANA digital accelerator core.

- a hardware-aware optimization of ZigZag cost modeling process, improving drastically the quality of the generated schedules.

Thanks to the above-mentioned contributions, we improved the results over the best SoA framework, HTVM, on single layers on average by an impressive 26%. While on the MLPerf Suite reference networks, Resnet, Mobilenet and DAE, the gains are respectively 16%, 0,6% and 16,5%.
The thesis is organized into 6 chapters, the first one has been a brief introduction to the challenges the thesis tackles on, then Chapter 2 focuses on the background theory information that are at the fundament of the thesis work, ranging from the Deep Neural Networks fundaments to the world of hardware platforms for AI and finally completing the chapter with the compilers theory aspect, defining also the differences from a tipycal compiler w.r.t. an AI one. In Chapter 3, the related work and state of the art are explored, analyzing the current landscape for compilation tools targeting edge devices and also the ones for heterogeneous systems, subsequently also the Design Space Exploration tools are analyzed and finally the chapter is completed by an analysis on the current deployment results. Chapter 4 delves into the methodologies and setup used, examining the challenges faced and the process that led to the integration to complete the core of the compiler. Chapter 5 instead presents the set of results that were gathered and highlights important insights that can be observed. Finally, Chapter 6 concludes the thesis with some final thoughts over the complete project and also some possible future extensions ideas.

Chapter 2

Background

2.1 Deep Neural Networks

Deep Neural Networks (DNNs) [8] have emerged as a revolutionary breakthrough in the field of Artificial Intelligence and Machine Learning. Inspired by the structure and functioning of the human brain, DNNs are a subset of neural networks that can model complex relationships and solve highly intricate problems.
We'll focus more in depth on Convolutional Neural Networks with respect to the other models since the goal of the thesis is to design a new framework do deploy neural networks on HW accelerators.

2.1.1 Shallow networks: Perceptron

The perceptron is an historical concept that was used as a building block for the majority of modern neural networks. The perceptron [9] was proposed during the 1950s as a simple way to model a biological neuron. At the core a perceptron is a computational unit, which takes a set of input values, process them through a weighted sum and applies an activation function that will generate an output. The activation function plays a crucial role in the perceptron and can be a simple step function, a sigmoid, a ReLU or a tanh function. The role of this function is to introduce non-linearity to the model, enabling the perceptron to learn patterns that are slightly more complex.
Perceptrons have several limitations, from which the main one is the fact that they can only model linearly separable functions, like the XOR problem [10] which has also been one of the historical reasons to continue research for more sophisticated solutions. Other shorcomings of the perceptrons regards its lack of depth, the sensitiveness to noisy data points and finally the fact that they are designed to do binary classification.

Even with these limitations the perceptron has played a crucial role in the development of AI. It can be also easily modeled, in fact a perceptron that has n inputs can be modeled through the following formula:

$$y = ActivationFunction(\sum_{i=0}^{n} x_i * w_i)$$

2.1.2 The Basics of Neural Networks

A neural network is a model composed of various linked nodes, known as artificial neurons or perceptrons. These neurons work collectively to process information, enabling the network to make predictions or decisions.
The neuron is the fundamental building block of a neural network, it receives inputs, processes them using a weighted sum, applies an activation function, and finally produces an output. These activations and weights defines the core of the neural network's learning process and what makes them different.
While first neural networks, that are nowadays known as shallow neural networks, were able to handle few tasks effectively, they were many limitations in tackling complex problems. Shallow networks consist of only a few hidden layers, which limits their ability to represent intricate patterns in the data.
Deep neural networks instead have multiple hidden layers that processes the data between the input and output layers. These intermediate layers allow DNNs to learn hierarchical representations of data, capturing both low-level features and high-level abstractions. This hierarchical feature extraction is the key to their success in handling complex and high-dimensional data.

2.1.3 Convolutional Neural Networks

Convolutional Neural Networks [11] are a type of Neural Networks that exploits the intrinsic additional information that resides in the evaluated data, these types of networks are broadly used with images, where usually there are lots of repeating patterns, compositionality, locality, and self-similarity across the same domain.
These patterns are also shift-invariant, meaning that traslations don't change the content of the data. All of these patterns are exploited as a prior.
CNNs are specialized for image and video processing tasks, and because of these features they result being highly effective in tasks such as object recognition, image classification, and segmentation. They are tipically composed of convolutional layers and pooling ones. The latter are layers that are used to reduce dimensionalities up to a certain range. Pooling reduces the size of dimensions significantly while still holding the main information. Pooling can be done in several ways but the most popular one is max pooling, where from each region the local maximum value is the output.

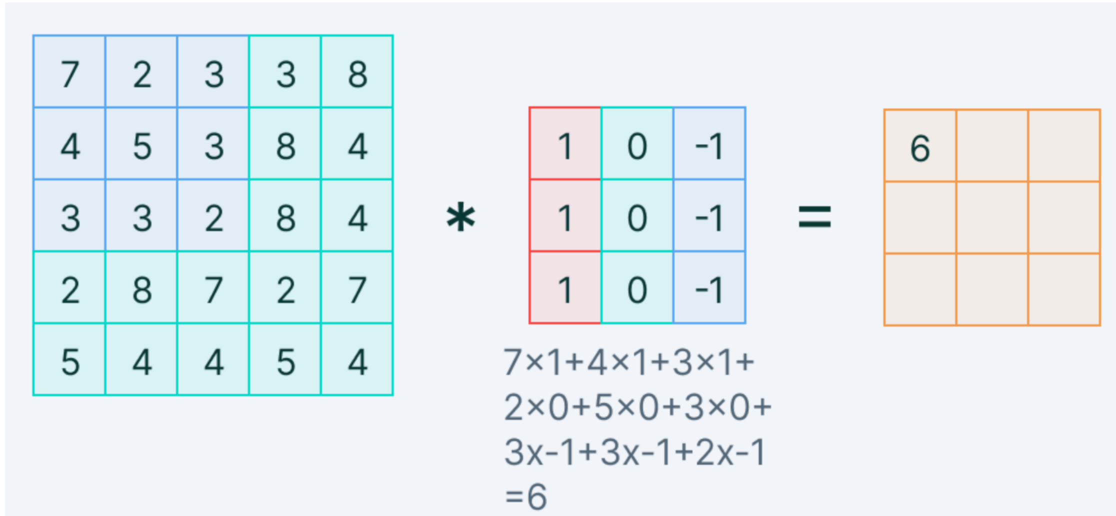

Figure 2.1: Taken by [12], convolutional layer from a computational view.

2.1.4 Convolutional layers

Convolutional layers are the core of a CNN. This is because they can be found in the hidden layers of the network and they perform the basic operation of the model, they convolve the input against a feature map. What makes this operation quite significant is the similarity it has with the response our visual cortex [13] has to stimulations.
While fully connected layers can offer simplicity and the capability to learn several features, they come with several limitations and issues. These layers are not inherently capable of translation invariance, therefore they struggle heavily to recognize the same pattern over different positions. Additionally due to their nature they include a high number of parameters and, as a consequence, they end up being very computationally intensive.
Due to these limitations nowadays, for image processing taks, the convolution operation is used. The convolution inherently possesses shift invariance, and can therefore be used to optimize the research of patterns. The convolution is an operation that is characterized by the following formula:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

This operation essentially performs a product where the feature map is moving step by step inside the input data, where Fig. 2.1 represent a step of the convolution. Additionally with respect to fully-connected the number of parameters used decrease significantly thanks to parameter sharing. Instead of connecting each neuron to

every neuron in the previous layer, convolutional layers use a set of learnable filters that are applied across the input.
The convolution operation allows CNNs to learn and detect various low-level and high-level features, such as edges, textures, and patterns, from raw pixel values. By applying convolutions to different layers of an image, CNNs can create increasingly abstract representations of the data, enabling them to recognize complex patterns and objects.
A convolutional layer is tipically composed of 2 main parts:

- **Convolution**, this operator is the main component of this layer
- **Activation Function**, an activation function is a simple function used to introduce non-linearities to the layer, thus conveniently giving it the possibility to model more complex structures and data, this function usually can range from a simple ReLU, to a sigmoid, up to a Leaky ReLu.

Finally to define the structure of the convolution, some parameters must be set:

- The **Stride** defines how much does the filter moves through the input at each step, so at stride 1 the filter moves one input data at a time, while for stride 2 it is pretty much as if the filter skips one iteration each time etc.
- **Padding**, to not lose the information by the borders of the input, usually a layer of padding is added, this means that around the borders the input is covered with 0 values. This parameter avoids also for border values to be computed less times than interior ones.
- **Dilatation**, optionally we may want to have a gap between each filter data, the result through dilatation is similar to having a chess board, where boxes of the same color will be separated, in this case they can be separeted by multiple boxes.

A convolutional layer actually depends on more parameters. These parameters vary slightly due to the dimensionality of the convolution. We'll analyze the 2D case and the shape of the 3 tensors that are part of the convolution, the output one, the weights one and finally the input tensor.
With a 2D convolution the output ends up being over a 4D space. Just from the output there are then 4 parameter, the first one called OX represents the width, the second one OY is instead the height. The third dimension K reflects the number of output channels the convolution will be computed over. The final dimension, which can be optional, is B and is the number of batches used.
Also the weights for a 2D convolution are on a 4 dimensions, that are FX, FY, C and B. FX is the width of the weights, while FY is the height. C instead is the

parameter that sets both the number of weights channels and input channels. B as stated before is for the number of batches.
Finally the input is also processed over 4 dimensions, which are IX, IY, C and B. IX is the width of the input, while IY is the height. C and B are exactly the same as the weights ones.
Out of these dimensions there are some that are not completely independent. In fact, during the convolution a windows of the weights is covering a piece of the input to generate a single output cell, thus making the input width and height dependent over the output shape and weights shape. This dependency relies on stride and dilatation because these two parameters alter the way the weights are moved across the input data. The relationship between the input shape and the other two tensors shape can be then modeled through the following equations:

$$\begin{aligned} IX &= StrideX * (OX - 1) + DilatationX * (FX - 1) + 1 \\ IY &= StrideY * (OY - 1) + DilatationY * (FY - 1) + 1 \end{aligned} \tag{2.1}$$

Therefore, when analyzing a 2D convolution, there are a grand total of 7 dimensions that are actually independent and needed. The computation of a single output cell is as follows:

$$O[k][oy][ox] = \sum_{ci}^{C} \sum_{fyi}^{FY} \sum_{fxi}^{FX} W[ci][fyi][fxi] * I[ci][oy + fyi][ox + fxi]$$

This example reflects well the amount of complexity brought by a single convolutional layer. For example with a layer that has 16 output and input channels, 1 batch, 32 pixels in width and height for the output and the input and 1 pixel for the kernel the computations required are 262144. Notably, a single computation can also require more than one cycle. A real network includes several layers with different levels of complexity, is it therefore really important to optimize the execution of networks, making it possible to achieve a bigger level of abstraction and use AI as a programming paradigm.
The following table recaps the list of parameters that are used for a 2d convolutional layer:
While deep neural networks have achieved remarkable success, they also face significant challenges. Some of these challenges include overfitting, computational complexity, and the need for vast amounts of labeled data for the training process, which can require lots of manual effort in the current landscape. Some of these limitations can be tackled through transfer learning, data augmentation, and novel architecture designs.
Deep neural networks have revolutionized Artificial Intelligence and Machine Learning, pushing the boundaries of what machines can achieve. Their ability to learn

Acronym	Name
B	Number of batches
K	Number of output channels
C	Number of input channels
OY	Output height
OX	Output width
IY	Input height
IX	Input width
FY	Kernel height
FX	Kernel width

Table 2.1: 2D convolutional layer parameters.

Metric	Single cloud server	Edge gateway	Sensor
Number of compute units (cores)	50 (CPU cores) 5000 (GPU cores)	1-4 (CPU cores) 200 (GPU cores)	1 (CPU core) **No GPU**
Operating Frequency	GHz	100s of MHz	kHz/MHz
Memory (main)	100s of GBs	GBs	**MBs**
Memory (mass)	100s of TBs	GBs	**Absent / kBs / MBs**
Power consumption	KWs	10s of W	**mW/W (active)**
Cost	10000 €	50 €	**1 €**

Figure 2.2: This table list the sheer difference in computational capability of hardware used in the cloud and the edge devices.

hierarchical representations and handle complex data has enabled groundbreaking applications across various domains. The impact of DNNs is expected to grow even further following the constant research in the field.

2.2 Hardware Platforms for AI

With the rise of AI the demand of hardware capable to efficiently run these workloads with several types of constraints, like energy consumption or area ones, have been significant. As a consequence, in the last decades there has been an explosion of devices including AI specific features.
The hardware platforms for AI usually either target the cloud or the edge, with some subdivision in these two main pots. The cloud and the edge are significantly different in terms of computational capacities. A single cloud server includes usually 50 or more CPU cores, while the typical edge gateway arrives to 4 (Fig. 2.2). Therefore, the disparity is quite evident and needs to be addressed through

special optimization techniques.

2.2.1 Edge devices

These last decades have been the dawn of the on-going research and further development of many interconnected devices, which are now proliferating also in new areas such as the Internet of Things.
Internet of Things refers to the dense and diverse network of interconnected things that exploit the Internet, or other general networks, to communicate and exchange data. The term things is used to generalize the wide array of objects that can be part of this field, objects like sensors or home devices for example. These devices that are part of this network are the so called edge devices. Which currently, due to their computing limitations, are used just to collect data and forward it to the cloud, where the data is then transformed. A solution to these limitations brings intelligence on these devices, through efficient and optimized dedicated hardware. On the other hand, in the current landscape the majority of AI workloads, are executed on the cloud, bringing the following problems:

- **Network pressure**, considering a typical application of video classification of a low resolution IoT camera the system requires up to 50Mbps of upload bandwidth. This is already a big requirement, and it's not even considering a realistic commercial case study, that usually includes hundreds of cameras.
- Inherently using cloud computation introduces some non-optimalities, the main one refers to the **latency**, which can be high and unpredictable, hurting the entire system.
- **Energy consumption**, which includes all the points of the cloud ecosystem, so the initial wireless transmission, from IoT devices or other edge devices, of data and the energy consumed by the servers. Wireless transmission energy has been stable during these years, without any significant breakthrough in research. The energy consumed by servers is several orders of magnitude higher than the edge devices one.
- The last concern regards one of the aspects that is becoming more and more important for people around the world. In fact sending sensitive raw data to the cloud without any workaround brings **privacy** concerns.

Executing DNNs directly at the edge [14] will then bring advantages in terms of latency, scalability, energy and privacy as well. However, this process requires a transformation of the methodologies used. In fact we cannot just deploy the same models over edge devices as they are, because of size, inference speeds and other constraints brought up by edge devices.

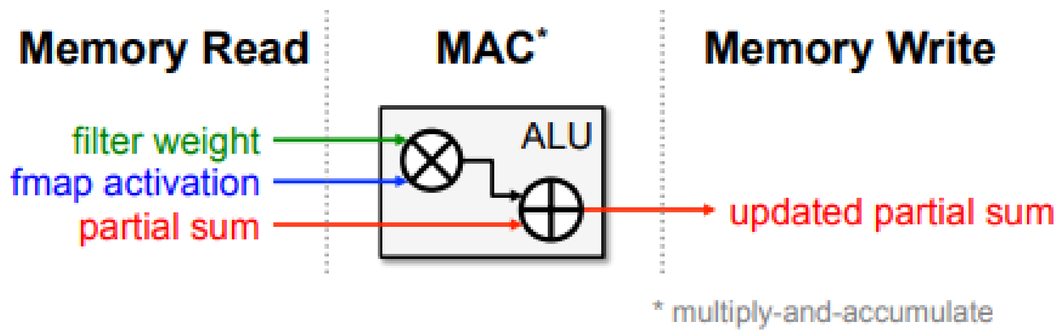

Figure 2.3: Example of a MAC unit

Edge and deep-edge

It is also important to differentiate edge devices by capabilites and features. The landscape of these devices can be then segmented in 2 main categories, the edge and the deep-edge.
Edge devices comprise more powerful hardware, like mini computers, that feature an Operating System (OS) and in general a more complete platform. On the other hand the deep-edge includes super low power devices that don't include any OS. This lack introduce several problems, even more in the DNN use case. The problems span from the inefficient resource management to several security challenges that are usually limited by the OS firewalls, updates and other mechanisms. It becomes also harder to mantain and monitor them in a reliable way. Examples of devices that are part of the deep-edge world can be smart sensors or Microcontrollers (MCUs).
The differences in terms of computational capability and platform constraints makes it even harder to deploy DNNs on the deep-edge.

2.2.2 AI accelerators to improve efficiency

Deep edge devices are usually heavily limited in computational capability, being comprised of 1 or few cores. Therefore, it's not feasible to execute efficiently the networks that are used for modern applications on deep edge hardware. These applications, like autonomous driving, robotics and healthcare, feature networks that are more and more complicated. So, it is necessary for to utilize dedicated accelerators that can compute more efficiently AI workloads. As a consequence, there have been proposed many different architectures of dedicated accelerators targeting a specific field and its requirements, such as low energy consumption, memory limitations, area size, or just raw efficient performance results.

Addressing for example the previous section dedicated to edge devices, AI accelerators are fundamental in those computing platforms, where there are many requirements about latency and specially energy efficiency during processing. Using these AI devices capabilities, edge devices can perform AI tasks locally without relying on cloud-based resources.

These AI accelerators are specialized and dedicated hardware IPs that process AI tasks very efficiently. Instead of being general purpose hardware such as CPUs or GPGPUS(General Purpose GPUs), that have to support a wide variety of operations needed to execute an arbitrary program fully, they instead focus on a small set of operation that usually don't include conditional queries. This leads designers to exploit and leverage the deterministic nature of some AI workload.

Many AI accelerators usually target a specific model to support, or a small set of models, this leads to such hardware to be tailored for a certain operation, therefore not every accelerator is able to infer every DNN, this is even more sheer when evaluating networks that include new up and coming operations.

The usual operations that are required by a typical AI workload are multiplications and accumulation, and additionally also shifting. These operations are usually executed in an efficient way by dedicated units, called MAC units [15], that are composed of registers where the result is saved and accumulated and a simple multiplier. MAC units, as it can be seen from Fig. 2.3, usually also include in their circuitry a part dedicated to compute in a faster way the typical activation functions like ReLU. Additionally many MAC units provide the ability to work with mixed precision, thus being more flexible to different networks.

AI tasks are also very good candidates for parallelization because of the presence of big matrixes, for this reason the accelerators usually are composed of a wide array of MAC units that give the possibility to unroll the computation of the operations between matrixes.

If we analyze more in depth the several AI accelerators used, an important differentiating factor reside in the precision. Because of the fact that most AI workloads can tolerate a lower precision without accuracy loss, many accelerators are designed with very low-precision hardware that can carry out operations even until 2 bits. This design decision can bring many benefits, starting from the obvious memory shrinking ones to lower latencies and energy consumption results. The low-bits accelerators are usually adopted in fields with limited resource like the IoT one, where as in other ones this doesn't represent an issue and precision is much more important.

Another crucial part of an AI accelerator stands in the designed memory hierarchy. In fact the data movement between the memory levels is quite impactful on the overall performance. To tackle this point designers try to employ complicated hierarchies that can optimize memory access patterns to minimize latency and maximize throughput.

The memory hierarchy can be diluted upon 2, 3 or even more levels, and usually is designed to separate memories according to the common DNNs operands, so often it is composed of one or more memory istances dedicated to store the weights of the workload, and one or more instances for the activations. These memories are usually diverse in terms of size and bandwidth, and may be optimized for a certain data structure, this is even more evident in the case of weight memories, in fact there are many different ways to represent a conglomerate of weights, the complication regarding the layout can arise significally in the presence of some fused layers that include many operations, where developers tend to create tailored structures that minimize the data movement.
Finally another important factor in the realization of AI accelerator stands in the types of dataflows supported by the computational array, the dataflows represents how the local registers of MAC units are used and how data is fetched from the buffers, there are 3 types of dataflows:

- **Weight stationary**, where weights are read into the local registers of the computational units and are kept stationary for subsequent accesses. The processing runs as many operations that require the same weights as possible while they are present in the registers. This dataflow is designed to minimize the energy consumption of reading weights and it maximizes convolutional and filter reuse of weights as well. An example of this dataflow can be seen in Fig. 2.4

- **Output stationary**, in this case instead the local registers are used to store the accumulation results of that cell as can be seen in Fig. 2.5, this dataflow is designed to minimize the energy consumption of reading and writing the partial sums.

- Instead of the first two approaches that maximize data reuse of only a specific data type the **row stationary** dataflow tries to optimize data reuse for all data types, by keeping the row of the filter weight inside the local registers of the computational units and using only a cell for the accumulation (Fig. 2.6)

- Finally because of area inefficiency of small registers many designers opt to allocate that space for the overall buffers like in Fig. 2.7, increasing tremendously the traffic by the computational array, this technique maximize the storage capacity.

