POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

A Machine Learning Approach to Optimizing CNN Deployment on Tile-Based Systems-on-Chip

Supervisors

Prof. Mario Roberto CASU

Prof. Luca CARLONI

Candidate

William BAISI

October 2024

Abstract

Convolutional Neural Networks (CNNs) play a crucial role in many AI applications, such as image recognition and classification. Efficient execution of CNNs on hardware accelerators is critical, particularly in edge computing, where performance, power efficiency, and real-time constraints must be balanced due to limited resources and strict power budgets.

This thesis presents an optimization framework for deploying CNN inference tasks on tile-based System-on-Chip (SoC) architectures. The study investigates various hardware configurations, including multiple accelerator tiles, memory bandwidth, computational capabilities, and on-chip local memory capacity, along with different parallelization strategies to efficiently distribute the CNN workload.

The experiments were conducted leveraging the Embedded Scalable Platform (ESP), an open-source, tile-based SoC architecture for heterogeneous computing. ESP allows for the integration of custom accelerators connected through a Network-on-Chip (NoC) and provides an automated flow to prototype designs on FPGAs, enabling efficient evaluation of different SoC configurations with various software applications.

CNNs exhibit significant variability in complexity across layers. For example, the memory footprint, the ratio of input feature maps (ifmaps) to weight parameters, and the computational intensity can vary substantially between layers. This heterogeneity, combined with the configurable nature of tiled architectures, introduces several trade-offs when optimizing deployment. Each CNN benefits from an optimal selection and distribution of on-chip resources and each layer in the network requires custom resource mapping to achieve optimal performance, making it challenging to determine the best resource allocation and mapping.

To address this, a dataset was collected from extensive FPGA experiments, capturing the execution latency of CNN inference tasks across different SoC configurations and mapping strategies. While heuristics could help find optimal mappings, this thesis adopts a Machine Learning (ML) approach, using models trained on empirical performance data to predict optimal mappings. Such models enabled the identification of complex relationships between hardware configurations, CNN topologies, and parallelization schemes that traditional heuristics may overlook. Models such as Random Forest and Extreme Gradient Boosting were trained to predict the execution latency of CNN layers mapped onto a given hardware instance. These models were then integrated into a mapping tool designed to select optimal configurations for executing CNN layers on the target SoC. Once trained, these models can generalize to networks with characteristics similar to those in

the training set, reducing the need for profiling new networks and speeding up the deployment process.

In conclusion, this thesis demonstrates that ML models trained on empirical data can optimize CNN deployment on tile-based SoCs, eliminating the need for complex system models or heuristics. By leveraging ML and ESP's automated flows, this work enables more efficient CNN deployment in edge computing environments.

Acknowledgements

First of all, I would like to thank my supervisors, Mario Casu and Luca Carloni. My time as a visiting student in the System-Level Design group was one of the most enriching experiences of my life, and it wouldn't have been possible without their support. They also provided me with valuable advice, guiding me throughout my project

I would also like to thank Gabriele Tombesi for the help and assistance he gave me during these months, using his time to guide me in my work. A special thanks to the entire SLD group for accepting me as a member and a friend—you have all been amazing.

Of course, I cannot forget my family and my beloved Chiara, as well as Paolo and Paola, for their endless patience and support. Lastly, my thanks go to my friends from LVG, as well as Giovanni and Daniele, for being such great companions throughout these years.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
Artificial Intelligence