2.2.3 Heterogeneous platforms

Heterogeneous systems refers to hybrid computing architectures that integrate different types of hardware in a single platforms, they can be composed of various

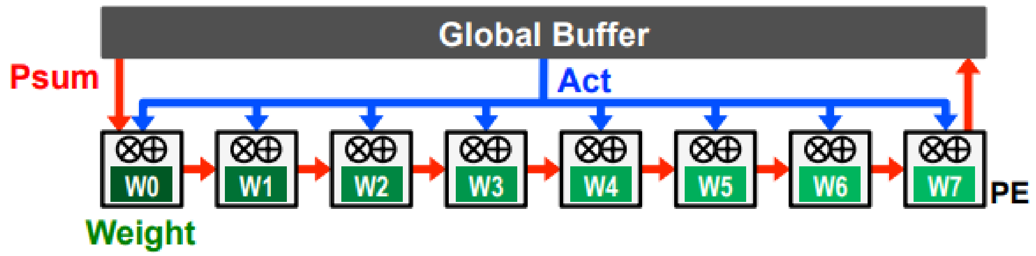

Figure 2.4: Weight stationary

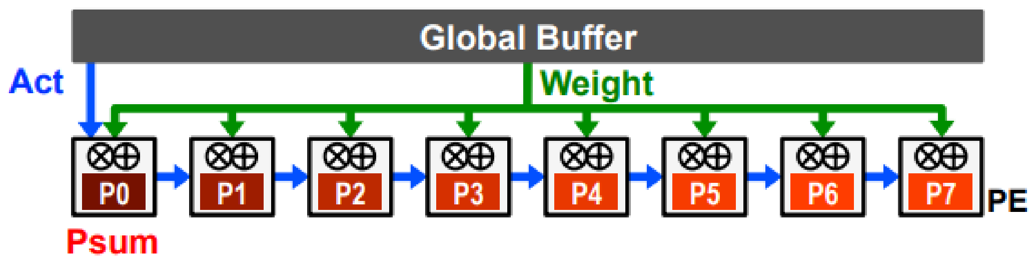

Figure 2.5: Output stationary

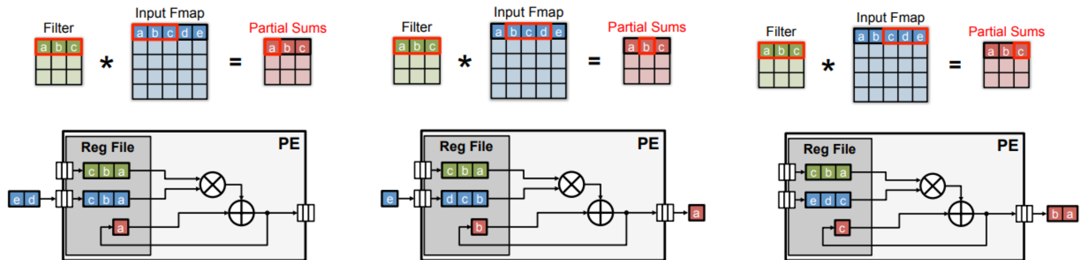

Figure 2.6: Row stationary dataflow in a 1d convolutional layer example.

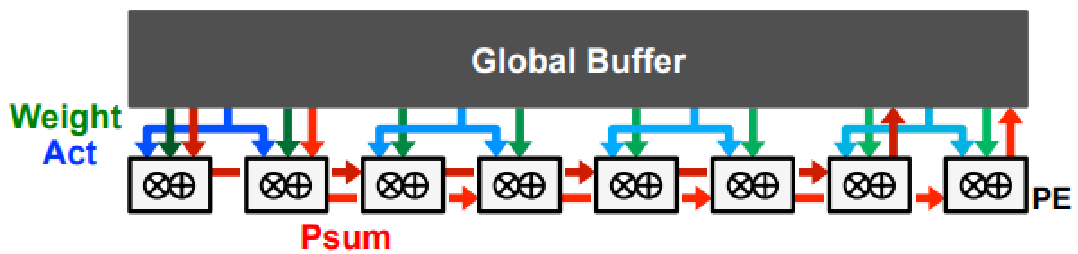

Figure 2.7: No local reuse

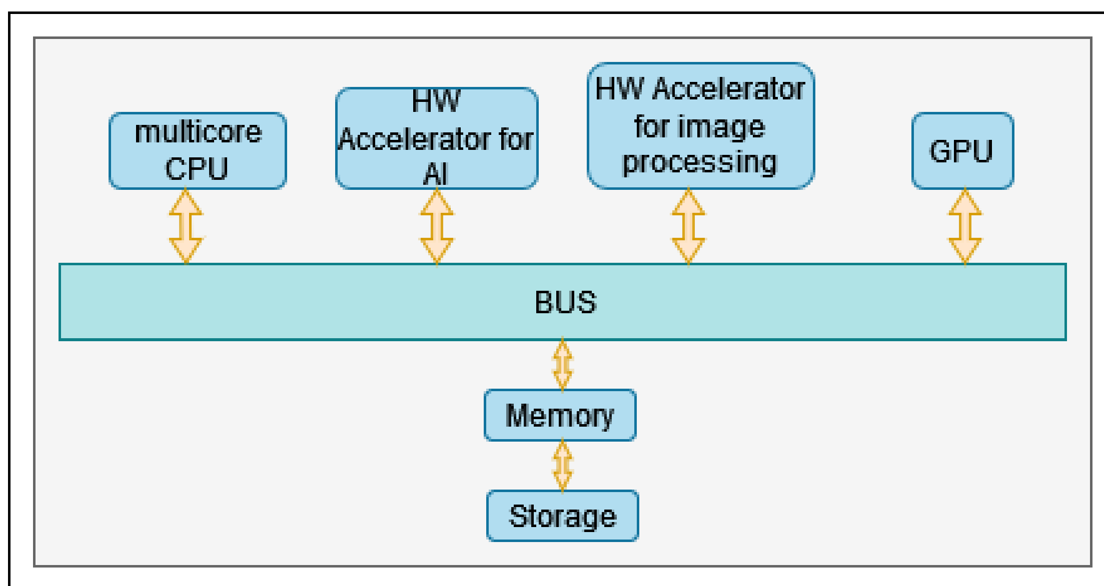

Figure 2.8: An example of a generic heterogeneous platform

processing units. These range from the most notable CPUs to specialized accelerators, a generic example is depicted in Fig. 2.8. These processing units can be used for the same tasks or for different one like video encoding, encryption and many more domain-specific workloads. These platforms can be found in every field of computing, from high end servers till low-power devices.
Incorporating domain-specific aware hardware leads to significant improvements in every notable result, like latency or power consumption, and allows to free up resources of the other general purpose hardware that would be otherwise blocked and stuck performing not optimized operations. The general purpose unit then usually can act just as a collector of the result or a coordinator between these hardware.
With computer architecture hardware following this paradigm and incorporating multiple diverse units, some new challenges arises:

- **Task distribution**: having several hardware that can execute the same task, it is very important to be able to dispatch the current one to the optimal unit, which can be decreed upon performance needs or other user specificed goals. This problem becomes even bigger when we consider bottlenecks, to prevent these obstacles therefore it is crucial to find efficient load balancing algorithms and techniques that can prevent these situation, to utilize fully each unit.

- **Portability**, inherently with this paradigm a big tradeoff comes with efficient portability, writing optimal code for a specific architecture to fully utilize each

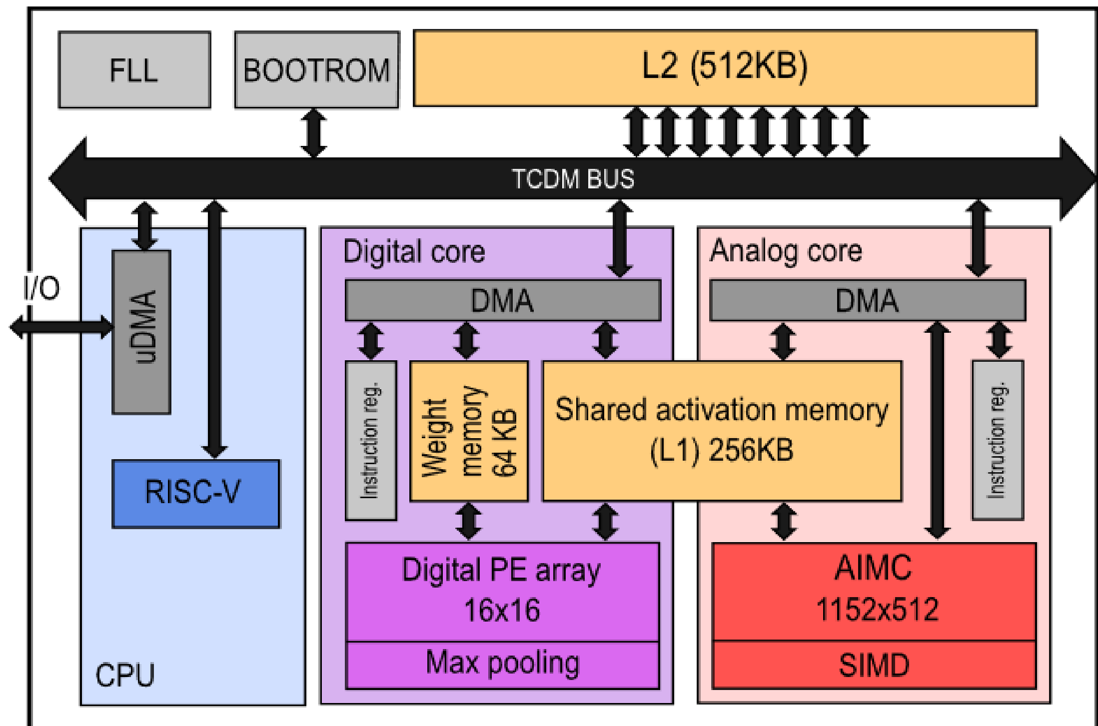

Figure 2.9: DIANA architecture that includes two different accelerator cores.

unit usually comes at the cost of portability across other architectures. Hence usually developers have to find a balance between these two goals.

- Because of the diverse hardware it is often a big challenge designing an efficient way to perform **inter-component communication**, the overhead brought up can be quite high because of bandwidth limitations or other hardware specific ones, thus planning correctly the memory transfers is key to the success.

Despite these challenges Heterogeneous Systems are a revolutionary approach to computing that has been successfull in many areas to exploit the benefits of the diversity introduced, in fact the impact brought by these systems has helped significantly innovation in several growing fields like Artificial Intelligience or Autonomous Driving.

Diana, an example of a heterogeneous platform

DIANA [16] is an heterogeneous device that in a single SoC unifies a simple RISC-V host processor with two different AI accelerator, an analog accelerator and also a digital one.

This architecture grants the possibility to support great performances for a wide variety of Deep Neural Networks tasks since these two accelerators have different strenghts and tradeoffs.
The analog accelerator can bring up an incredible amount of computational parallelism and efficiency but with some loss on accuracy and dataflow flexibility, while the digital is accurate because of its deterministic nature but is not able to reach the analog core in terms of computational density.
These tradeoffs are significant when it comes to the execution, in fact for example because of the low weights reuse granted by a fully connected layer it is challenging to achieve a good temporal utilization on the analog core, making these types of layer more suited for the digital core, while on the other hand layers that require more spatial utilization such as a convolution layer can be limited on the digital core while they represent a good fit for the analog core.
Additionally to these cores to make communication more efficient and also possible between these cores a shared memory hierarchy has been proposed. The memory hierarchy, as we can see in Fig. 2.9, is based on three-levels: the first level is based on registers for each operand, that are included directly on the accelerators. While the second level instead features a scratchpad activation memory of 256KB, that is shared between the digital core and the analog one, and a weight memory of 64KB for the digital core. Finally the third level is dedicated to a single SRAM that has a size of 512KB. This memory is the only one that can be directly addressed by the RISC-V core, without utilizing the DMA. In fact, for example to transfer data to and from the activation memory, the CPU must program the DMA.
The flexibility that is brought up by DIANA can be very useful, especially if the Deep Neural Networks models are trained or compiled through an hardware-aware system that is able to execute each layer on the most convenient core alternative. DIANA flexibility is key also with regards of the supported dataflows, in fact the cores can support the weight stationary dataflow and also the output stationary one.

2.3 Compilers

Compilers are the software component that are needed to translate and optimize a high-level programming code down to machine code or more generally to a lower abstraction level. Increasing the level of abstraction is an important task that facilitate development significantly, improving drastically the efficiency.
The compilation process involves several stages, each serving a specific purpose in translating the code from one level of abstraction to another.
They are used in several fields, and, not just with programming languages: a compiler more generally transforms the input from a level of abstraction to a lower

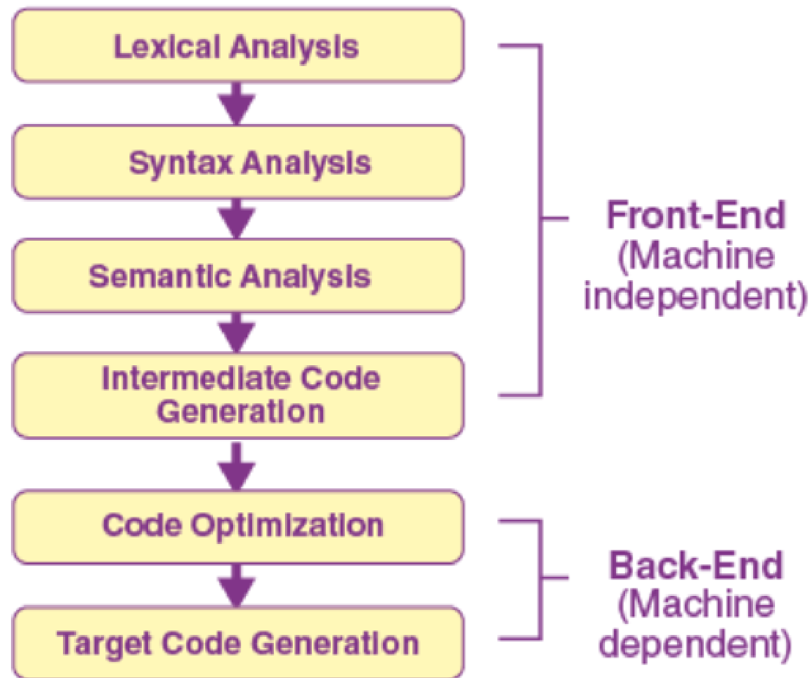

Figure 2.10: Taken by [17], this flow-chart mark the separation from the back-end to the fron-end of a compiler.

one.
An example are AI compilers, also known as neural network compilers. They are specialized tools that play a pivotal role in optimizing and deploying deep learning models. They serve as a bridge between the high-level specifications of a deep learning model and the low-level code required to execute that model efficiently on various hardware platforms.

2.3.1 Compilers structure

Traditional compilers have been broken down into two fundumental modules, the front-end one and the back-end one(Fig. 2.10).
The goal of this division is to make each part more modular and independent, making it possible for a compiler to be more mantainable and portable. This modularity make it possible to generate new language features independently and update in an easier way its syntax. Also multiple languages can all target the same back-end making it possible to support new up and coming languages and also speeding up the prototyping process.
On the other hand instead we can have more back-ends that may want to achieve

different goals(smaller binary size, improved performances etc.). More portability can be achieved in this way since on the verge of the development of new hardware the same front-end can be used with just the need of a new back-end for example. The front-end part of a compiler is the language-specific part. In fact, it specifies the syntax of the compiled language and tipically compiles it down to a more lower level code, known as the intermediate one. During this phase the goal is to generate an optimized intermediate code, through several hardware-agnostic optimization steps.
So the compiler front-end is usually responsible for this set of tasks:

- **Lexical Analysis**, in this step the source code is analyzed as a string of text and is directly divided in tokens, these tokens have a type and are further processed by the following steps, a token can for example represent a constant, a value or for example keywords such as "if", "for" or other ones. The tokens can have additionally a value, this is important for certain tokens that may represent a constant or just to get the name of a variable. The component responsible for the the lexical analysis is usually called Scanner.

- **Syntax Analysis**, during this step one of the core components of the compiler that is called Parser receive as an input a sequence of tokens and after building a Abstract Syntax Tree with them, it starts to analyze the syntax of the code thanks to the set of grammar rules, where each rule represent a set of tokens in a certain order, and each time a sequence of tokens is assimilable to a rule this rule is further analyzed and activated. During this step the Parser validate the syntax correctness of the input.

- **Semantic Analyzer**, this step as the syntax analysis is incorporated onto the Parser, the semantic analysis is the process that verifies that the activated grammar rules make an actual sense, so in this step the Parser captures for example the type errors or other language inconsistencies.

- **Intermediate Code Generation**, the Parser usually have attached to some rules a certain action, and each time this rule is found to be present the action is executed, this action can range from various types, for example it can save the value of the present token that will be useful to a following rule, certain rules will instead generate some code, the generated code can be directly a binary but because of Compiler division usually in this part the target will be a specific Intermediate Representation(IR) such as LLVM [18] or others.

After having generated this intermediate representation the back-end part is finally triggered and processes this code through various optimizations, that are specific on the instruction set, memory layout and in general architecture of the target, to finally produce machine-level code that can be linked and executed.

Finally the back-end part of the compiler is usually responsible for the following steps:

- **Intermediate Code Optimization**, during this step a pipeline of optimizations are executed one after the other, this optimizations are really important because they can improve performance levels drastically and so also the energy consumption.
- **Code Generation**, having the optimized Intermediate Code and a specific back-end hardware target the binary code is generated taking in consideration memory layout and all the relevant target information.
- **Linking**, inside more complex programs that require multiple source files the compiler needs to combine all of them into a single executable file, in this process the compiler resolves references to external functions and libraries, this can be done through dynamic linking or static one.

The back-end target can be any type of hardware, so it can be a typical x86 CPU, an NVIDIA GPU, an ARM CPU, but also more dedicated hardware like a specific accelerator.

2.3.2 AI compilers

The success of Deep Learning heavily realies on an efficient execution of large scale models on a wide variety of devices and hardware, such as CPUs, GPUs, GPGPUs [19], or more specialized accelerators, like the Google's TPUs or programmed FPGAs, to achieve higher inference of models compilers play a very crucial role, and aim to improve the model's performance on memory utilization, latency, or energy consumption. As a consequence, to optimize the execution of AI tasks, AI compilers are designed to enhance the performance, reduce memory usage, minimize latency, and improve energy efficiency of neural networks across a wide range of devices, including CPUs, GPUs, dedicated accelerators, and edge devices.
AI compilers are particularly valuable in scenarios where achieving optimal model execution is challenging due to hardware constraints. For instance, when deploying neural networks on edge devices or Internet of Things (IoT) devices, which often have limited computational resources, AI compilers come into play to adapt and optimize the model for such environments. Likewise, they play a crucial role in accelerating the training process of deep learning models by automating gradient computations during backpropagation, thereby expediting the training phase.
AI compilers are indispensable tools in the field of artificial intelligence, ensuring that deep learning models can be executed efficiently across diverse hardware architectures. They enable the deployment of AI applications on resource-constrained devices while also contributing to faster and more effective model training.

2.3.3 How does an AI Compiler work

First of all, let's introduce the concept of Computational Graphs. Each NN is mapped onto a graph. Many possible graph representations are possible, and these graphs are generated by Machine Learning frameworks (i.e. PyTorch, Tensorflow, etc.).
More specifically a Computational graph is a Direct Acyclic Graph [20] where nodes represent the operations performed on the data, and edges represent the flow of data between these operations. Each node takes input data, performs a mathematical operation on it, and produces an output, which is passed to subsequent nodes in the graph. Nodes in the graph can represent operations such as matrix multiplications, convolutions, activation functions etc..
An AI compiler takes as input an NN computational graph and performs a series of operations to translate it in a lower-level graph, which can be used as input for other tools, or directly deployed on edge devices. The post important executed steps are:

- **Constant folding**, during this transformation the compiler can identify constant nodes in the computational graph whose values do not change during execution so values that can be calculated statically. During this optimization step these nodes are replaced with their pre-computed value, thus relieving the execution with a small trade-off with respect to size and compilation time. This optimization is particularly beneficial when dealing with fixed hyperparameters or pre-trained weights, as it reduces the graph's complexity and accelerates execution.

- **Operator fusion** [21], this is a very powerful optimization technique that helps with reducing memory access overhead and computation time. Several operators usually can be fused together because of their nature, for example a convolution operator can be fused with a following activation function thus resulting in a fused convolution, thus achieving far more efficient inference if applicable.

- **Data Layout Transformations**, there are various ways of storing a certain tensor in a computational graph, ranging from easy ones like row or column major to more complicated ones that may be used by certain edge devices to achieve various back-end optimizations. A compiler that want to target multiple back-ends and giving the possibility to achieve the inference of a single neural network on multiple back-ends(multiple accelerators, or an accelerator that may not be optimal for all the operators) must so be able to optimally reshuffle the tensors. This optimization so converts a computational graph into one that can use better internal data layouts for execution on the targeted hardware back-end.

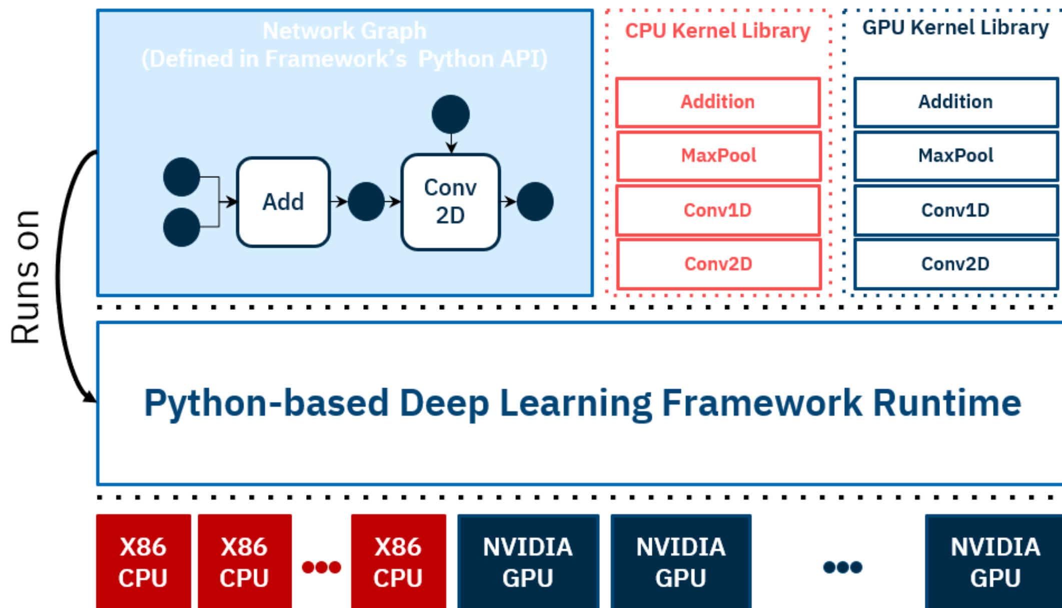

Figure 2.11: Structure of AI compilers targeting general purpose hardware.

- **Auto-Differentiation**, with respect to the other techniques, auto-differentiation is used on compilers that are built to help with the optimization of the training process instead of the inference one. With this technique a compiler is capable of embedding directly on the computational graph the necessary gradient computations. By automatically differentiating the graph, the compiler ensures that gradients can be efficiently computed during backpropagation, resulting in faster and more effective training.

After manipulating the Computational Graph a compiler must be able to offload each node to a target back-end and be able to generate the resulting low level code or another intermediate representation that can still be transformed and optimized. Compilers for Neural Networks are then very hard to build and mantain, because of the continuous ongoing evolution of the field, where new operators and hardware back-ends are introduced quite often.
AI compilers for general purpose hardware, like Tensorflow [22] for example, include a heavy runtime interpreter. This runtime handles the whole compilation process starting from loading the model, optimizing it, and also other operations that manage the correct execution like memory allocation and error management. These AI compilers require vendors to build their own platform-specific libraries that include further optimizations. These libraries are heavily tuned and can be different even for a slightly different architecture built by the same vendor. The typical

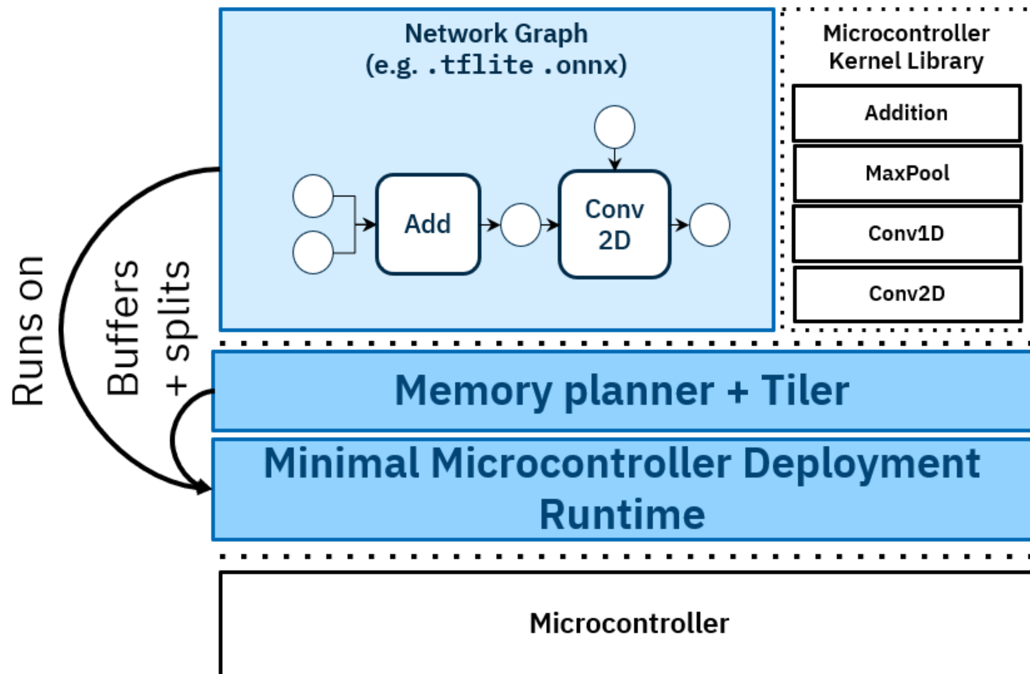

Figure 2.12: AI compiler for MCUs structure.

architecture of these AI compilers can be seen in Fig. 2.11.
When targeting MCUs, we cannot afford to interpret directly a model, but instead we need an offline interpreted one and a set of generated and optimized libraries which can execute directly the network. Therefore AI compilers for MCUs don't include a runtime interpreter. At its place a minimal and lightweight microcontroller runtime is used, which doesn't include an interpreter but instead just executed precompiled layers. These AI compilers have a structure like the one in Fig. 2.12.

Chapter 3

Related Works

An increasing amount of specialized heterogeneous devices for Neural Networks have led to the need of new paradigms to deploy and execute Neural Networks onto this dedicated hardware.

3.1 Compilers for Edge devices

Deploying different wokloads on new hardware may require a lot of manual effort, because current Deep Learning frameworks rely heavily on graph-level optimizations, that are however too high-level to be aware of hardware back-end specific operator-level transformations.
These frameworks are either targeting GPUs, without taking in consideration hardware-specific optimization, or specific hardware SoCs, that are characterized by special optimizations that are not portable on different hardware.

3.1.1 TVM

TVM [3] is one of the first compilers trying to cope with the above-mentioned problems, targetting both GPUs, but also custom hardware platforms. This is possible thanks to its modular structure, which divides the compilation process into several stages. TVM is an open source end-to-end compiler that exploits both graph-level and also operator-level optimizations that has the goal to provide portability across different hardware back-ends.
TVM take as an input a high-level specification from various Deep Learning frameworks such as Tensorflow, PyTorch and others and transforms it into a computational graph representation used then to generate low-level optimized code for different hardware back-ends.

During the computational graph optimization process TVM assess different transformations on the graph nodes such as operator fusion, data layout alterations, quantizations, tensorizations and even costant folding.
After having transformed the graph, TVM searches for different possible code optimizations for a single operator, but instead of using a predefined cost-model or blackbox optimizations TVM to evaluate and guide the schedule search uses a Machine Learning based cost model that can adapt and improve over new data.
This machine learning model represents a great innovation in the field, giving the possibility to learn from the history while not requiring detailed hardware information. An important factor to consider when using a ML model to predict hardware performance is regarding its cost. To be actually useful in fact this overhead must be less than the time to actually measure performance on real hardware. To achieve this speed requirement the model doesn't predict the actual execution time but just a relative order of runtime cost based on memory access count and reuse ratio of each memory buffer at each loop level.
TVM also permits developers to specify some domain-specific knobs in the schedule space through its schedule template specification API, and also if necessary to specificy directly possibile schedules.
The separation of concerns from the graph-level optimizations to the implementation optimization is built upon Halide's idea of decoupling the description of an operator to its computational rules.
Due to the variegated landscape on the hardware side of AI accelerator TVM supports mainly CPUs, server class GPUs and a restricted set of accelerators like some TPUs, leaving up to the vendors the possibility to additionally integrate their hardware back-ends thanks to TVM BYOC.
The BYOC [4] framework partitions the graph into 2 parts: host and accelerator. The partitioning process is executed through a cost based pattern grouping. Each pattern can be declared through simple annotations. The accelerator part is based on 1 or more subgraphs, each one with its own compilation flow. While for the host part the remaining operator will go through the normal compilation process. Instead, the accelerator subgraphs with external function calls on the overall host graph.

3.1.2 DORY

DORY [6] is a compiler built for multi-core architecures, characterized by a multi-level memory hierarchy. DORY makes it possible to handle directly memory transfers between the memory level, even without explicit caching mechanism, by exploiting the Direct Memory Access (DMA) [23]. Noteworthy, architectures that don't include caches and handle memory transfers exclusively through software are becoming the main ones to execute efficiently DNNs at the edge. This is due to the

big energy consumption savings obtained by the lack of caches. Mapping a Deep Neural Network on devices at the extreme edge of the Internet-Of-Things require explicit DMA-based memory-transfers between different levels of the memory hierarchy. This is due to the presence of scractchpads, that helps with energy efficiency.
Thus requiring very topology-dependent tiling, Dory acts then as an automatic tool to deploy Deep Neural Networks on these edge devices. Dory targets three-levels memory hierarachy architectures such as DIANA.
Tiling on Dory is solved in two steps abstracted as a Constraint Programming problem, that starts with the tiling problem of the L3-L2 memories where the goal is to store in L3 weights and activations, and continues with the L2-L1 tiling that tends to maximize L1 memory utilization. Additionally some topological constraints can be defined and imposed for each layer. Dory is then used to also generate the necessary C code able to manipulate memory transfers and computation phases. This code generation is not completetely platform-agnostic, but it can anyway be generalized to almost any computing core with a three-level memory hierarchy.
Note that the tiling dimensions produced by DORY are critical hyperparameters for the efficient deployment of NNs on the target platform. From [cite], we can notice that uncorrect tiling dimensions can cause loss of performance of up to 80%.

3.2 Compilers for heterogeneous systems

3.2.1 HTVM

HTVM [5] is a compiler framework targeted to maximize the utilization of heterogeneous hardware and minimize data movements. This compiler stacks is built on top of TVM, integrating onto it Dory [6]. This project permits the deployment of standard Machine Learning suites on DIANA, an SoC composed of two different AI accelerator cores
The dispatcher mechanism on HTVM is composed by a pattern matcher and some accelerator-aware rules. The pattern matcher is used to settle the plausible layers that can be offloaded directly to the dedicated accelerator.
Additionally the accelerator-aware rules are used to check more specific pattern related details, like stride size for example, and they are used to determine if the accelerator can execute the selected pattern.
Therefore, for accelerable layers, the HTVM framework(Fig. 3.1) exploits DORY for the deployment process. After having generated low level code for each supported layer HTVM integrates also DORY runtime library as well as TVM ones. This flow, which is depicted in Fig. 3.1, follows the one advised by the TVM BYOC proposal and it exploits DORY as BYOC to generate tiling and code for accelerable layers. BYOC was developed to help towards a unified abstraction layer that enables

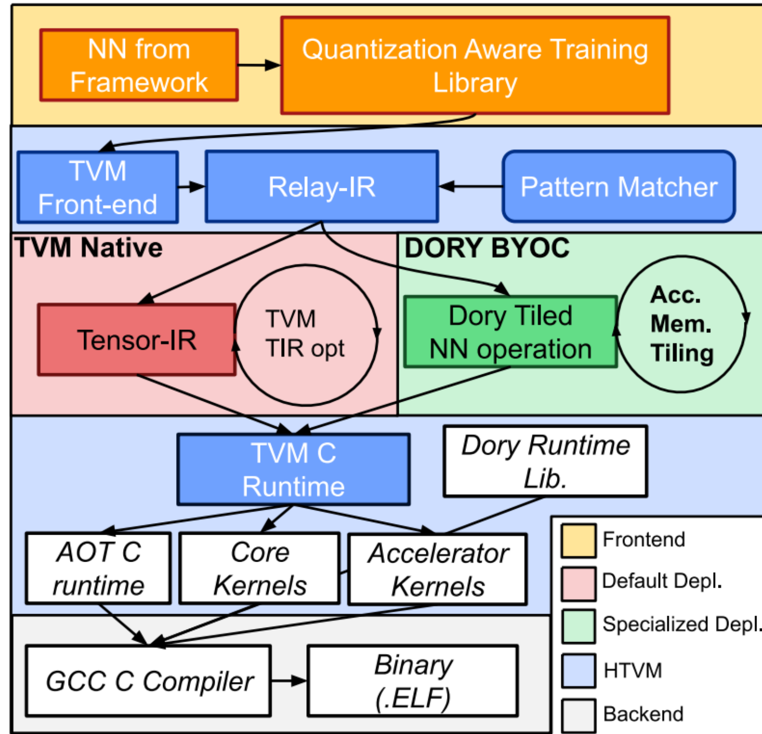

Figure 3.1: Figure taken by [5], which introduces HTVM and explain how to fuse DORY inside TVM throughout the BYOC infrastructure.

a seamless integration between the typical high level Deep Learning frameworks and hardware back-ends. This layer is therefore responsible for the usual shared activities of every hardware back-end, finally relieving accelerator vendors from the stress to build a full fledged compiler stack and focus only on the specific back-end kernels to fully utilize the custom hardware blocks.

3.3 ZigZag

Deep Neural Networks deployment presents a two-fold problem: on the one side, one issue is to find the optimal mapping of a neural network onto a predefined hardware. On the other side, given a network, the problem is to define a new hardware block to execute it efficiently.
ZigZag was built with these two topics at the front. ZigZag [7] is a complex Design Space Exploration tool that tackles the most common issues solving them in a layer by layer pattern.
This design space representation is generally based on a memory centric approach

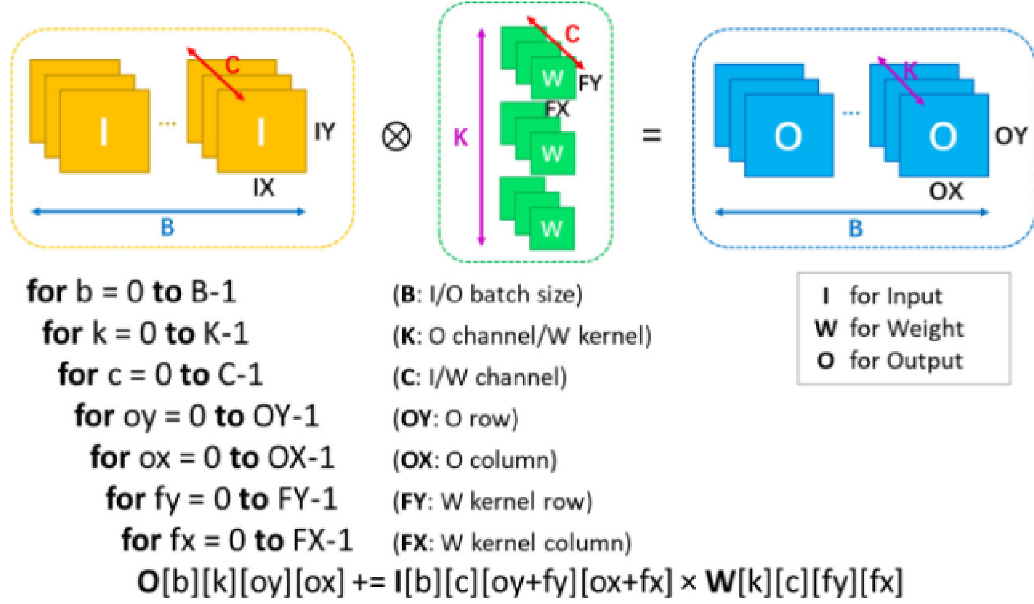

Figure 3.2: Taken by [7]. Workload representation on ZigZag.

and combines three main aspects of the layer: the algorithm, the hardware architecture and finally the mappings that bridges the algorithm to the hardware.
The algorithm of the layer is represented on ZigZag with a series of loops, the relationship between them, the operands, the precision of the operands and finally the equation and the name of the algorithm used. Fig. 3.2 display an example for a convolution, with the definition of the operand, the equation that relates them and also the necessary nested loops.
The hardware architecture instead is formally defined through a memory hierarchy and the computational array definition(MAC units), where the memory hierarchy incorporates each memory unit that is represented thanks to the size and bandwidth of that instance and through a four-way connection to the upper and lower memory levels. The memory instance representation encapsulates also which operands it is destined to.
Finally the final mapping is represented through a set of ordered for-loops, and for each operand the chosen memory allocation given to the loops. As in Fig. 3.3 the memory allocation process supports unbalanced memory hieararchies and also uneven mapping, by being independent on an operator level.
ZigZag is organized as a set of stages that form a sort of a pipeline, where each stage transform the given input and can feed it into another stage to finally generate a certain output, this structure is quite armonious and give the possibility to other developer to implement new stages and so contribute in a simple way.

Weight			Input			Output		
DRAM	K	32		K	32	DRAM	K	32
	C	12	Activ.	C	12	Activ.	C	12
	OYu|FYu|OYu	13|5|2	Mem	OX	13	Mem (I/O)	OX	13
	OX	13	(I/O)	C	2		OYu|FYu|OYu	13|5|2
	C	2		OX	2		C	2
	OX	2		OYu|FYu|OYu	13|5|2		OX	2
	FX	5		FX	5		FX	5
Reg File	C	2	Reg File	C	2	Reg File	C	2
(W)	K	8	(I)	K	8	(O)	K	8
MAC Level			MAC Level			MAC Level		

Figure 3.3: Taken by [7]. Mapping representation on ZigZag.

ZigZag has three main features that contribute to help with the already discussed problems:

- **Mapping Engine**, the mapping search can be done both spatially and temporally with ZigZag, the temporal search is specifically delivered through the LOMA engine that is based around the concept of Loop Prime Factors, this engine can cover up a great part of the search space without losing optimality.
- **Cost Estimation**, with an analytic model ZigZag can estimate both in terms of energy and in terms of latency perspective the performances of a certain mapping on the defined architecture, the used analytic is supposedly correct in a 10% range, the cost model can adapt to different types of situations, for example it is capable of guessing if there are double-buffering opportunities.
- **Architecture Generator**, this tool explores automatically the large search space around the memory hierarchy design.

3.3.1 Mapping Engine

The mapping of a network is divided onto a spatial mapping and a temporal mapping, the spatial mapping refers to the assignment of each part of the analyzed layer to a set of the computing units of the dedicated hardware, since AI accelerators have a grid of MAC units in practical terms the spatial mapping is done to unroll a certain component of the network(loop) on a given dimension of the grid.
The spatial mapping plays then a crucial role to optimize the performances of dedicated AI accelerators, since it is defining if the computational units are all

used fully and optimally, if the spatial mapping is done uncorrectly we can achieve therefore big losses.
On ZigZag the spatial mapping can either be done through an exhaustive search, which can be limited thanks to user defined values, or thanks to 2 possible heuristic search options, where the first one is based upon the pruning of data reuse symmetry and the second one on the pareto surface of data reuse.
The temporal mapping instead stand for the ordering and sequencing of the operations that must be done on the layer, since ZigZag make use of the loop view for each layer the temporal mapping is then the ordering of these loops. ZigZag exploits the LOMA engine for the temporal mapping search.
The temporal scheduling of a given workload is one of the main factors of the energy and perfomances costs for a given accelerator. There are various Design Space Exploration frameworks that are trying to explore the vast search space around this problem, but currently most of them end up being quite slow, because of the exhaustive search, or they can't find the optimal solution.
LOMA [24], instead of the alternatives that usually resorts to Genetic Algorithms or Reinforcement Learning is performing an exhaustive search that exploits the nature of the nested for loops, pruning away equivalent mappings. LOMA allows to conclude a fast and lightweight search by generating partial permutations for each loop type separetely. This speeds up the process significantly and the memory required to hold the generated permutations decreases as well.
This DSE engine processes the given layer by means of Loop Prime Factors, which represent the division of the various loop iterations in smaller loops: specifically, each dimension is divided in its prime factors and a minimum of one loop is generated for each of them. For instance, if we have K =14, the minimum number of loops that we will have will be 2, with K1 = 7 and K2 = 2. The way in which each loop is executed and the position in memory of its array strongly impacts the memory access pattern and the energy consumption.
Specifically the division of a certain loop iteration is done by factorizing that value into its smallest possible prime factors, and acording to the initial size this process may increase greatly the total number of schedules that needs to be assessed. To tune down the number of generated orderings, LOMA gives the user the possibility to tune the algorithm by setting a limit on the number of loops that can be generated by the factorization. This limitation is executed through an algorithm that scans the loops generated by the complete factorization, and for each iteration it checks for the dimension with the most factors. Given this dimension LOMA merges its smallest factors by multiplying their values. This is done until the number of loops reaches the user defined limit.
After the LPFs generation the second step in the LOMA engine pipeline resides in the loop ordering process, that is done through the generation of the permutations of these LPFs. Instead of generating through a brute force approach all the

permutations, which would be unfeasible on various parts, LOMA generate partial permutations for each loop type separetely, speeeding up significantly the generation reducing the sheer size of memory required to keep track of generated permutations. These partial permutations [25] are generated through a multiset, that automatically takes into account the LPF multiplicities to avoid generating equivalent permutations. After the partial permutations generation, the resulting multisets are recombined by looping through all of them one by one across loop types, thus producing a complete ordering. This whole process ends up being quite fast with respect to other DSE tools.
Each ordering is then fed onto the memory allocation step. With LOMA this is performed passing through every operand memories, from bottom to top. For non shared-memories the allocation process objective is to fill the memory. While for shared memories all possibilities have to be considered. LOMA then chooses as the optimal memory allocation the one with the highest data-reuse value. This value is based on the relevancy of the loops for the operator. The allocation of the loops to the different memory instances causes the tiling of the layer, the computation of it will in fact occur only on the innermost memory level. Finally LOMA support both even and also uneven mapping. Even mapping defines the case where each loop is associated to the same memory level for each operand. This is more limiting, specifically to more complicated memory hierarchies but it is crucial for some hardware. Instead uneven mapping allow for a fully flexible scheduling in which all the operand data of a specific loop can be stored at different levels in the memory hierarchy, and as a result, the scheduler has to allocate the LPFs for each operand individually to their best memory level.