ML
Machine Learning

NN
Neural Network

DNN
Deep Neural Network

CNN
Convolutional Neural Network

SoC
System on Chip

ESP
Embedded Scalable Platform

FPGA
Field-Programmable Gate Array

DSE
Design Space Exploration

PE
Processing Element

FC

Fully Connected

CPU

Central Processing Unit

GPU

Graphics Processing Unit

ALU

Arithmetic Logic Unit

SIMD

Single Instruction Multiple Data

GLB

Global Local Buffer

HLS

High Level Synthesis

PLM

Private Local Memory

MAC

Multiply-Accumulate

DSP

Digital Signal Processor

NoC

Network on Chip

TPU

Tensor Processing Unit

EWA

Element-Wise Accelerator

CONV2D

2D Convolution Accelerator

ASIC

Application-Specific Integrated Circuit

IFMAPS

Input Feature Maps

MT

Memory Tiles

BLAS

Basic Linear Algebra Subroutines

GeMM

General Matrix Multiplication

RF

Random Forest

XGB

Extreme Gradient Boosting

MLP

Multi-Layer Perceptron

PCA

Principal Component Analysis

PBM

Predicted Best Mapping

MBM

Measured Best Mapping

Chapter 1

Introduction

In recent years Artificial Intelligence (AI) has become of growing importance in our everyday life, transforming how we solve problems and improving our productivity. The general definition of AI refers to a machine's ability to mimic human thought and perform human tasks in real-world conditions. Machine Learning (ML) is a subset of AI that uses algorithms and data to automatically learn patterns and make decisions that solve a specific problem[1]. Inside ML great attention is given to Neural Networks (NN). The building block of a NN is, as in a human brain, the neuron, more precisely the mathematical representation of it. A biological neuron consists of three parts: dendrites, an axon, and a cell (soma). The dendrites are the ways the neuron has to receive inputs from other cells, the soma processes the information and the axon, if activated, forwards the processed output to other neurons. The structure of a biological neuron is shown in figure 1.1.

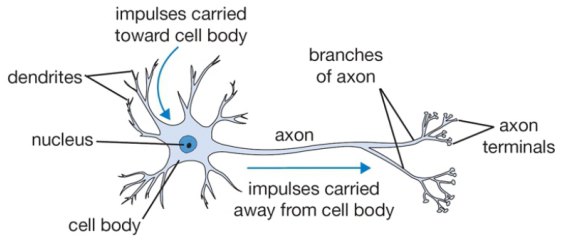

Figure 1.1: Biological Neuron [2]

The mathematical representation of this building block, shown in figure 1.2, is a set of N inputs that carry information from other cells or from initial inputs that are multiplied by N weights and added together. This weighted sum is added to a bias, which represents a possible offset, and is the input of a nonlinear activation function. The output of this function is the information that is propagated to other neurons.

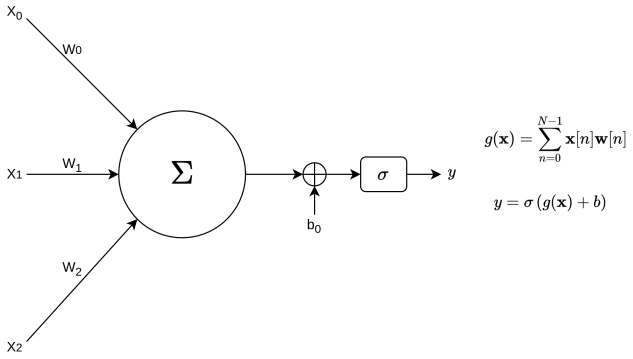

Figure 1.2: Mathematical representation of the Neuron

1.1 Multi Layer Perceptron

By concatenating three or more layers of neurons, we build what is known as a Multi-layer Perceptron (MLP). An MLP has a first layer that receives inputs, a certain number of hidden layers, and an output layer. Each pair of layers of neurons is known as a Fully Connected (FC) layer. A representation of the MLP is shown in figure 1.3.

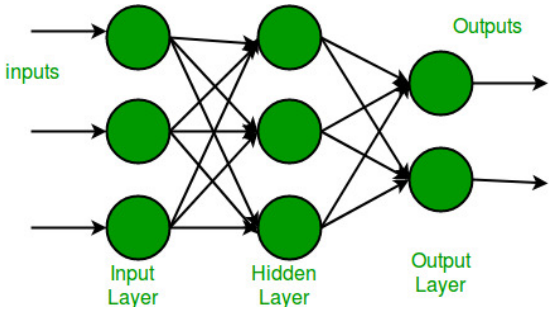

Figure 1.3: Multi Layer Perceptron [3]

Delving gradually into the mathematical aspect of a NN we can show that we can represent formally FC layer as vector-matrix multiplication, as shown in figure 1.4.

A NN with a large number of layers is called Deep Neural Network (DNN). DNNs are part of a bigger paradigm called Deep Learning (DL).