3.3.2 Cost Evaluation

ZigZag final stage of the pipeline is where each mapping is being evaluated, starting with a technology independent loop information extraction that leads to an integration of these found values with the technology charateristics.
The cost evaluation introduces a loop relevancy principle, which indicates if a certain loop is relevant for an operand or not, or if it is actually just partially relevant. Not every loop is in fact relevant for all operands: if we take in consideration a 2d convolution we can see that through the nested-loops view it is carried out in a 7d computing space, where each operand is on a 4d space, meaning that not each one of the 7 dimensions of the convolution is impactful for one of the 4 dimensions of the operand.
The relevancy is a simple concept for most operands, but for the input it is a more complicated situation: in this case there is the necessity to declare some dimensions as partially relevant, since the inner dimensions of the input depends on them through an equation. For example, in the case of a 2D convolution the input

dimensions are related with the weight and output ones through the equation 2.1. While looping through a relevant dimension the need of new data fetch or data generation is present, while insted looping through a not relevant one, or some specific partially relevant cases, some data reuse opportunities arise.
Based on this loop relevancy principle many loop informations are extracted, such as the total data reuse factor, the sizes for memory levels and operands, the required bandwidths and other values. Finally all of gathered informations are used to evaluate mainly the energy that should be consumed and the latency. Energy is estimated by accounting the number of active MAC units with the number of idle ones and reuniting this ratio with the energy that should be consumed for each operation by a single MAC unit. Instead latency evaluation is highly dependent on the spatial under-utilization and temporal under-utilization as well, where the first one comes from the mismatch of the spatial mapping with the MAC array sizes and the latter one is due to memory bottlenecks that are caused during computation by an insufficient bandwidth in some memory level.

3.4 Deployment on Edge Devices: TinyML leader board

The deployment of DNNs over edge devices is a complex task, currently the reference benchmark is represented by the TinyML suite [26]. The rankings over this suite are lead by Greenwaves Technologies through their GAP9 SoC and a proprietary and manually tuned compilation toolchain.
The main competitors over the leader board end up being the following:

- **GreenWaves Technologies**, which maximizes the capabilites of the GAP9 platform through the GAPFlow compilation stack. Overall this platform leads on both the performance aspect and the energy one, achieving a speed-up of over 5 times, w.r.t to the other competitors, for the first concern and an improvement of over 2 times for the latter. These gains appear over all the benchmark suite tests.

- **Syntiant**, that sits second overall in the leader board, they achieves impressive results especially considering the low frequency of the utilized hardware.

- **STMicroelectronics**, which exploits their ARM CPUs through an optimized ARM kernel, the lack of a dedicated accelerator is evident tho, both in terms of performances and energy consumption.

- **Silicon Labs**, even though they are using an SoC comprising an AI accelerator, their results ends up being inferior to STMicroelectronics ones.

Noteworthy, outside of the leader-board HTVM represents a close competitor to GAPFlow. HTVM offers more flexibility and the loss in terms of performances is not as big. In fact deploying the TinyML suite on DIANA through HTVM obtains results worthy of the second position on the rankings on average as is shown in Tab. 3.1.

Network	GreenWaves GAPFlow	HTVM	Leaderboard runner-up
DSCNN	0,73	1,92	1,48 (Syntiant)
MobileNet	1,73	6,75	4,1 (Syntiant)
ResNet	0,95	1,21	5,12 (Syntiant)
ToyAdmos	0,27	0,36	1,82 (STMicroelectronics)

Table 3.1: TinyML leader board including HTVM results. Latency is in ms.

Chapter 4

Material & Methods

In this chapter we'll dive deep into the objective of the thesis, getting an initial idea of the tasks and the benefits of the contributions. Then we'll see more in detail how they have been pursued and also the challenges that arose during the thesis. To give a brief introduction to the following section the thesis aims to be a research work about the compilation process for a Deep Neural Network on edge heterogeneous hardware, resulting into an integration of ZigZag into TVM, following the BYOC approach. This work has been benchmarked on the Diana SoC, a platform designed by a team of researcher in the Leuven university, which encopass a RISC-V core and two AI hardware accelerators.

4.1 Objective

The fundamental foundation for the remainder of this work is HTVM, and consequently, TVM. Due to the extensive work already conducted on these projects, they can serve as a comprehensive starting point. The thesis aims to build upon this foundation by incorporating ZigZag as the internal scheduling optimization engine.
The objective is thus the introduction of ZigZag to execute the supported layers by the digital accelerator core of DIANA directly in it. The idea is to replace the current BYOC branch, DORY, using ZigZag to produce more efficient kernels. During the thesis the DIANA board has been used as an example of an edge heterogeneous system, but the goal of the thesis is to simplify the compilation process for a vast set of architectures through a simple update of the back-end library used. By choosing ZigZag there is a set of features that are introduced and are not available into HTVM internal engine, DORY, and can inherently bring some advantages. The benefits of using ZigZag are the convenient loop ordering representation, the capability to generate unevenly mapped schedules and the

possibility to have a tighter software and hardware design due to the inherent hardware design search feature of ZigZag.
ZigZag execute an exhaustive search where only the intrisically equivalent schedules are pruned, while the rest are all evaluated by ZigZag. This approach makes it possible for ZigZag to guarantee optimality. Additionally the search space of ZigZag can be limited, to fasten up even more the process, with a user defined limit over the number of LPFs that can be generated, with a minimal impact in optimality. Noteworthy, the uneven mapping support in ZigZag enriches the memory allocation capabilities by a wide margin with respect to DORY. Finally, the generated schedules are evaluated by ZigZag own cost model and the one which achieves the better results over the chosen metric is chosen. As a consequence, the accuracy of ZigZag's cost model is key and the better it gets also the schedules generated will be better. The targeted metric by ZigZag can be either latency, energy consumption or finally a combination of these 2. During this thesis the goal has always been latency optimality.
Instead Dory applies an iterative approach for the tiling generation searching in a space limited by a set of constraints defined per each architecture. Dory ultimately outputs directly the low level C code of the layer, while ZigZag generates a temporal mapping with its corresponding cost evaluation. The temporal mapping is defined as the set of loops creaeted by the loop prime factors search and their optimal ordering, as well as the ordering the memory level associated with each operator for each loop is also present.
HTVM includes a virtual environment where all the necessary dependencies to build a whole application that targets the DIANA system are included. The main dependencies are the following:

- **DORY**, which is used for the code generation process
- **Dory-Hal**: a library of kernels specialized for the DIANA SoC.
- **PULP SDK**: the toolchain used to compile code for DIANA.

The HTVM process that leads up to the compilation is initiated by a TVM core that builds up the Relay Intermediate Representation of the model, and then this new representation is fed to a new component, the driver that serves as a wrapper for the partitioning, the compilation and the execution steps. The driver have then to generate some C code for all the partitioned network layers and some C code that will tie all these layers together. The partitioning system checks the validity of each layer and offloads the compilation of that layer to the defined code generation function, that in the presence of an accelerable layer extrapoletes the informations needed by Dory and though it generates low-level C code for that layer, that will be used by the TVM runtime.

The main contribution of this thesis will then be a modification of the code generation function with the goal of exploiting ZigZag instead of DORY, leaving the partitioning system untouched. This process can be broken down essentially into these tasks:

- To offload the temporal mapping search to ZigZag it is required to define the **workload** in the supported definition.
- **Code Generation**, given that ZigZag doesn't generate directly C code like Dory this additional step is required.

4.2 Description of the workflow

Inside this section the workflow will be analyzed in a brief way initially and it will be analyzed more in depth in the parts that have been modified. The workflow ends up being pretty similar to the HTVM one: until the code generation part it is quite identical.
The input is a TVM Relay IR module, which can be generated through a set of utility functions already available in HTVM. These functions are used to build with ease layers that are accelerable by DIANA. For example to generate a convolutional layer we use the relay_soma_conv2d one. This function can be seen in Fig. 4.2. It creates a TVM relay IR module of a fused convolutional layer.
Having generated the TVM module the driver takes care of the partitioning and gathers all the resulting C code in a single place, that will be later on compiled down to executable by GCC, this final blob, tied with the C code, represents the output of the whole process, a simple representation of the proposed workflow can be seen in Fig. 4.1.
Checking out the code generation part there are some steps that are needed to be able to utilize ZigZag properly. In fact the ZigZag API requires 3 inputs, the workload definition, the memory links and operands, and finally the accelerator definition. Out of these 3 inputs only the first is dynamic, while the rest are static and don't need to be generated each time, but it's better to set them as objects that can be imported easily. The accelerator object is built with a set of ZigZag classes, and in the end it includes the memory hierarchy structure and the types of memory instances used, and the definition of the computational array structure, so the organization of the MAC units.
For the last object instead the definition is similar to a JSON object that maps for each layer type the loops that would be associated to the operational array. The main contributions brought by this thesis are then 2 interfaces, one that can parse through a TVM module and generate a ZigZag workload. And the latter is an interface that, thanks to output of ZigZag, can generate C code for the targeted architecture.

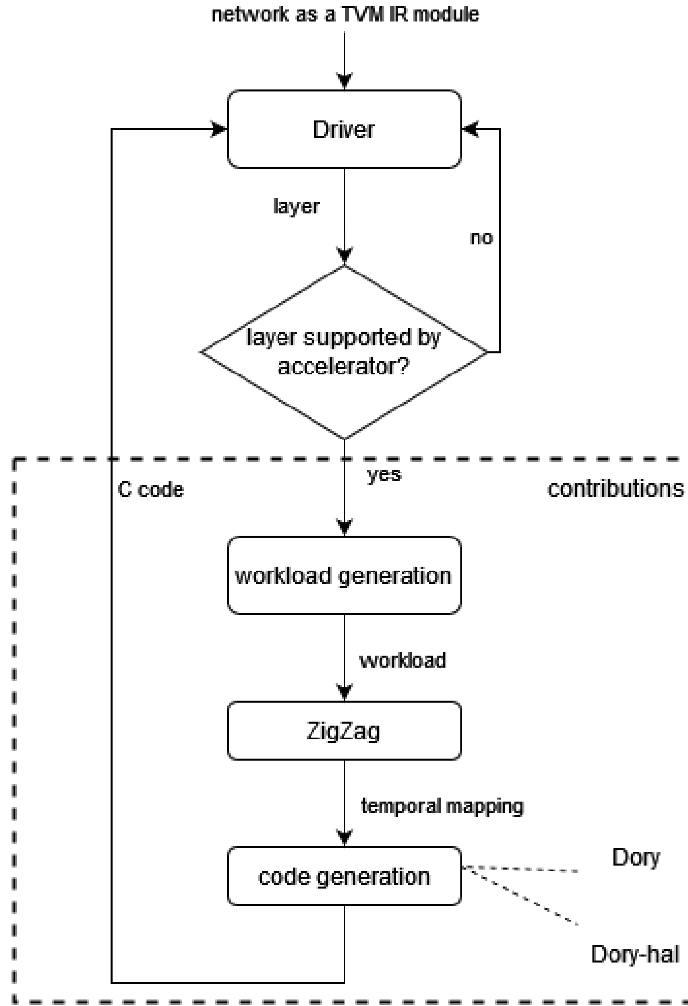

Figure 4.1: Workflow of the compilation process.

4.2.1 TVM-to-ZigZag: Workload generation

To generate a suitable workload for ZigZag there are 2 main approaches, this is due to ZigZag that accepts as a workload input either a dedicated JSON object with specific parameters, or a more standard ONNX model. Initially after some research we found that TVM internally includes an IRModule to ONNX converter and tried to exploit it, but unfortunately it was instantly clear that this product was not completely functional for the targeted use case since it didn't include a slate of operations that are necessary like the right shift, casting and other ones.

After this initial attempt and further careful evaluation of both approaches we

```python
# define weights and bias variables
weights_name = layer_name + '.weights'
bias_name = layer_name + '.bias'

# define relay input vars
w = relay.var(weights_name, relay.TensorType(w_value.shape, w_value.dtype))
b = relay.var(bias_name, relay.TensorType(b_value.shape, b_value.dtype))

# define weights and bias values in params
params = {weights_name: w_value, bias_name: b_value}

# define operations
x = relay.op.nn.conv2d(input_tensor, w,
                       strides=strides,
                       padding=padding,
                       groups=groups,
                       out_dtype=b_value.dtype)
x = relay.op.nn.bias_add(x, b)
x = relay.op.right_shift(x, relay.const(shift_bits))
# Optional: ReLU
if act:
    x = relay.op.clip(x, a_min=0, a_max=127)
else:
    x = relay.op.clip(x, a_min=-128, a_max=127)
x = relay.op.cast(x, 'int8')


return x, params
```

Figure 4.2: This function is a utility used to build a 2d convolution that is supported by the digital accelerator core of DIANA.

targeted the first one, because it guarantees an easier way to mantain the product, due to the fact that the workload JSON representation is almost the same as the one used internally, so it is easier to introduce new functionalities and to insert support information. Another concern was also about the support level of the ONNX models on ZigZag, since the support have been brought up quite recently and not fully tested.
The concerns raised necessitated the development of a new parser. This parser is designed to analyze the TVM IR Module and, for each part of the layer, which can be fused, gather relevant information and incorporate it into the workload object. The process for extracting information for each operation begins with a visiting function. Starting from the IRModule, this function traverses through a set of TVM

```python
class ZigZagWorkloadParser:
    def __init__(self,mod):
        self.mod=mod
        self.visit_router={
            "nn.conv2d": self.visit_conv_2d,
            "cast": self.visit_cast,
            "right_shift": self.visit_right_shift,
            "clip" : self.visit_clip,
            "nn.bias_add" : self.visit_bias_add,
        }
        self.supported_operators=tuple(self.visit_router.keys())
        self.workload=dict()
        self.index=0
        self.callsindexes=dict()
        self.calllist=[]
        self.var_and_consts=dict()
        self.cme=None
        self.energy=0
        self.latency=0
        self.attrs=dict()
        self.for_loops=[]
        self.type=None
        self.accelerator=None
        self.mapping=None
```

Figure 4.3: Definition and initialization function of the object that is responsible for the generation of the workload and then also the next steps

Call objects. These Call objects represent operators along with their associated attributes and inputs. It's important to note that these inputs can either be constants or other Call objects.

The logic behind this process is illustrated in Fig. 4.4 on the visit function's side.

So the parser traverse the IRModule Calls from the first up to the last one. And for each call the parameters are analyzed to save up the costants and then if the operator is supported the function that will visit that operator details is called. For example in case of a 2D convolution, the specific function will be called and it will store all the information that is needed, like stride, dilatation and shapes. In Fig. 4.5 the function adepeted for the convolution is depicted. The parser class,

```python
def visit(self):
    # get flow of functions to do them in order, to do so prepend
    # TODO : need to check if op.op is fine or in general how to get the nested call of a call
    call=self.mod.body.op.body
    while(call is not None):
        self.calllist.append(call)
        if len(call.args)>0 and isinstance(call.args[0],tvm.relay.Call):
            call=call.args[0]
        else:
            call=None
    self.calllist.reverse()
    var_and_consts_not_unrolled=dict()
    var_and_consts_unrolled=dict()
    for c in self.calllist:
        # fit variables and constants
        for a in c.args:
            if isinstance(a,tvm.relay.Var):
                ## this can be either a constant or the input still so let's check the real type
                if isinstance(self.mod.body.args[len(var_and_consts_not_unrolled)],tvm.relay.Var):
                    var_and_consts_not_unrolled[a.name_hint]=self.mod.body.args[len(var_and_consts_not_unrolled)]
                else:
                    var_and_consts_not_unrolled[str(c.op)+f'.param.{sum([str(c.op) in k for k in var_and_consts_not_unrolled.keys()])}']=\
                    self.mod.body.args[len(var_and_consts_not_unrolled)]
            elif isinstance(a,tvm.relay.Constant):
                var_and_consts_unrolled[str(c.op)+f'.param.{sum([str(c.op) in k for k in var_and_consts_not_unrolled.keys()])}']=a
        self.index+=1
        self.visit_call(c)
        self.callsindexes[c]=self.index
    self.var_and_consts={**var_and_consts_not_unrolled,**var_and_consts_unrolled}

def visit_call(self,call):
    if str(call.op) not in self.supported_operators:
        raise NotImplementedError(
            f"Currently the operator {str(call.op)} is not supported."
        )
    else:
        self.visit_router[str(call.op)](call,call.attrs)
```

Figure 4.4: These functions are used to follow through the operations that are part of a fused layer, and will use a router to call a function dedicated to extrapolate every information useful for that operator.

that is shown in Fig. 4.3, includes an object that maps each operator to its own function. This structure eases the process to support new operators.

Given the workload definition, accelerator object(Fig. 4.6) and also the object that includes the spatial mapping and memory link operation(Fig. 4.7) all the inputs required by ZigZag are built and the only thing left is to call the ZigZag API.

This API will generate an object of ZigZag's CostModelEvaluation class, which contains all the useful information about the latency costs and the energy costs and it encapsulates also the associated temporal mapping and spatial one. The temporal mapping obtained by this CME has the structure shown in Fig. 4.8 and is later on exploited to generate a more complete data structure containing also the memory level for each operand and other information through the process in Fig. 4.9.

```
def visit_conv_2d(self,call,attrs):
    '''we want to obtain something like this
    {  # conv1, stride 2
        'operator_type': 'Conv',
        'equation': 'O[b][k][oy][ox]+=W[k][c][fy][fx]*I[b][c][iy][ix]',
        'dimension_relations': ['ix=2*ox+1*fx', 'iy=2*oy+1*fy'],
        'loop_dim_size': {'B': 1, 'K': 64, 'C': 3, 'OY': 112, 'OX': 112, 'FY': 7, 'FX': 7},
        'operand_precision': {'O': 16, 'O_final': 8, 'W': 8, 'I': 8},
        'operand_source': {'W': [], 'I': []},
        'constant_operands': ['I', 'W'],
    }'''
    def get_io_from_layout(layout,data):
        if layout=='NCHW':
            n=data[0]
            c=data[1]
            h=data[2]
            w=data[3]
        else:
            #layout is nhwc
            n=data[0]
            c=data[3]
            h=data[1]
            w=data[2]
        return n,c,h,w

    itype=call.args[0].type_annotation.shape
    iprec=call.args[0].type_annotation.dtype
    wtype=call.args[1].type_annotation.shape
    wprec=call.args[1].type_annotation.dtype
    otype=call.checked_type.shape
    i_n,i_c,i_h,i_w=get_io_from_layout(attrs.data_layout,itype)
    o_n,o_c,o_h,o_w=get_io_from_layout(attrs.out_layout if attrs.out_layout!='' else attrs.data_layout,otype)
    padding=attrs.padding
    strides=attrs.strides
    dilations=attrs.dilation
    groups=attrs.groups
    if itype[0]!=otype[0]:
        raise NotImplementedError(f'Input batch size is {i_n}, while output batch size is {o_n}')
    dimension_relations=[f'ix={int(strides[0])}*ox+{int(dilations[0])}*fx',f'iy={int(strides[1])}*oy+{int(dilations[1])}*fy']
    kernel_size=attrs.kernel_size
    loop_dim_size={'B' : int(o_n), 'K' : ceil(int(o_c)/int(groups)), 'C': ceil(int(i_c)/int(groups)), 'OY':int(o_h) ,'OX': int(o_w),
    'FY': int(kernel_size[0]),'FX':int(kernel_size[1])}
    operand_precision={'O': self.get_bits('int8'), 'O_final': self.get_bits('int8'), 'W': self.get_bits(wtype), 'I': self.get_bits(itype)}
    padding={'IY':(int(padding[0]),int(padding[2])), 'IX': (int(padding[1]),int(padding[3]))}
    pr_loop_dim_size = {'IY': int(i_h) , 'IX': int(i_w)}
```

Figure 4.5: This function extrapolates all the useful information for a 2d convolution.

4.2.2 ZigZag-to-TVM: Code generation

For the code generation part, the proposed approach involves the definition of a template that will be transformed onto the resulting low-level source code, written in ANSI C. The template is then analyzed and used by a Python library called Mako.
The Mako library permits through a simple syntax to generate something based on the input given to its API, Mako allows to use also directly Python code inside the template, helping noticeably with the effort required to build a low-level source code.
The code generation part can be divided in two steps: the first one involves the compilation and execution of a layer into the host computer, so executing a convolution directly on x86 hardware. The second step instead targets the DIANA architecture. To facilitate the second step the template was built with a set of

```python
    memory_hierarchy_graph.add_memory(
        memory_instance=l2,
        operands=("I1", "I2", "O"),
        port_alloc=(
            {"fh": "w_port_1", "tl": "r_port_1", "fl": None, "th": None},
            {"fh": "w_port_1", "tl": "r_port_1", "fl": None, "th": None},
            {
                "fh": "w_port_1",
                "tl": "r_port_1",
                "fl": "w_port_1",
                "th": "r_port_1",
            },
        ),
        served_dimensions="all",
    )

    return memory_hierarchy_graph


def get_operational_array():
    """Multiplier array variables"""
    multiplier_input_precision = [8, 8]
    multiplier_energy = 0.04
    multiplier_area = 1
    dimensions = {"D1": 16, "D2": 16}  # {'D1': ('OX', 16), 'D2': ('K', 16)}
    multiplier = Multiplier(
        multiplier_input_precision, multiplier_energy, multiplier_area
    )
    multiplier_array = MultiplierArray(multiplier, dimensions)

    return multiplier_array


def get_dataflows():
    return [{"D1": ("OX", 16), "D2": ("K", 16)}]


def get_core(id):
    operational_array = get_operational_array()
    #get the memory hierarchy, from the l2 to the register level
    memory_hierarchy = get_memory_hierarchy(operational_array)
    dataflows = get_dataflows()
    core = Core(id, operational_array, memory_hierarchy, dataflows)
    return core

def generate_accelerator():
    """Generate the DIANA accelerator"""
    cores = {get_core(1)}
    global_buffer = None
    acc_name = 'Diana'
    return Accelerator(acc_name, cores)
```

Figure 4.6: Accelerator definition, that is built through the ZigZag classes, it includes the MAC array and the memory hierarchy.

```
if layer_name=='conv_2d':
    ox_val=loop_sizes['OX']
    k_val=loop_sizes['K']
    if ox_val>16:
        for divval in [_+1 for _ in range(16)][::-1]:
            if (loop_sizes['OX']%divval)==0:
                ox_val=divval
                break
    if k_val>16:
        for divval in [_+1 for _ in range(16)][::-1]:
            if (loop_sizes['K']%divval)==0:
                k_val=divval
                break
    return {
        "conv_2d": {
            "core_allocation": 1,
            "spatial_mapping": {'D1': ('OX', ox_val), 'D2': ('K', k_val)},
            "memory_operand_links": {'O': 'O', 'W': 'I2', 'I': 'I1'},
            "fixed_loops":['FX','FY','C'],
        }
    }
```

Figure 4.7: Object that includes for each layer type the associated optimal spatial mapping and other useful information like the mapping between the memory operands and the layer operands.

```
The temporal mapping is the following:
{'O': [[('C', 16), ('OX', 2)], [('OY', 2), ('OY', 2), ('OY', 2), ('OY', 2)],
[('OY', 2), ('K', 2), ('K', 2)]], 'W': [[], [('C', 16), ('OX', 2), ('OY', 2),
 ('OY', 2), ('OY', 2), ('OY', 2), ('OY', 2), ('K', 2), ('K', 2)], []], 'I': [
[], [('C', 16), ('OX', 2), ('OY', 2), ('OY', 2), ('OY', 2), ('OY', 2), ('OY',
 2), ('K', 2), ('K', 2)], []]}
```

Figure 4.8: Object that includes for each layer type the associated optimal spatial mapping and other useful information like the mapping between the memory operands and the layer operands.

"APIs", which initially were set as the x86 ones and then adapted to guarantee the execution on a real accelerator.

It is key to understand deeply how each layer is executed on DIANA. The L2 SRAM memory of DIANA can be written and read directly. HTVM stores there input and output and the layer receives their pointers. While it's up to the layer code to initialize the weight values on this memory.

The L1 memories instead are not accessible directly from the generated code, they in fact require to be managed through the DMA, Which can be contacted through the back-end functions. An important aspect to the DMA resides in the synchronization needed between each transfer operation.

Unfortunately through the current DIANA back-end the developer control ends up

```python
def zigzag(self,dory_tmap:bool=False,my_cost_model: bool=True, temporal_ordering: list=[]):
    # skip if not supported for testing purposes
    if len(self.workload.keys())==0:
        return
    needed_mem=(self.attrs['loop_sizes']['IX']*self.attrs['loop_sizes']['IY']*\
    self.attrs['loop_sizes']['C']*self.attrs['loop_sizes']['B']*(self.attrs[f'{self.type}_prec']['I']/8))\
    + (self.attrs['loop_sizes']['OX']*self.attrs['loop_sizes']['OY']*\
    self.attrs['loop_sizes']['K']*self.attrs['loop_sizes']['B']*(self.attrs[f'{self.type}_prec']['O_final']/8))\
     + (self.attrs['loop_sizes']['K']*self.attrs['loop_sizes']['C']*\
     self.attrs['loop_sizes']['FY']*self.attrs['loop_sizes']['FX']*(self.attrs[f'{self.type}_prec']['W']/8))
    if needed_mem>self.arch_def['limits']['max_memory']:
        raise Exception(f'This layer requires more memory than the one available, memory required {needed_mem} , memory available {diana_arch.max_memory}')
    print(f'Generating temporal mapping with following workload\n{self.workload}')
    self.mapping=self.arch_def['mapping'](self.attrs['loop_sizes'],self.type)
    if any([val[1]!=16 for key,val in self.mapping[self.type]['spatial_mapping'].items()]):
        # not supported yet feed to DORY
        self.workload=dict()
        return
    self.accelerator=self.arch_def['accelerator']()
    self.workload[1]['cost_model']=my_cost_model
    if not dory_tmap:
        energy, latency, cme = api.get_hardware_performance_zigzag(workload=self.workload,
                                                                   accelerator=self.accelerator,
                                                                   mapping=self.mapping,
                                                                   opt='latency',
                                                                   dump_filename_pattern=f"outputs/diana-{self.mod.attrs.global_symbol}-layer_?.json",
                                                                   pickle_filename=f"outputs/diana-{self.mod.attrs.global_symbol}-saved_list_of_cmes.pickle")
```

Figure 4.9: The figure pictures the process that is pursued to call the ZigZag API and transform its results.

at L1, while the lower level memory transfers are handled automatically. Because of this situation after each operand has transferred the needed data from the L2 SRAM memory up to the corresponding L1 memory it is necessary to execute the layer for all the data that is saved on L1 memory.

Code Generation for x86 hardware

To ensure the correctness of the code generation part and a correct build-up we decided to start with a template which that it possible to run a fused 2D convolutional layer.
The template needs to handle all the tasks needed to execute a layer, which includes:

- The **nested foor loops** which constitute the main computational structure of the layer.
- Setting of **auxiliary variables**.
- Updating correctly the **operand pointers**.
- **Management of memory transfers**, for contiguous and non contiguous data blocks.

So the first part of this work stood in the traduction of the temporal mapping to a set of correctly named, sized and ordered for loops. The ZigZag temporal mapping output diving more in depth is a python dictionary where the keys are the operands and the values are arrays of arrays, where each internal array represents the loops assigned to a certain memory level(Fig. 4.8).

```
[{'name': 'K', 'index': 2, 'fullname': 'K_2', 'size': 2, 'mem_O': 'dram', 'mem_W': 'weight_mem', 'mem_I': 'act_m
em'}, {'name': 'OX', 'index': 1, 'fullname': 'OX_1', 'size': 3, 'mem_O': 'dram', 'mem_W': 'weight_mem', 'mem_I':
 'act_mem'}, {'name': 'OY', 'index': 3, 'fullname': 'OY_3', 'size': 2, 'mem_O': 'act_mem', 'mem_W': 'weight_mem'
, 'mem_I': 'act_mem'}, {'name': 'OY', 'index': 2, 'fullname': 'OY_2', 'size': 2, 'mem_O': 'act_mem', 'mem_W': 'w
eight_mem', 'mem_I': 'act_mem'}, {'name': 'OY', 'index': 1, 'fullname': 'OY_1', 'size': 2, 'mem_O': 'act_mem', '
mem_W': 'weight_mem', 'mem_I': 'act_mem'}, {'name': 'OY', 'index': 0, 'fullname': 'OY', 'size': 2, 'mem_O': 'act
_mem', 'mem_W': 'weight_mem', 'mem_I': 'act_mem'}, {'name': 'K', 'index': 1, 'fullname': 'K_1', 'size': 2, 'mem_
O': 'act_mem', 'mem_W': 'weight_mem', 'mem_I': 'act_mem'}, {'name': 'C', 'index': 0, 'fullname': 'C', 'size': 16
, 'mem_O': 'act_mem', 'mem_W': 'weight_mem', 'mem_I': 'act_mem'}, {'name': 'OX', 'fullname': 'OX', 'size': 16, '
index': 0, 'mem_I': 'act_mem', 'mem_W': 'weight_mem', 'mem_O': 'act_mem'}, {'name': 'K', 'fullname': 'K', 'size'
: 16, 'index': 0, 'mem_I': 'act_mem', 'mem_W': 'weight_mem', 'mem_O': 'act_mem'}]
```


Figure 4.10: List containing structured for loops used by the template.

This representation is processed to rename the tiled loops and to store, for each loop, the memory instance name associated to each operand. This results in an array like the one in Fig. 4.10. Also as well with this internal representation for the loops a new description of the operands is also needed. For this object the relevancy of the loops for each operand is saved.
Given a stable and useful manner to represent a certain loop and along with it also a description of the used operands the next point is the definition of the initial shape and structure of the template.
Loop ordering is built on top of ZigZag output, by creating a set of nested for loops. Each loop has a unique name containing both the name of the dimension and the index relative to it is used.
Given the loops it is important to consider separetely the **spatial unrolled loops**, which are directly encapsulated by the back-end kernel that will map them onto the accelerator. The computation of which loops are spatially unrolled is handled outside of template. These loops are called kernel_loops, and are later on used by the template. This and other attributes, like the relevancy of each loop per each operator and others, are needed by the template to generate code and other structures.
Inside the template several **auxilary variables** are needed to hold useful information required at execution time that can't be precomputed easily statically. These variables range from structures of the operand, built specifically for that layer, to indexes useful for pointer computation. For example the structures internal to the template are computed by Mako with the code shown in Fig. 4.11, the resulting structure is not static and depends heavily on the layer, one possible result of this compilation for the output operand structure can end up be like the one figured in Fig. 4.12.
Defining instead a system to do **memory transfers** safely is more cumberstome since it requires calculating correctly both pointers associated with each memory tied to the transfer, for memory transfer intended for input operands this task requires only the calculation of the pointer on the higher end of the hierarchy, since the other pointer is obtained by the memory allocation process. The memory allocation process is defined through the response of the first free address of that

```
typedef struct dimension${layer}_${func_number}_t{
    % for l,m in dims[layer]['relevant_loops'].items():
    int dim_${m['mapping']}_size;
    int size_${m['mapping']}[4];
    % if 'partial_relevancy' in m:
    int size_${m['partial_relevancy']}[4];
    int addsize_${m['mapping']};
    % endif
    % if l!=m['mapping']:
    int pad_${m['mapping']}_x;
    int pad_${m['mapping']}_y;
    int max_${m['mapping']}_kernel_size;
    % endif
    % endfor
    % for tlname,tlsize in tiling_sizes.items():
    % if tlname.split('_')[0] in dims[layer]['relevant_loops']:
    int tile_${dims[layer]['relevant_loops'][tlname.split('_')[0]]['mapping']}${"" if len(tlname.split('_'))==1\
    else f'_{tlname.split("_")[1]}'}_size;
    % endif
    % endfor
}dimension${layer}_${func_number};
```

Figure 4.11: Code interpreted by Mako to generate a dimension structure.

```
typedef struct dimension0_18_t{
    int dim_K_size;
    int size_K[4];
    int dim_OY_size;
    int size_OY[4];
    int dim_OX_size;
    int size_OX[4];
    int tile_K_2_size;
    int tile_OY_1_size;
    int tile_K_1_size;
    int tile_OX_size;
}dimension0_18;
```

Figure 4.12: Output generated by Mako for a dimension structure, in this case the output one.

memory level, so it is just a contiguous allocator, that doesn't need any freeing. For the output operand instead each time there should be some data transfer between memory levels initially there is just a memory allocation and after the whole calculations there is always the memory transfer.

Memory transfers on the x86 code generation version are quite simple and are defined just through a simple memory copy done in a nested for-loops fashion, the most complicated aspect on this point is about the correct pointer calculations, since

it needs to take in consideration the schema of the operand, so where dimension stand on the plane of the operand. This calculation is carried out by exploiting the tile indexes of each dimension.

Then after the memory transfers the core calculations are executed, and they are encapsulated inside the kernel function, that carries out the specific function for the layer. In the case currently supported it is the convolution, and it's done between the data in the lower level memory is done by looping through all the kernel loops, that include the loops that are spatially mapped and the ones that are mapped at the lower level memories.

Finally to be fully functional between the loops it is important to **update correctly the operand pointers**. This is done exclusively in case of a memory transfer or right before the kernel function. The update is managed by analyzing over the past loops the ones relevant for the operand. The past loops are the ones from the last memory transfer until that moment. Additionally to this calculation at the end of each for loop all the information that is used to calculate the pointers is cleared. These are the tile indexes for each dimension that was calculated previously. The pointer updating done before the kernel function is crucial since ZigZag supports also uneven mapping.

Code Generation for DIANA

Proceeding on the development of the template to make code generation for the DIANA SoC accessible some modifications with respect to the x86 version are made. These changes are mainly about the development of new APIs specific for the PULP platform, and other few ones that target some small data layout differences.

During this integration experiment the APIs target the digital accelerator which is more flexible and precise on average with respect to the analog accelerator, but during this process is quite evident the fact that integrating different accelerators and more in general different platforms as well is not too complicated and therefore the thesis can be extended to support a vast array of heterogeneous edge devices.

The main APIs for the code generation on DIANA that needed a change feature the memory transfer ones and the kernel function one, in this experiment the used back-end for DIANA is the Dory-Hal one, which has a complete implementation of all theses aspects and . Other than this Dory-Hal dependency there is also the dependency on a specific function of the Dory library for the memory copy from a level to another one.

4.3 Testing set-up

The performance of the system has been evaluated in 2 different manners, through RTL simulations and the physical DIANA SoC. For the RTL simulations the traces have been used to collect performance values. Instead, while using directly the DIANA SoC an inner performance counter is used.

4.3.1 RTL simulations

RTL simulations [27] can be very useful to analyze more in depth each signal. However, they are way slower than real hardware. They generate traces that give access to waves for several component. Out of all the signals DIANA-RTL gives access to, the main ones that have been analyzed regard the memory lanes and the digital accelerator state.
To collect performance data for every type of contribution two cursors are used. These cursors, put at the start of the measured signal and at the end of it, display a time in ps. The times of the cursors are then substracted and the value is divided by 200000. This value is used because for the accelerator is set to have a clock of 500Mhz.
The RTL simulations have been used mainly to analyze in a more complete manner how DIANA execute internally the Dory-hal back-end kernels and memory transfers.

4.3.2 DIANA setup

DIANA can be connected in two different ways, through a USB-C to USB-B cable or through the internet if it connected with an ethernet cable to the same network, the assessed experiments were all carried out in the first way. Having successfully generated some code for DIANA thanks to TVM+ZigZag the next step is to execute it directly on hardware, to do so the current approach requires a few steps, first of all after having phisically connected the board to the computer it is necessary to connect to it through a SSH channel and power it up, to do so it is advised to follow this pattern:

- On a brand new terminal run "ssh xilinx@192.168.1.1"
- After the ssh connection is created succesfully it is necessary to run some code that requires a superuser, so to simplify the following steps run "sudo su" on the same terminal
- This terminal is used to power on the board, currently to do it is required to first run "export XILINXUSER", then "cd faim-sdk" and finally "python3 00test.py", the last command should power on correctly the board and display

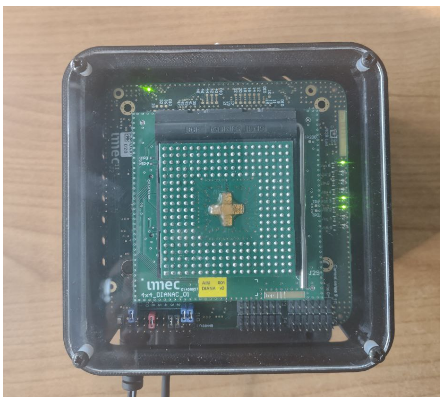

Figure 4.13: DIANA board connected properly with an USB-C cable

the current voltages and other hardware characteristics, it can be used to assess the fact that the board is not faulting.

After having powered up correctly the board like in Fig. 4.13 it is necessary to start on DIANA openocd as well, this tool is used to start up a remote GDB server, that will be exploited to run the code later on. To correctly execute this procedure these steps are advised:

- Since the code generation setup is encapsulated inside a docker container it is necessary to connect to the board and exploit ssh tunneling, in this way the board servers can be contacted from the container as well, to do so on a brand new terminal run "ssh -L 3333:localhost:3333 xilinx@192.168.3.1"

- Similarly to before also here some actions require superuser permissions, so on the same terminal run "sudo su" and right after "export XILINX_XRT=/usr"

- finally to start up openocd run "sudo /home/xilinx/openocd/openocd -f /home/xilinx/openocd/openocd-xlnx-pcie-xvc.cfg"

The last two steps are required to setup correctly the board and making it possible to connect to a GDB server used to execute the necessary code. On a terminal where the active directory is placed on the folder of the compiled code now it is possible to run "/pulp/bin/gdb -x gdb demo.sh pulpissimo/demo/demo" that will execute the gdb script on the demo executable.
The current setup of DIANA will be in future improved through a Python script which will automate the process.

4.4 ZigZag modification

After having built a working solution for the 2D convolution, the goal was to maximize the performance of the tool. For this purpose an accurate cost model is needed. However, following sessions of testing, performed as described in the previous section, we noticed that ZigZag cost model had big inaccuracies w.r.t. to real hardware performance. Overall, the cost model evaluation is off by up to 73 times in some of the tested cases.
Some differences are expected since the programming overhead is not taken into account. However, the projected difference between the cost model evaluation of ZigZag and the actual result should always be around 5% or less. This should be even more true on some of the analyzed cases, that do not require any tiling. Meaning that they can be pretty much approximated as just a kernel function call, thus reducing to a net minumum all the programming overhead.
Finally, it is clear that the inaccuracies of ZigZag's cost model were not only due to general programming overhead. Therefore further research is needed to understand the reasons causing the differences. To carry out the research several trace of DIANA-RTL simulations have been analyzed.

4.4.1 New Computational Model Engine

Checking out the simulation traces we notice various factors that are not taken in consideration inside ZigZag. These factors range from ones regarding the computation aspect to the memory transfer ones.
Concerning the computation the cost model doesn't consider these factors:

- Pipeline used for each **partial computation**.
- **Overhead** needed to manage MAC units.

While the memory evaluation carried out by ZigZag's cost model doesn't take in consideration these parts:

- **Input and weights** transfers unified.
- **Multiplicities** of transfers.
- **Programming overhead** for each transfer.

4.4.2 Computational aspect

Regarding the computation process the main difference stands in how ZigZag expects the MAC units to work. In fact the computation latency on ZigZag is evaluated taking in consideration only the ideal number of cycles, with the given MAC array,

and a negligible overhead brought up by bandwidth bottlenecks. However, this last factor is trivial and accounts for only 1% of the whole computation latency on average.
The first contribution to this overall computation latency can be easily calculated by the following formula:

$$idealtemporalcycles = \prod_{i=1}^{lp_n} lpsize_i / (\prod_{i=1}^{spatlp_n} spatlpsize_i)$$

Where lpsize groups the sizes of each for loop and spatlpsize groups the sizes of the for loops that are part of the spatial mapping.
Instead, while checking the simulation trace it can be noted that the computation process of the back-end kernel is very different from the ideal one that ZigZag expects. In fact, instead of a continuous computation, DIANA, through Dory-hal back-end kernel, employs a 3 staged pipeline for the 2D convolution. This pipeline is executed to compute each partial output. The stages are the following ones:

- A cycle is used to set the address of the first input used and therefore load the input data on the following cycle.
- Another cycle is used to set the weights address that will be subsequently used to load the weights data. But this cycle is mostly used for the general preparation of the MAC units, thus making them available for the following cycles.
- Finally there will be one or more cycles where the outputs are generated by the MAC units, and in the meantime other inputs or weights are loaded as well.

These stages are coded by the digital accelerator state unit in the values showed in Fig. 4.14. Where the first stage is coded with 1, the second with 3 and finally the computing one with 4.
The computing stage for the convolution is processed over a number of cycles that is equivalent to the width of the weights, FX. In fact if we analyze the simulations of layers with different values of FX the behaviour changes. In Fig. 4.15 the state of the digital accelerator is shown for a layer where FX has size equal to 1 and as expected the stage marked with 4 takes only 1 cycle every time. While in Fig. 4.16 a layer where the weights have a width of 7 is displayed and the last stage is always processed over 7 cycles.
This pipeline is executed to carry out the computation of just a single partial result. So, it must be repeated for every dimension that is not unrolled by the MAC units. The number of times the pipeline is executed depends on 2 values: the sizes of the dimensions, and also the value of the padding on the height. In fact, with the

```
reg [LL_FSM_bits-1:0] state;
reg [LL_FSM_bits-1:0] next_state;

reg layer_finished;
reg next_layer_finished;

localparam              INITIAL = 0,
                        CONV_FILLING_INPUT_FIFO = 1,
                        CONV_MAC_SHUFFLING = 2,
                        CONV_PRE_MAC = 3,
                        CONV_MAC = 4,
                        CONV_FILLING_SHUFFLING_BUFFER = 5,
                        CONV_PRE_PASSING_OUTPUTS_VERTICAL = 6,
                        CONV_PASSING_OUTPUTS_VERTICAL = 7,
                        CONV_CLEAR_MAC = 8,
```

Figure 4.14: Accelerator state values, focusing on the convolutional ones.