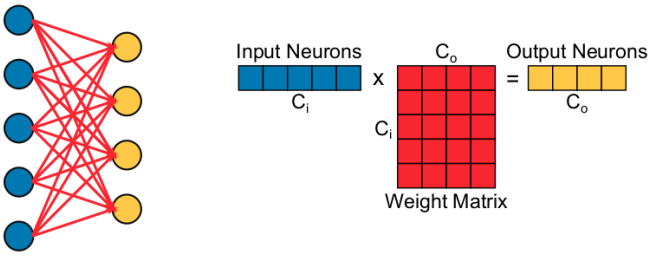

Figure 1.4: Fully Connected Layer [4]

1.2 Convolutional Neural Networks

The high degree of connectivity present in FC layers could lead to an explosion of parameters. Convolutional Neural Networks (CNNs) were introduced to address the problem of image recognition in high-resolution images [5]. The key idea of this type of network is to exploit the convolution operation to extrapolate features from the image. Biological studies demonstrate that some neurons activate when there is a vertical line in their receptive field while others neurons activate when the line has some different angle. To apply this concept in a Convolutional (CONV) Layer, a series of weights are convolved with the inputs to extract different kinds of features, and the outputs are forwarded to the following layer. Early layers detect low-level patterns like dots and lines while, as we move toward the end of the network, higher-level features like objects are targeted. The output of the network is a vector representing the probability of the presence of a given classification object in the image.

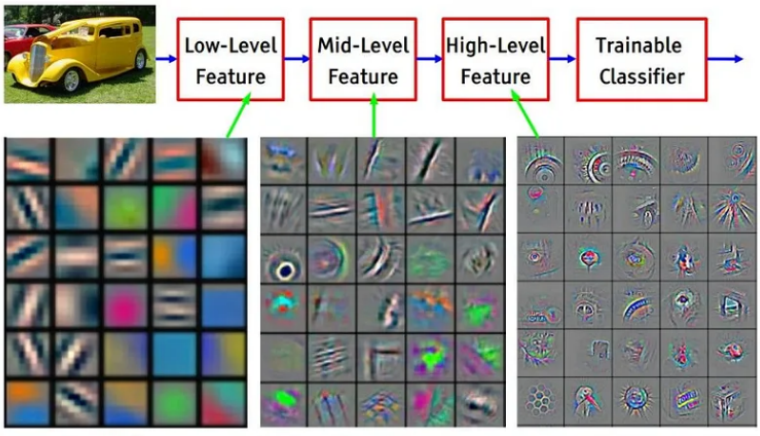

Figure 1.5: Evolution of the detected features towards the network [6]

Figure 1.5 provides an overview of a CONV Layer. The layer is characterized by a certain number of input and output channels of a given size, together with a filter dimension. For each output channel, a set of filters is convolved with the input feature maps, producing an output feature map. This operation is shown in figure 1.6.

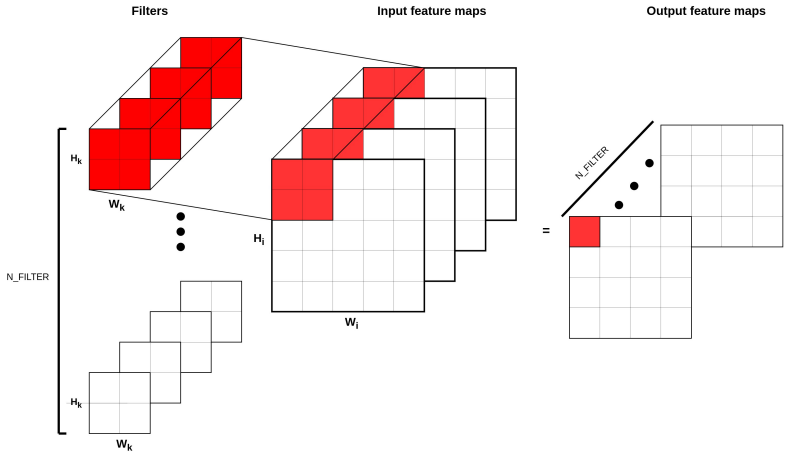

Figure 1.6: Convolution layer

1.3 Thesis focus

The thesis focuses on optimizing the deployment of CNNs on tile-based System-on-Chip (SoC) architectures, specifically for edge computing, where performance, power efficiency, and real-time constraints are critical. This work addresses the challenge of efficiently executing CNN inference tasks on resource-limited hardware accelerators.

To achieve this, the study explores different hardware configurations, such as multiple accelerator tiles, memory bandwidth, and local memory capacity, alongside parallelization strategies to distribute CNN workloads effectively. The experiments conducted leverage the Embedded Scalable Platform (ESP), a tile-based SoC architecture, which allows rapid prototyping of customized SoCs evaluating them on Field Programmable Gate Array (FPGA).

CNNs exhibit variability across layers in terms of memory and computational intensity, making efficient deployment complex. This work collects a comprehensive dataset from FPGA experiments to analyze the impact of different configurations on the end-to-end CNN execution latency. Finally, we adopt an ML approach based on Random Forest and Extreme Gradient Boosting models to predict optimal hardware mappings for CNN layers.