Figure 4.15: Simulation trace focusing over the state of the accelerator while computing a layer where the weights have a width of 1.

presence of padding many partial calculations can be skipped because they would all be resulting in just a bunch of zeros.
Finally there is another overhead that is happening on the computation process. In fact, after every set of results it is necessary to: pursue the activate function, if required, then clearing up the MAC units, and finally moving all the results generated onto L1 memory. All of this overhead accounts for 23 cycles.
Therefore, the number of cycles needed for computation by DIANA can be known apriori. This value can be calculated through the following formula:

$$CompCycles = \sum_{pad=0}^{pad_Y} ceil(K * (pad == 0?OY - 2 * pad_Y : 2) * OX/SpatSize) * \\ (C * (FY - pad) * FX + 2 * C * (FY - pad) + 23) \quad (4.1)$$

Finally, to improve the accuracy of ZigZag's cost model for the computational part, the formula 4.1 must be implemented instead of the ideal calculation. This is done through the code shown in Fig. 4.17.

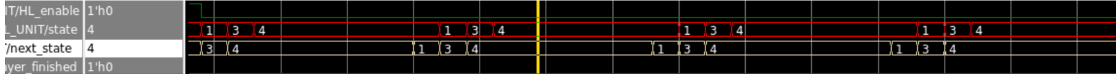

Figure 4.16: Simulation trace focusing over the state of the accelerator while computing a layer where the weights have a width of 7.

```
spatial_mapping_sizes=prod([dim[1] for (key,dim) in self.layer_node.user_spatial_mapping.items()])
no_pad_size=self.layer_node.loop_dim_size['K']*(self.layer_node.loop_dim_size['OY']-2\
            *self.layer_node.padding['IY'][0])*self.layer_node.loop_dim_size['OX']
pad_size=self.layer_node.loop_dim_size['K']*2*self.layer_node.loop_dim_size['OX']
contrib=[((self.layer_node.loop_dim_size['C']*(self.layer_node.loop_dim_size['FY']-pad)*\
        self.layer_node.loop_dim_size['FX'])+(self.layer_node.loop_dim_size['C']*\
        (self.layer_node.loop_dim_size['FY']-pad)*2)+23) for pad in range(self.layer_node.padding['IY'][0]+1)]
self.contrib=contrib
self.pad_size=pad_size
self.no_pad_size=no_pad_size
self.spatial_mapping_sizes=spatial_mapping_sizes
self.total_cycle = sum([ceil((no_pad_size if pad==0 else pad_size)/spatial_mapping_sizes)*\
                        contrib[pad] for pad in range(self.layer_node.padding['IY'][0]+1)])
```

Figure 4.17: Code used to calculate properly the number of cycles needed for the computation of a 2D convolutional layer over DIANA.

4.4.3 Memory transfer aspect

While ZigZag evaluates accurately the sheer number of cycles for the memory transfers between the L2 SRAM memory and the L1 memories, the cost model is still inaccurate. The memory transfer differences are due to 3 main factors: the significant programming overhead, how ZigZag mixes up weights and inputs, and lastly the fact that ZigZag accounts only for the first transfer of data without considering the correct multiplicities.
The first problem is the fact that ZigZag hides the cost related to one of the inputs, by accounting only for the maximum value between the two costs. Since weights transfers and activation ones are not done in parallel it is important to untie these inputs values and sum them up instead of picking just the maximum.
Instead, regarding the programming overhead of each transfer, this factor is quite trivial over big transfers. However, many transfers cannot be performed as a single entity. This can happen due to tiling, that forces memory transfers over non contiguous areas. Therefore, a single 3D memory transfer can be broken down over multiple 2D or 1D ones, which will be smaller. For each one of these smaller transfers we have to account for the programming overhead, that can become huge at the end.
Finally the last difference stands in the fact that ZigZag accounts only for the first data transfer of an operand. So, in cases requiring tiling ZigZig take in consideration only the transfer of the first tile.
To fix these last two issues we compute, given a certain temporal mapping the

```
def calc_multiplicity_l2_and_transfer_overheads(self):
    multiplicity_l2={key:prod([v[1] for v in val[len(val)-1]]) for (key,val) in self.temporal_mapping.mapping_dic_stationary.items()}
    # diana contrib
    multiplicity_l2['W']=max([multiplicity_l2['O'],multiplicity_l2['I']])
    multiplicity_l2['O']=multiplicity_l2['W']
    tmap=self.temporal_mapping.mapping_dic_stationary
    lsize=self.temporal_mapping.layer_node.loop_dim_size
    relmap={key:{'r':val['r']+sorted([v_[0] for k_,v_ in val['pr'].items()])[::-1],'ir':val['ir']}\
            for (key,val) in self.temporal_mapping.layer_node.operand_loop_dim.items()}
    multiplicity_rel_L2={operand:{reldim:prod([val[1] for val in tmap[operand][len(tmap[operand])-1] if val[0]==reldim])\
            for reldim in relmap[operand]['r']} for operand in self.temporal_mapping.operand_list}
    for comm in set(relmap['O']['r']).intersection(set(relmap['I']['r'])):
        multiplicity_rel_L2['O'][comm]=max([multiplicity_rel_L2['O'][comm],multiplicity_rel_L2['I'][comm]])
    def get_transfer_calls_per_time_from_to_l2(operand):
        if operand not in ['O','I']:
            return 1
        len_rel_map_operand=len(relmap[operand]['r'])
        for ind in range(len_rel_map_operand)[::-1]:
            if ind==0:
                return 1
            if multiplicity_rel_L2[operand][relmap[operand]['r'][ind]]!=1:
                return prod([lsize[relmap[operand]['r'][prod_lp]]/multiplicity_rel_L2[operand][relmap[operand]['r'][prod_lp]] for prod_lp in range(ind)])
    self.transfer_calls_per_time_from_to_l2={operand:get_transfer_calls_per_time_from_to_l2(operand) for operand in self.temporal_mapping.operand_list}
    self.relmap=relmap
    self.multiplicity_l2=multiplicity_l2
    self.multiplicity_rel_L2=multiplicity_rel_L2
```

Figure 4.18: Code used to calculate properly the number of transfers that are actually executed to move a block, and the overhead brought by them.

correct multiplicites. These multiplicities take in consideration the actual type of transfer exectued, if 3D, 2D or 1D, and finally also the overall number of transfers needed for an operand. The code represented in Fig. 4.18 is embedded onto ZigZag's cost model.
Finally, thanks to the inserted hardware-aware informations, the quality of the generated temporal schedule improves drastically. ZigZag can assess the schedules in a way that resembles more the reality of the hardware, and can finally output better schedules.

Chapter 5

Experimental Results

This chapter is dedicated to the visualization and analysis of the most interesting results obtained during this project.
Each of the results has been gathered on the DIANA board, exploiting the RISC-V core and the digital accelerator one, thus leaving for a future extension the analog accelerator core.
The baseline is defined as the inference of the network, compiled through TVM, pursued by the RISC-V core: these results are really useful to understand the huge need for a dedicated hardware module.

5.1 Single Fused Convolutional Layer

The models analyzed for the majority of this thesis consist of a set of benchmarks that comprise a single fused layer. This layer includes a 2D convolution layer, a bias layer, a right shift layer, and finally, the activation function.
The results are obtained over a set of single-layer networks with varying hyperparameters. The selected hyperparameters target several case studies that require different tiling constraints. Additionally, these hyperparameters set the dimensions of the layer and act as a virtual limit for the DIANA's memory sizes. Setting smaller memories for the accelerator will require ZigZag and DORY to allocate some loops in higher levels of the memory hierarchy. This is necessary to ensure that many use cases are tested and to understand the importance of temporal mapping and tiling.
The group of layers used is divided into four sets, each representing a fixed memory configuration for the accelerator. For the first set (Set A), the activation memory is fixed at 24KB, while the weight memory and the L2 SRAM remain unchanged. The following table lists the layer configurations for this set:
For the second set (Set B), the layers can use up to 64KB of activation memory

Name	K	C	FY	FX	OY	OX
L1	64	16	1	1	16	48
L2	64	16	1	1	48	16
L3	64	16	1	1	16	64
L4	64	16	1	1	48	32
L5	64	16	1	1	32	32
L6	64	16	1	1	16	16
L7	64	16	1	1	32	48
L8	64	16	1	1	48	48
L9	32	16	1	1	16	64
L10	32	16	1	1	32	64
L11	16	32	1	1	12	64
L12	16	32	1	1	8	80
L13	16	80	1	1	6	80

Table 5.1: First Set of Layers (24KB of activation memory and 64KB of weight memory)

and 64KB of weight memory. The following table lists the layer configurations for this set:

Name	K	C	FY	FX	OY	OX
L1	32	32	1	1	50	32
L2	32	32	1	1	64	32
L3	16	10	1	1	50	64

Table 5.2: Second Set of Layers (64KB of activation memory and 64KB of weight memory)

For the third and last set (Set C), the layers can use up to 128KB of activation memory. The following table lists the layer configurations for this set:

5.1.1 Comparison of TVM+ZigZag with the Baseline

The first results we will analyze feature those obtained by exploiting the baseline configuration and also those gathered by TVM+ZigZag.
The results (Fig. 5.1) for the first set (Tab. 5.1) show massive improvements. In fact, with this configuration, TVM+ZigZag outperforms TVM by an average factor of 21, reaching a speedup that peaks at around 100 times.
Over the second set (Tab. 5.2), the improvements (Fig. 5.2) are even more significant, thanks to the larger activation memory that TVM+ZigZag can utilize.

Name	K	C	FY	FX	OY	OX
L1	64	70	1	1	32	32
L2	32	70	1	1	32	48
L3	32	70	1	1	48	32
L4	32	70	1	1	25	64

Table 5.3: Third Set of Layers (128KB of activation memory and 64KB of weight memory)

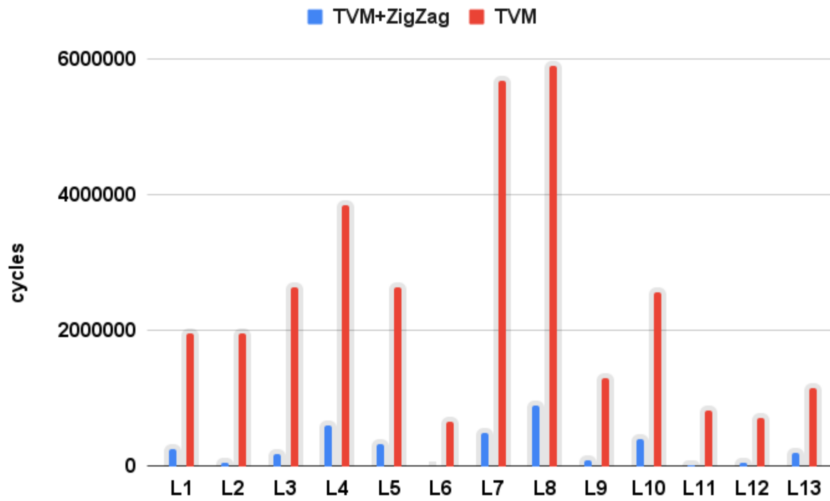

Figure 5.1: Results of the inference for the layers of the first set (Tab. 5.1), comparing the baseline configuration with TVM+ZigZag.

On average, TVM+ZigZag provides a speed-up of 28 times over the baseline. However, the peak performance is smaller due to the smaller size of the set, with peaks of 64 times.

Finally, the results (Fig. 5.3) for the last set (Tab. 5.3) display even more substantial improvements, both in average and peak results. TVM+ZigZag reduces the cycles needed for the inference of a single layer by an average of 55 times, with peaks of 124 times. The improvements from set A to set B, and from the latter to set C, confirm that with a larger memory array size, ZigZag can significantly enhance the results.

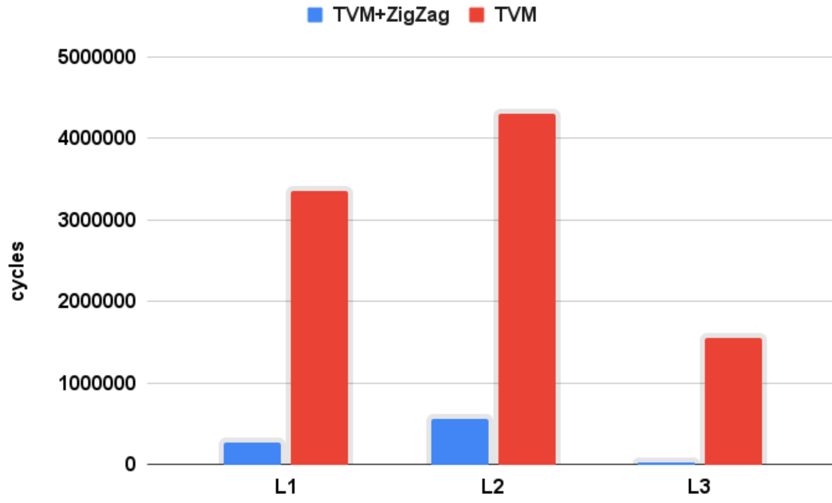

Figure 5.2: Results of the inference for the layers of the second set (Tab. 5.2), comparing the baseline configuration with TVM+ZigZag.

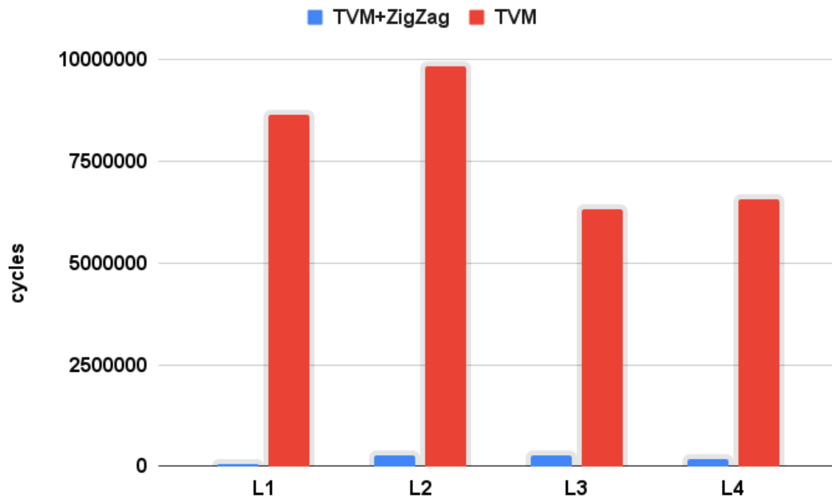

Figure 5.3: Results of the inference for the layers of the third set (Tab. 5.3), comparing the baseline configuration with TVM+ZigZag.

5.1.2 Comparison of TVM+ZigZag with HTVM

After this initial round of results, the next step involves an analysis with the current state-of-the-art, HTVM.
The results (Fig. 5.4) for the first set (Tab. 5.1) demonstrate that HTVM

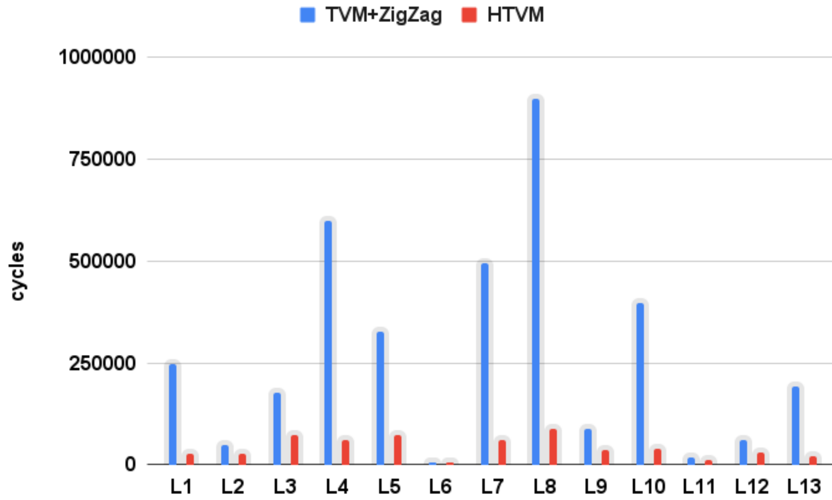

Figure 5.4: Results of the inference of the layers of the first set (Tab. 5.1), comparing the HTVM configuration with TVM+ZigZag.

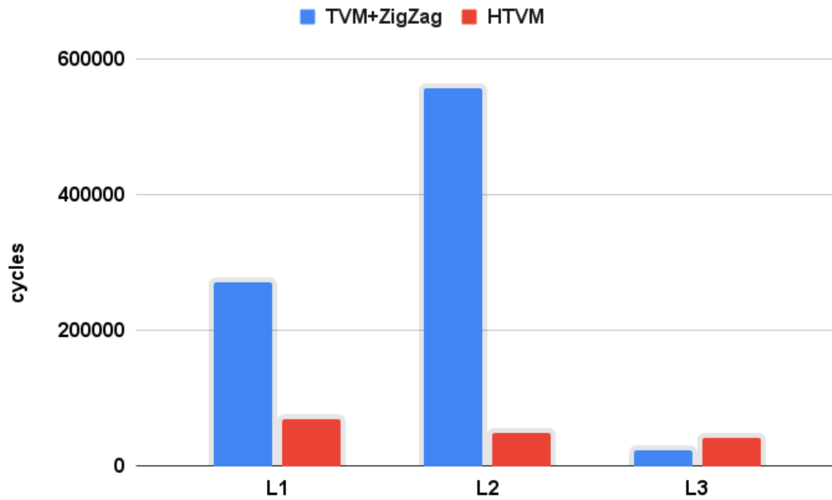

Figure 5.5: Results of the inference of the layers of the second set (Tab. 5.2), comparing the HTVM configuration with TVM+ZigZag.

outperforms TVM+ZigZag significantly. On average, HTVM achieves results that are 5 times better, while the best TVM+ZigZag result on the same layer achieves the same latency as HTVM.
Analyzing the results (Fig. 5.5) obtained over the second set (Tab. 5.2), the

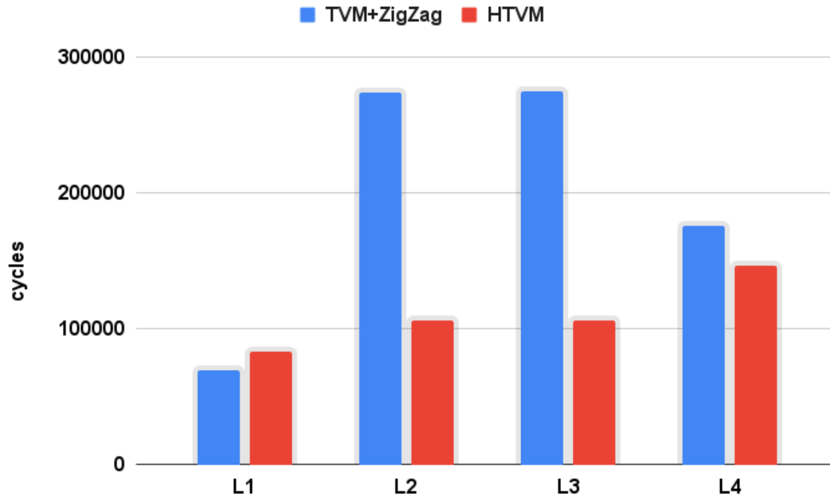

Figure 5.6: Results of the inference of the layers of the third set (Tab. 5.3), comparing the HTVM configuration with TVM+ZigZag.

trend remains similar on average, while the best-case scenario of this set shows an improvement of over 1.5 times compared to HTVM.
Finally, checking the results for the final set (Fig. 5.6) (Tab. 5.3), it is evident how the memory size significantly impacts the quality of ZigZag's scheduling. These results are a substantial improvement over the previous ones, with this thesis's performance being just 1.8 times worse on average, and in the best cases, TVM+ZigZag brings improvements of about 1.2 times over HTVM.
Summing up all these results, TVM+ZigZag has, on average, a latency that is 4 times worse than HTVM. However, in the best cases, it shows improvements over HTVM by about 1.8 times.

5.1.3 Problems of TVM+ZigZag

To comprehend why HTVM outperforms TVM+ZigZag on average, a comprehensive breakdown of the performance data has been conducted. As expected, the primary difference lies in the tiling and scheduling generated by DORY and ZigZag. To gain insights into ZigZag's scheduling choices, several simulations of DIANA executing TVM+ZigZag, achieved through DIANA-RTL, were performed. Simulations were chosen as they provide latency results for specific parts of the layer. The simulation data was then compared to the corresponding data obtained from the ZigZag Cost Evaluation Model.
The considered data can be categorized into two parts. The first one pertains to

the computational aspect of the layer, specifically the kernel function. The second one instead relates to the costs associated with memory transfers.
To analyze the generated data, two distinct groups of layers were used. The first group encompasses cases that do not require tiling, consisting of layers small enough to fit within a 256KB activation memory, and can be executed directly by the kernel function. An important hyperparameter in this group was the size of padding over OY and OX. Here are the layers belonging to this group:

Name	K	C	FY	FX	OY	OX	pad OY	pad OX
L1	64	64	3	3	32	32	1	1
L2	64	64	1	1	32	32	0	0
L3	64	64	3	1	32	32	1	0
L4	64	64	1	3	32	32	0	1
L5	64	16	7	7	48	48	3	3

Table 5.4: First Group of Layers (256KB of activation memory and 64KB of weight memory) used for analyzing and comparing simulation data with ZigZag's expected results

The second group, on the other hand, consists of layers with a smaller activation memory of 24KB and a requirement for tiling. Specifically, these layers tile one of the dimensions by a factor of 2, and the execution calls the kernel function twice.

Name	K	C	FY	FX	OY	OX	pad OY	pad OX
L1	32	80	1	1	16	16	0	0
L2	32	64	1	1	32	16	0	0

Table 5.5: Second Group of Layers (24KB of activation memory and 64KB of weight memory) used for analyzing and comparing simulation data with ZigZag's expected results

Computational Cost Analysis

The computational cost is expressed as the number of cycles required to execute the layer without considering the overhead brought by memory movements. The data gathered (Fig. 5.7) over the first group (Tab. 5.4) reveals a significant difference between the ZigZag evaluation and the actual cost. In fact, the ZigZag evaluation deviates by around 40% on average, and in the best cases, it deviates by 20%. The best cases for the ZigZag Cost Model Evaluation in this group are those where the layer has more padding over the width (OX). Out of the 5 layers used in this group, in the 2 cases with no padding on the width, ZigZag's evaluation deviates

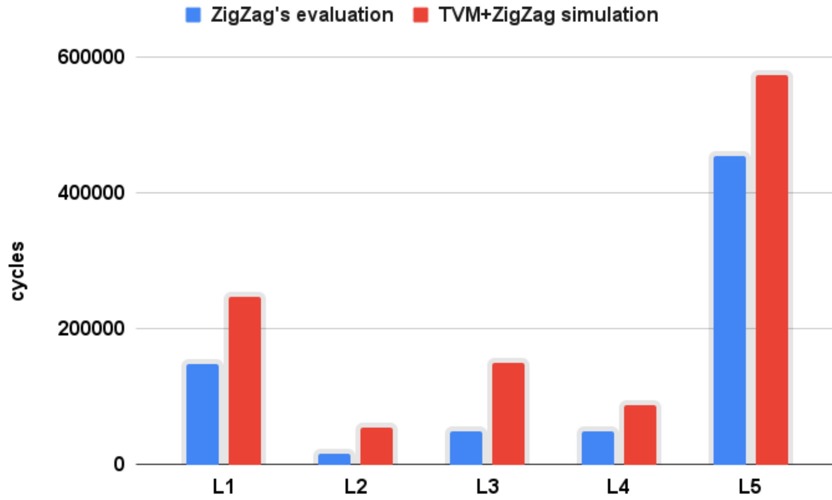

Figure 5.7: Results of the computational cost for the inference of the layers in the first group (Tab. 5.4) compared with ZigZag's evaluation.

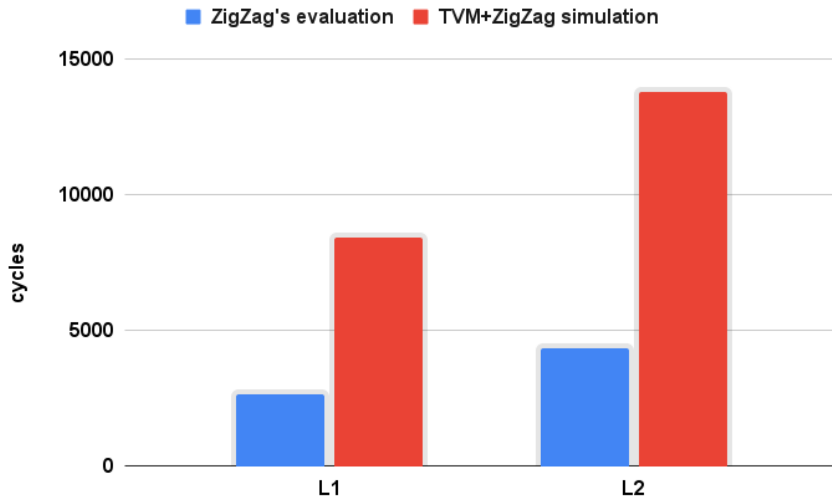

Figure 5.8: Results of the computational cost for the inference of the layers in the second group (Tab. 5.5) compared with ZigZag's evaluation.

by 67%. This discrepancy is due to how the kernel is executed, as the convolution kernel's performance is highly dependent on the width padding.
Checking out the second group(Tab. 5.5) instead the disproportion(Fig. 5.8) over the two configurations is quite big. Both of these layers don't include any padding,

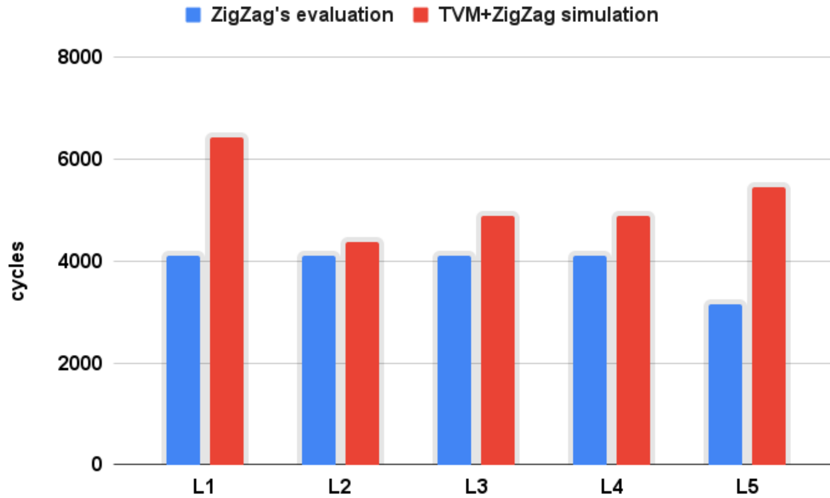

Figure 5.9: Comparison between the evaluation of the memory onloading cost fullfilled by ZigZag cost model and the actual latency that it takes for the layers that are part of the first group(Tab. 5.4).

and as analyzed previously this is key for the performance results over DIANA. ZigZag ends up underestimating the cycles required by 68%, which is the same value received by the unpadded layers of the first group(Tab. 5.4).

Memory transfers cost analysis

The memory transfers results are broken down as ZigZag does, so by dividing them into 2 categories, onloading and offloading. The first category includes the transfers of weights and inputs and the second one only the output transfers.
Regarding the onloading transfers the results gathered(Fig. 5.9) over the first group(Tab. 5.4) display a slight difference between the 2 configurations, ZigZag's data ends up being 1.3 times smaller on average than the actual value. This difference is due to the fact that ZigZag picks as the overall onloading cost the maximum value between the weight transfers and the input ones. In fact, it is possible to dive deeper into the evaluations of ZigZag and break down the data into weight transfers and input ones. Following this procedure we can see on the chart of the weight transfers(Fig. 5.10) that ZigZag evaluates correctly this cost, the same happens for the input values(Fig. 5.11).
Checking out the second group(Tab. 5.5) results(Fig. 5.12) the difference between the evaluation and the simulation varies a lot from layer to layer, the first layer doesn't make it necessary to move the input multiple times but just once, while the second one require multiple transfers for the input. This multiplicity is not

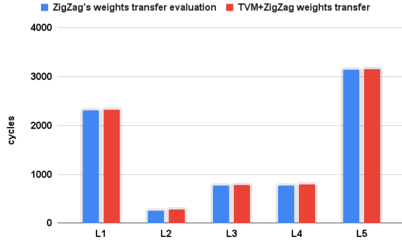

Figure 5.10: Comparison between the evaluation of the weight transfer cost fullfilled by ZigZag cost model and the actual latency that it takes for the layers that are part of the first group(Tab. 5.4).

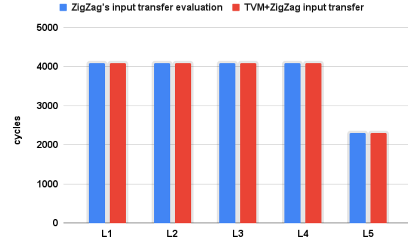

Figure 5.11: Comparison between the evaluation of the inpu transfer cost fullfilled by ZigZag cost model and the actual latency that it takes for the layers that are part of the first group(Tab. 5.4).

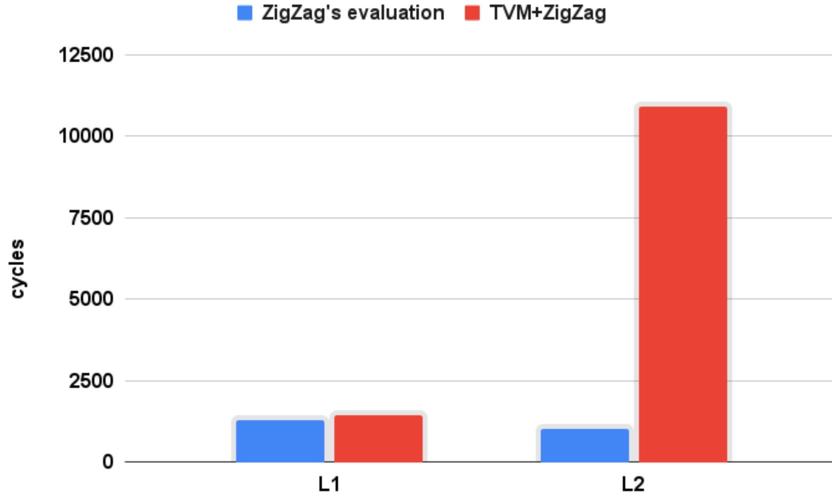

Figure 5.12: Comparison between the evaluation of the memory onloading cost fullfilled by ZigZag cost model and the actual latency that it takes for the layers that are part of the second group(Tab. 5.5).

considered by ZigZag cost model, and only the first transfer is accounted. Another parameter that is not accounted by ZigZag is the overhead necessary for a transfer, which depends on the type of transfer.
Analyzing more in depth the second layer results for only the input transfers we can see how each part contributes for the complete results. In Fig. 5.13 we can see that if we consider only the memory transfer cost ZigZag evaluation is quite accurate if we normalize the value with the correct multiplicity. The big difference

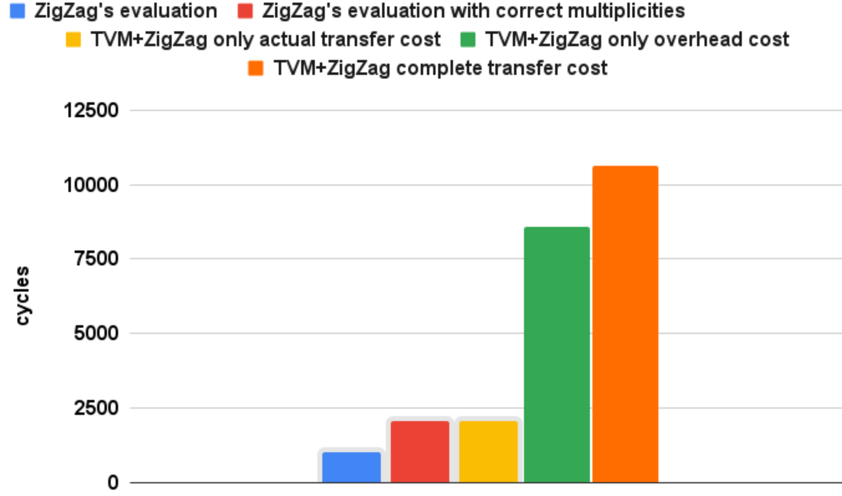

Figure 5.13: Results of the input transfer for L2 of the second group(Tab. 5.5) browken down over actual transfer and overhead.

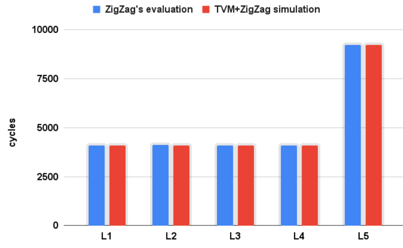

Figure 5.14: Comparison between the memory offloading cost evaluation by ZigZag's cost model and the actual latency for the layers in the first group (Tab. 5.4).

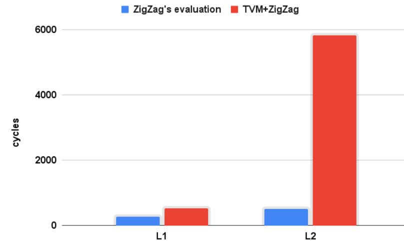

Figure 5.15: Comparison between the memory offloading cost evaluation by ZigZag's cost model and the actual latency for the layers in the second group (Tab. 5.5).

in this case is due to the programming overhead required to handle this 3D transfer, which is broken down over 64 smaller 1D transfers. For each one of these smaller transfers we need to account for the programming overhead. Overall if we consider the complete tranfer cost, the overhead cost is 80% of the whole transfer. This part is not considered by ZigZag and is the most important one.
For the offloading transfers, the situation is similar, and the evaluations are very accurate (Fig. 5.14) for the first group (Tab. 5.4), while for the second group (Tab. 5.5), the multiple output transfers and their associated overhead are affecting ZigZag's cost model (Fig. 5.15).

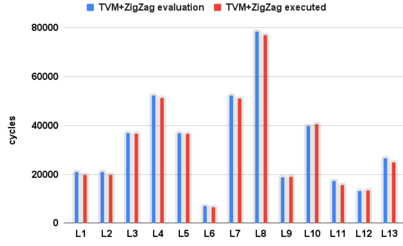

Figure 5.16: Comparison between the evaluation fulfilled by ZigZag's updated cost model and the actual results for set A (Tab. 5.1).

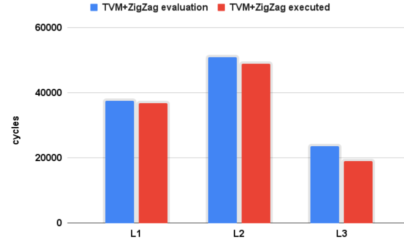

Figure 5.17: Comparison between the evaluation fulfilled by ZigZag's updated cost model and the actual results for set B (Tab. 5.2).

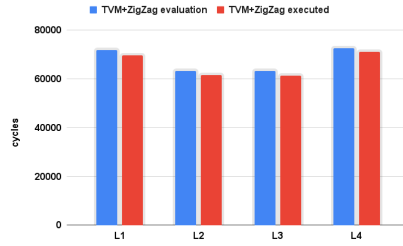

Figure 5.18: Comparison between the evaluation fulfilled by ZigZag's updated cost model and the actual results for set C (Tab. 5.3).

Accuracy Improvements in the Cost Model Evaluation

With a new cost model that accounts for the previously omitted programming overheads, multiplicity of transfers, and correct computational cost calculation, ZigZag's evaluation becomes highly accurate.

For the first set (Tab. 5.1), the quality of the evaluation (Fig. 5.16) is notably high. On average, ZigZag overestimates the actual latency by just 3.2%, and in the best case, L5, the estimation deviates by only 0.2%.

The situation is somewhat different for the second set (Tab. 5.2), where, on average, ZigZag's estimation deviates by 10.31% (Fig. 5.17). This is partly due to the smaller set and a specific case, L3, which is evaluated incorrectly by 24%. This specific case is more sensitive to overestimations of the overhead cost due to its small size. However, the remaining two layers are evaluated more accurately, with evaluation accuracy rates of 98% and 96%, respectively.

Finally checking the evaluation accuracies for the third and final set(Tab. 5.3) the results(Fig. 5.18) are even better than both previous sets. In fact in these cases

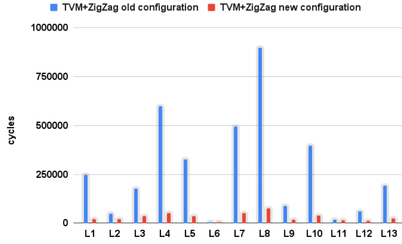

Figure 5.19: Results and comparison of the inference of layers in set A (Tab. 5.1).

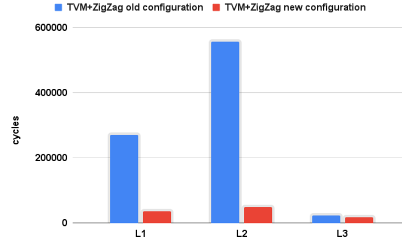

Figure 5.20: Results and comparison of the inference of layers in set B (Tab. 5.2).

ZigZag estimation is overestimating the number of cycles required to infer a layer by just 2,78% on average.
If we consider all of the sets, the new cost model proves to be significantly more accurate and successfully achieves the goal of staying within a 5% error range. Across these layers, the average error is only 4.18%.

5.1.4 TVM+ZigZag New vs Old Cost Models vs HTVM

With a more precise cost model, the quality of the schedules generated by ZigZag improves dramatically. Comparing this new configuration with the initial TVM+ZigZag one shows significant improvements.
If we consider only the first set (Tab. 5.1), this new configuration outperforms the off-the-shelf ZigZag by 72.37% on average (Fig. 5.19), with a peak of 92%.
The results (Fig. 5.20) are almost identical for the second set (Tab. 5.2). On average, this new configuration is better by 66.23%. Also, for the third and final set (Tab. 5.3), the improvements are quite significant, averaging 53.67% (Fig. **??**).
These last two sets have smaller gains compared to the first one, mainly due to the quality of the results from the starting configuration, which was already competitive with HTVM. With the higher memory capacity, ZigZag was already performing quite well.
As expected, the quality of the schedules is much better across all sets. To illustrate the impact, we can analyze one layer as an example.

TVM+ZigZag Old Configuration vs New One - Single Layer Analysis

The selected layer for this analysis is L1 of set A (Tab. 5.1). The old configuration's scheduling (Fig. 5.21) tends to prefer tiling the OX and K dimensions, which impacts the optimality of memory transfers. The new configuration (Fig. 5.22), being aware of the overhead caused by suboptimal data movement, chooses to

Temporal Loops	O	W	I
for K in [0:2)	dram	weight_mem	act_mem
for OX in [0:3)	dram	weight_mem	act_mem
for OY in [0:2)	act_mem	weight_mem	act_mem
for OY in [0:2)	act_mem	weight_mem	act_mem
for OY in [0:2)	act_mem	weight_mem	act_mem
for OY in [0:2)	act_mem	weight_mem	act_mem
for K in [0:2)	rf_4B	weight_mem	act_mem
for C in [0:16)	rf_4B	weight_mem	act_mem

Figure 5.21: Scheduling produced by the off-the-shelf version of ZigZag for L1 of set A (Tab. 5.1).

Temporal Loops	O	W	I
for K in [0:2)	dram	weight_mem	act_mem
for K in [0:2)	dram	weight_mem	act_mem
for OX in [0:3)	act_mem	weight_mem	act_mem
for OY in [0:4)	act_mem	weight_mem	act_mem
for OY in [0:4)	act_mem	weight_mem	act_mem
for C in [0:16)	rf_4B	weight_mem	act_mem

Figure 5.22: Scheduling produced by the new and upgraded version of ZigZag for L1 of set A (Tab. 5.1).

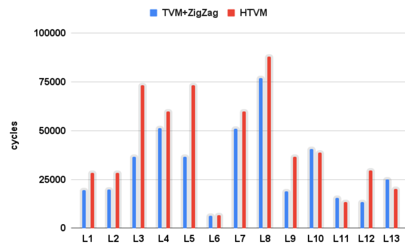

Figure 5.23: Results and comparison of the inference of layers of the first set (Tab. 5.1) compiled with TVM+ZigZag and HTVM.

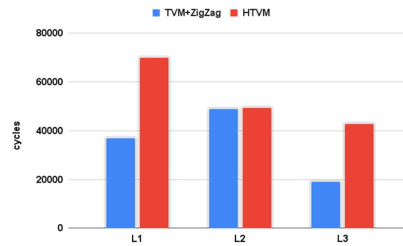

Figure 5.24: Results and comparison of the inference of layers of the second set (Tab. 5.2) compiled with TVM+ZigZag and HTVM.

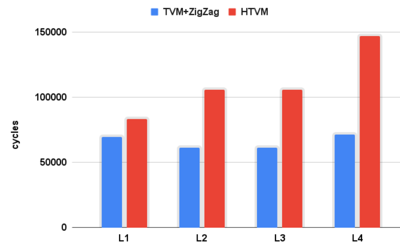

Figure 5.25: Results and comparison of the inference of layers of the third set (Tab. 5.3) compiled with TVM+ZigZag and HTVM.

tile only the K dimension, resulting in a noticeable improvement due to the data layout.

Finally, comparing this new configuration of TVM+ZigZag with HTVM over the three sets, significant improvements have been observed. When considering only

Temporal Loops	O	W	I
for K in [0:8)	dram	dram	act_mem
for OY in [0:16)	act_mem	weight_mem	act_mem
for OX in [0:4)	act_mem	weight_mem	act_mem
for C in [0:16)	rf_4B	weight_mem	act_mem

Figure 5.26: Scheduling produced by HTVM for L3 of set A (Tab. 5.1).

Temporal Loops	O	W	I
for K in [0:4)	dram	weight_mem	act_mem
for OY in [0:2)	dram	weight_mem	act_mem
for OX in [0:4)	act_mem	weight_mem	act_mem
for OY in [0:8)	act_mem	weight_mem	act_mem
for C in [0:16)	rf_4B	weight_mem	act_mem

Figure 5.27: Scheduling produced by TVM+ZigZag for L3 of set A (Tab. 5.1).

the layers that are part of the first set (Tab. 5.1), the results (Fig. 5.23) show an average gain of 20.35%. Increasing the activation memory size limits brings even more substantial improvements. In fact, for the second set (Tab. 5.2) with 64KB of activation memory, TVM+ZigZag achieves an average speed-up of 34.55% (Fig. 5.24). On the last set (Tab. 5.3), the average gains reach as high as 38% (Fig. 5.25). Overall, TVM+ZigZag can achieve a 55% peak improvement over a single layer compared to HTVM.

TVM+ZigZag vs HTVM - Single Layer Analysis

To gain a clearer understanding and uncover the main differences between the two tools, we can analyze one of the layers and its scheduling by both tools. The selected layer for this analysis is part of the first set (Tab. 5.1), which is L3, and it presents significant challenges for the memory allocator.
HTVM doesn't support unevenly mapped schedules. Therefore, to avoid costly input transfer overheads, it prefers to tile only the K dimension (Fig. 5.26). This choice is influenced by DORY's heuristics, which penalize tiling over certain dimensions. However, this decision results in suboptimal accelerator usage, reaching only 50%. On the other hand, TVM+ZigZag, thanks to its support for uneven mapping, offers a more flexible memory allocator that can tile the K dimension in a way that fully utilizes the accelerator and uses the OY dimension only for the output (Fig. 5.27), keeping transfer costs low.

5.2 MLPerf Tiny

In this section, we'll analyze the results achieved using entire networks rather than sets of single layers. Specifically, we target the MLPerf Tiny benchmark suite, which includes four testing networks. Due to the lack of support for some layers, such as depthwise convolution, currently, TVM+ZigZag can only partially compile three of these models.
The three networks that TVM+ZigZag can partly support are ResNet, MobileNet, and DAE.

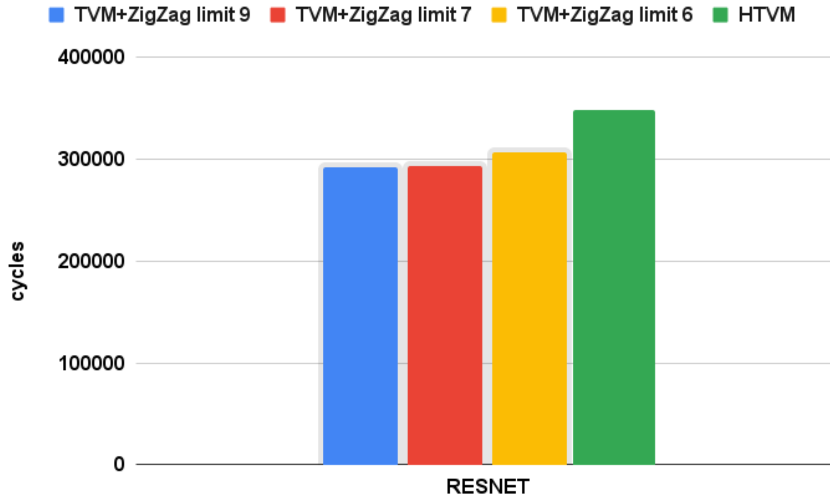

Figure 5.28: Results of the inference of ResNet with three configurations of TVM+ZigZag, each with different LPF limit values, and HTVM.

These networks have different degrees of support, which impact the performance results.
For TVM+ZigZag, three configurations with different user limit values have been used. The limit refers to the maximum number of LPFs (Loop Prime Factors) that can be generated, and this value affects the size of the search space. The three configurations range from the LPF limit of 9 used throughout the thesis to 7, representing a balanced trade-off between optimality and time requirements, and finally to 6, which imposes a significant limit on the search space for some layers but results in better compilation times.

5.2.1 ResNet

The ResNet [28] network consists of a total of 13 fused layers that are supported by the DIANA digital accelerator. Currently, TVM+ZigZag can compile six of these layers, while the remaining seven are processed by the pre-existing DORY module. The unsupported layers include convolutional layers with a stride greater than 1, which are not yet fully functional, and layers with other operands not yet implemented by TVM+ZigZag.
Despite having only half of the layers compiled by TVM+ZigZag, the results (Fig. 5.28) are already impressive. TVM+ZigZag configurations with a higher LPF limit achieve a speed-up of 16% compared to HTVM, while even with a smaller search space due to the fixed LPF limit, TVM+ZigZag still outperforms HTVM by

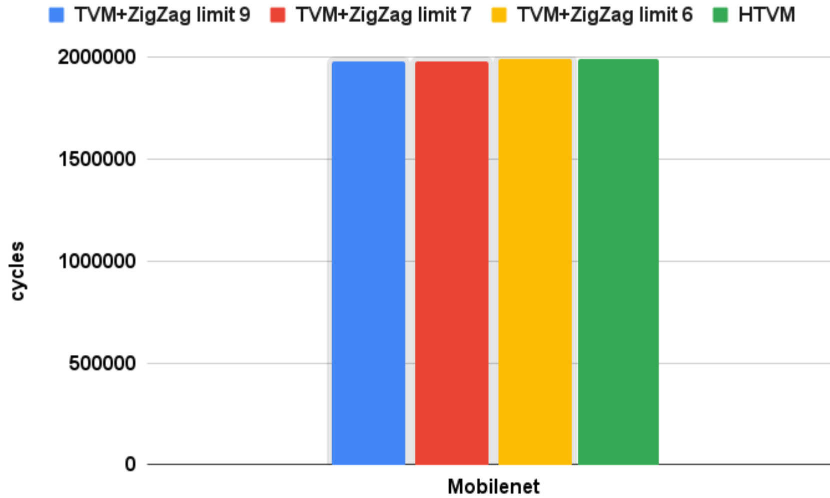

Figure 5.29: Results of the inference of MobileNet with three configurations of TVM+ZigZag, each with different LPF limit values, and HTVM.

11.67%.

5.2.2 MobileNet

MobileNet [29] contains several depthwise layers that are not supported, as well as other operators like pooling and add. As a result, TVM+ZigZag can only compile one of the layers in MobileNet, which is a 2D fused convolutional layer.
The level of support significantly impacts the results (Fig. 5.29). We can see that with an LPF limit of 6, the latency obtained is identical to that of HTVM, while with more freedom in the search space due to higher LPF limits, the remaining two TVM+ZigZag configurations achieve a 0.6% gain in performance.

5.2.3 DAE

The last network in this benchmark is DAE [30], which mainly consists of fully connected layers. Out of the ten layers, TVM+ZigZag compiles eight, while the remaining two are processed by DORY.
The results (Fig. 5.30) demonstrate significant performance gains of 16.5% compared to HTVM for the higher-end TVM+ZigZag configurations. However, when using a smaller LPF limit like 6, HTVM outperforms TVM+ZigZag by 8%. This discrepancy is due to the current landscape of the fully connected layers in TVM+ZigZag.

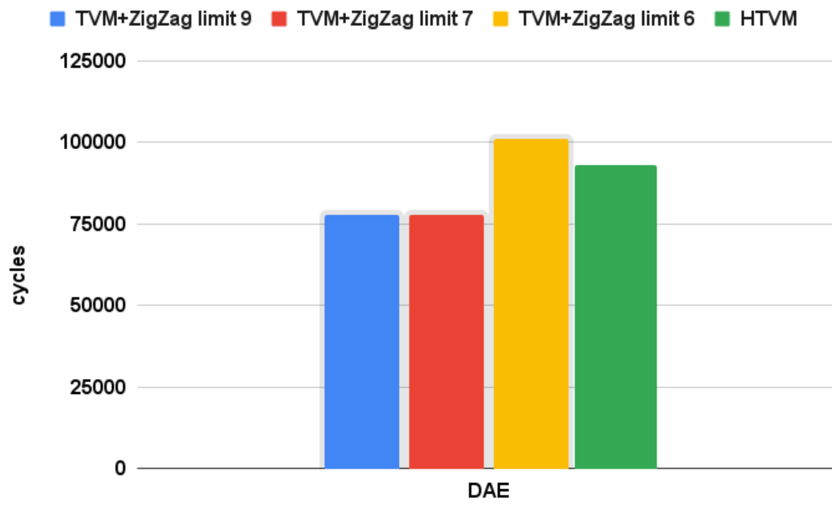

Figure 5.30: Results of the inference of the DAE network with three configurations of TVM+ZigZag, each with different LPF limit values, and HTVM.

Chapter 6

Conclusions and future works

In the current landscape DNNs are almost exclusively executed on the cloud, with server class GPUs and other high-end dedicated hardware. This has been the case since the first implementations of AI, but it brings with itself some negative aspects, like privacy concerns, unpredictable latencies, network pressure and high energy consumption, which could be solved by a local execution of the AI tasks. To execute successfully DNNs on the edge it is necessary to change methodologies due to the natural differences between cloud and edge hardware.
The deployment of DNNs can be very challenging when dealing with edge heterogenous devices. The difficulties are brought the current landscape that forces each hardware vendor to develop a complete compiler stack, thus limiting portability of the models.
The aim of this thesis was the integration of ZigZag onto TVM, extending the state-of-the-art work, targeting edge heterogenous devices, that for the analyzed experiments was DIANA.
While working on this topic during the thesis it is quite evident how hard is it to develop a portable solution that can be implemented by several AI hardware vendor. The variety in the landscape of AI devices is tremendous and it make it really hard to optimize a network in a sensible way for several architectures. ZigZag helps with this point by simplifying a lot the schedule generation aspect for any architecture.
A key point that emerged during this work has been the importance of hardware-aware information for tools like ZigZag to optimize the quality of the results. It is crucial to leverage this information while at the same time keeping the interface between the developers and the framework quite user-friendly.
During the thesis we conducted also a deep analysis on the results, where we

can see the impact that is brought by different scheduling, and therefore how important it is to search wholly the solution space. Additionally it was noticeable the impact of having a complete modeling of the costs, utilizing only heuristics can be beneficial in terms of schedule generation but this abstraction can deteriorize the final performance results.

Another impactful factor that emerged is the support to unevenly mapped schedules, which have been underrated up until now. These types of schedules can perform much better than even ones, that have many constraints and can require many more data transfers than required.

The work developed by this thesis has shown the importance of the optimization for the inference of DNNs on edge heterogeneous devices. This thesis can be extended by several future works. One of them can regard continuous work over this project to support the remaining layers, and also optimize the currently supported ones in an even better way. Some of the layers that can be supported are the depthwise one, the pooling one and also the add one. With this set of layers this tool could compile wholly the MLPerf Tiny benchmark suite and obtain even better results on these networks.

Also additionally to layer a future extension may be the support for other architectures such as GAP9 and other edge heterogeneous devices.

Another future development that could extend the current tool could be the inseriment of ZigZag's architecture generation in the process. This addition could helps significantly the co-design aspect of software and hardware regarding AI devices.

Finally the last proposed extension tackles the utilized tools, in fact it could be useful to experiment on the possibility of integration between MLIR and ZigZag. Or more generally experimenting with other Deep Learning compiler technologies.

Bibliograpy

[1] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. *Zero-Shot Text-to-Image Generation.* arXiv.org. Feb. 24, 2021. URL: https://arxiv.org/abs/2102.12092v2 (visited on 10/15/2023) (cit. on p. 1).

[2] Komala C R, Krishan Kumar Bhushan, Prasanna Hridaynadan Anthony, Hussainkhan A. Jahagirdar, Apurva Mahadore, and Syed Mustafa. «Multimedia generation using neural network DeepDream». In: Book Title: 2023 International Conference on Advances in Electronics, Communication, Computing and Intelligent Information Systems (ICAECIS) ISSN: 1017-0138. IEEE, 2023, pp. 167–172. ISBN: 9798350348057. DOI: 10.1109/ICAECIS58353.2023.10170138. URL: https://ieeexplore.ieee.org/document/10170138 (visited on 10/15/2023) (cit. on p. 1).

[3] Tianqi Chen et al. «{TVM}: An Automated {End-to-End} Optimizing Compiler for Deep Learning». In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018, pp. 578–594. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/chen (visited on 08/22/2023) (cit. on pp. 1, 24).

[4] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. «Bring Your Own Codegen to Deep Learning Compiler». In: *arXiv.org* (2021). Place: Ithaca Publisher: Cornell University Library, arXiv.org. ISSN: 2331-8422. DOI: 10.48550/arxiv.2105.03215 (cit. on pp. 1, 25).

[5] Josse Van Delm, Maarten Vandersteegen, Alessio Burrello, Giuseppe Maria Sarda, Francesco Conti, Daniele Jahier Pagliari, Luca Benini, and Marian Verhelst. «HTVM: Efficient Neural Network Deployment On Heterogeneous TinyML Platforms». In: Book Title: 2023 60th ACM/IEEE Design Automation Conference (DAC). IEEE, 2023, pp. 1–6. ISBN: 9798350323481. DOI: 10.1109/DAC56929.2023.10247664. URL: https://ieeexplore.ieee.org/document/10247664 (visited on 10/12/2023) (cit. on pp. 2, 26, 27).

[6] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. «DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs». In: *IEEE transactions on computers* 70.8 (2021). Place: Ithaca Publisher: IEEE, pp. 1253–1268. ISSN: 0018-9340. DOI: 10.1109/TC.2021.3066883. URL: https://ieeexplore.ieee.org/document/9381618 (visited on 08/22/2023) (cit. on pp. 2, 25, 26).

[7] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. «ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators». In: *IEEE Transactions on Computers* 70.8 (Aug. 2021). Conference Name: IEEE Transactions on Computers, pp. 1160–1174. ISSN: 1557-9956. DOI: 10.1109/TC.2021.3059962 (cit. on pp. 2, 27–29).

[8] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. Aug. 13, 2017. arXiv: 1703.09039[cs]. URL: http://arxiv.org/abs/1703.09039 (visited on 08/22/2023) (cit. on p. 4).

[9] F. Rosenblatt. «The perceptron: A probabilistic model for information storage and organization in the brain.» In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519. URL: http://doi.apa.org/getdoi.cfm?doi=10.1037/h0042519 (visited on 10/15/2023) (cit. on p. 4).

[10] Zhao Yanling, Deng Bimin, and Wang Zhanrong. «Analysis and study of perceptron to solve XOR problem». In: *The 2nd International Workshop on Autonomous Decentralized System, 2002.* The 2nd International Workshop on Autonomous Decentralized System, 2002. Nov. 2002, pp. 168–173. DOI: 10.1109/IWADS.2002.1194667. URL: https://ieeexplore.ieee.org/document/1194667 (visited on 10/15/2023) (cit. on p. 4).

[11] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. Dec. 2, 2015. DOI: 10.48550/arXiv.1511.08458. arXiv: 1511.08458[cs]. URL: http://arxiv.org/abs/1511.08458 (visited on 10/12/2023) (cit. on p. 5).

[12] *Convolutional Neural Networks: Architectures, Types & Examples*. URL: https://www.v7labs.com/blog/convolutional-neural-networks-guide,%20https://www.v7labs.com/blog/convolutional-neural-networks-guide (visited on 10/12/2023) (cit. on p. 6).

[13] Mobeen Ahmad, Jooyeon Joe, and Dongil Han. «CortexNet: Convolutional Neural Network with Visual Cortex in human brain». In: *2018 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*. 2018 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia). June 2018, pp. 206–212. DOI: 10.1109/ICCE-ASIA.2018.8552151. URL: https:

//ieeexplore.ieee.org/document/8552151 (visited on 10/15/2023) (cit. on p. 6).

[14] Di Liu, Hao Kong, Xiangzhong Luo, Weichen Liu, and Ravi Subramaniam. «Bringing AI to edge: From deep learning's perspective». In: *Neurocomputing (Amsterdam)* 485 (2022). Place: Ithaca Publisher: Elsevier B.V, pp. 297–320. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2021.04.141 (cit. on p. 10).

[15] Hang Xiao, Haobo Xu, Xiaoming Chen, Yujie Wang, and Yinhe Han. «Fast and High-Accuracy Approximate MAC Unit Design for CNN Computing». In: *IEEE embedded systems letters* 14.3 (2022). Publisher: IEEE, pp. 155–158. ISSN: 1943-0663. DOI: 10.1109/LES.2021.3137335. URL: https://ieeexplore.ieee.org/document/9657057 (visited on 10/15/2023) (cit. on p. 12).

[16] Kodai Ueyoshi et al. «DIANA: An End-to-End Energy-Efficient Digital and ANAlog Hybrid Neural Network SoC». In: *2022 IEEE International Solid-State Circuits Conference (ISSCC)*. 2022 IEEE International Solid- State Circuits Conference (ISSCC). Vol. 65. ISSN: 2376-8606. Feb. 2022, pp. 1–3. DOI: 10.1109/ISSCC42614.2022.9731716 (cit. on p. 16).

[17] R. Maruthamuthu, Dharmesh Dhabliya, Kala Priyadarshini G, Ahmed H. R. Abbas, Abdullaeva Barno, and V. Vignesh Kumar. «Advancements in Compiler Design and Optimization Techniques». In: *E3S Web of Conferences* 399 (2023). Ed. by V. Vijayan and T.S. Senthil Kumar, p. 04047. ISSN: 2267-1242. DOI: 10.1051/e3sconf/202339904047. URL: https://www.e3s-conferences.org/10.1051/e3sconf/202339904047 (visited on 10/12/2023) (cit. on p. 18).

[18] Chris Lattner and Vikram Adve. «LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation». In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. CGO '04. USA: IEEE Computer Society, Mar. 20, 2004, p. 75. ISBN: 978-0-7695-2102-2. (Visited on 10/14/2023) (cit. on p. 19).

[19] Liang Hu, Xilong Che, and Si-Qing Zheng. «A Closer Look at GPGPU». In: *ACM computing surveys* 48.4 (2016). Place: Baltimore Publisher: ACM, pp. 1–20. ISSN: 0360-0300. DOI: 10.1145/2873053 (cit. on p. 20).

[20] Xiao Feng, Chen Shushan, Han Xingxing, Huang Shujuan, and Zhang Wenjuan. «A new direct acyclic graph task scheduling method for heterogeneous Multi-Core processors». In: *Computers & electrical engineering* 104 (2022). Publisher: Elsevier Ltd, pp. 108464–. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2022.108464 (cit. on p. 21).

[21] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. «DNN-Fusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion». In: *arXiv.org* (2021). Place: Ithaca Publisher: Cornell University Library, arXiv.org. ISSN: 2331-8422. DOI: 10.1145/3416510 (cit. on p. 21).

[22] Manpreet Singh Ghotra. *Neural network programming with tensorflow: unleash the power of tensorflow to train efficient neural networks*. In collab. with Rajdeep Dua. 1st edition. Birmingham, England ; Packt, 2017. ISBN: 978-1-78839-775-9 (cit. on p. 22).

[23] Wang Zhiguang and Gao Qingyun. «Design of DMA controller of PCIe bus interface based on FPGA». In: *Dianzi Jishu Yingyong* 44.1 (2018). Publisher: National Computer System Engineering Research Institute of China, pp. 9–12. ISSN: 0258-7998. DOI: 10.16157/j.issn.0258-7998.172445. URL: https://doaj.org/article/fb5d47b5d6964df78d6e6d5f606e3482 (visited on 10/12/2023) (cit. on p. 25).

[24] Arne Symons, Linyan Mei, and Marian Verhelst. «LOMA: Fast Auto-Scheduling on DNN Accelerators through Loop-Order-based Memory Allocation». In: Book Title: 2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS). IEEE, 2021, pp. 1–4. ISBN: 978-1-66541-913-0. DOI: 10.1109/AICAS51828.2021.9458493. URL: https://ieeexplore.ieee.org/document/9458493 (visited on 10/15/2023) (cit. on p. 30).

[25] Noah Arbesfeld. «Partial permutations avoiding pairs of patterns». In: *Discrete mathematics* 313.22 (2013). Publisher: Elsevier B.V, pp. 2614–2625. ISSN: 0012-365X. DOI: 10.1016/j.disc.2013.08.004 (cit. on p. 31).

[26] Colby Banbury et al. *MLPerf Tiny Benchmark*. Aug. 24, 2021. DOI: 10.48550/arXiv.2106.07597. arXiv: 2106.07597[cs]. URL: http://arxiv.org/abs/2106.07597 (visited on 10/12/2023) (cit. on p. 32).

[27] Hao Qian and Yangdong Deng. «Accelerating RTL simulation with GPUs». In: Book Title: 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) ISSN: 1092-3152. IEEE Press, 2011, pp. 687–693. ISBN: 978-1-4577-1398-9. DOI: 10.1109/ICCAD.2011.6105404. URL: https://ieeexplore.ieee.org/document/6105404 (visited on 10/12/2023) (cit. on p. 48).

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. Dec. 10, 2015. DOI: 10.48550/arXiv.1512.03385. arXiv: 1512.03385[cs]. URL: http://arxiv.org/abs/1512.03385 (visited on 10/12/2023) (cit. on p. 70).

[29] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. Apr. 16, 2017. DOI: 10.48550/arXiv.1704.04861. arXiv: 1704.04861[cs]. URL: http://arxiv.org/abs/1704.04861 (visited on 10/12/2023) (cit. on p. 71).

[30] Umberto Michelucci. *An Introduction to Autoencoders*. Jan. 11, 2022. DOI: 10.48550/arXiv.2201.03898. arXiv: 2201.03898[cs]. URL: http://arxiv.org/abs/2201.03898 (visited on 10/12/2023) (cit. on p. 71).