These data-driven techniques enable the selection of optimal configurations, speeding up the deployment process, eliminating the need for complex heuristics, and mitigating the need for system-level modeling. This approach improves resource allocation and performance, making it ideal for edge computing scenarios.

1.4 Thesis Outline

In the following, we provide the outline of each chapter to guide the reader through this work.

- **Background on Neural Network Acceleration** This chapter discusses the main contributions in the field of Neural Network acceleration, with a particular focus on tile-based SoC architectures designed for CNN acceleration. We delve into the architecture and workflow of the ESP, the tile-based SoC used as the target for CNN acceleration in this work. At the end of the section, we present state-of-the-art techniques employed for predicting Neural Network latency.
- **Accelerator description** This chapter presents the 2D-Convolution accelerator used in the experiments. Since we treated the accelerator as a black box, we introduce its high-level features as well as its integration within the ESP architecture.
- **Acceleration of CNNs on ESP** In this chapter, we describe how we accelerated different CNNs on ESP, explaining the parallelization strategies supported and the flexibility in the resource allocation, such as the number of accelerators and memory tiles. We then describe the structure of the software application and the performance obtained.
- **Experimental Setup and Dataset creation** This chapter focuses on how we built the dataset by profiling CNN inferences on FPGA and the subsequent post-processing steps to prepare the data for use with the learning algorithm.
- **Neural Network Mapping Tool for SoC Architecture leveraging ML Algorithms** This chapter provides an overview of the optimization framework developed to implement a Mapping Tool which optimizes the execution of CNN layers on ESP tile-based SoC instances, by providing mappings that guarantee lower latency. In the second part of the chapter, we describe the methodology and the models used for the latency prediction task, along with the performance evaluations.
- **Conclusions and Future Works** In the final section, we present the conclusions and discuss potential improvements to this work within the broader context of leveraging machine learning for hardware design space exploration.

Chapter 2

Background on Neural Network Acceleration

In this section, we first provide a comprehensive overview of the existing approaches for accelerating Deep Neural Networks, with particular emphasis on CNNs. We then focus on tile-based accelerators, a widely adopted class of accelerators built on arrays of Processing Elements (PEs). We introduce ESP as the target platform for developing tile-based accelerators in this work. In the final section of this chapter, we highlight state-of-the-art work on modeling inference using Machine Learning approaches.

2.1 List of existing approaches

Neural networks are algorithms that exhibit inherent parallelism. There is topological parallelism in the neurons of FC and CONV layers, as the Multiply-and-Accumulate (MAC) operations have no data dependencies. This parallelism can be exploited using parallel computing paradigms to enhance the performance of neural network hardware implementations.

There are two main types of architectures for parallel computation: temporal and spatial. An example of these two architectures is shown in figure 2.1. Temporal architectures feature centralized control, with multiple PEs accessing data solely from central memory and lacking interconnections. In contrast, spatial architectures provide each PE with control logic and local memory, and PEs are interconnected to exchange data, forming a processing array. [4]

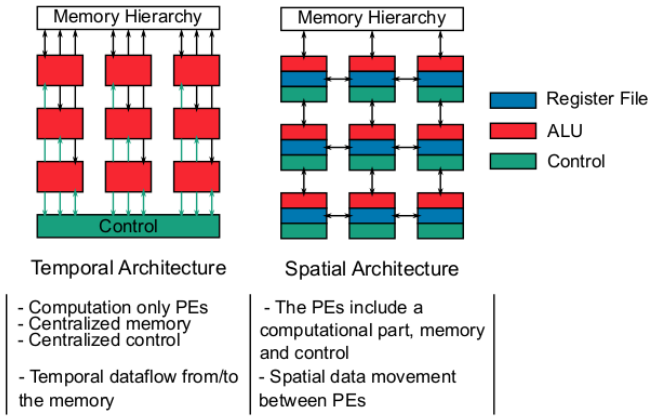

Figure 2.1: Spatial vs Temporal architecture [4]

2.1.1 Temporal architecture and software optimization approaches

Temporal architectures are commonly used in general-purpose platforms, such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs). Modern CPUs are designed as vector processors, where multiple Arithmetic Logic Units (ALUs) operate simultaneously under the Single-Instruction-Multiple-Data (SIMD) paradigm. However, CPUs typically reach lower performance in terms of Floating Point Operations (FLOPS) and FLOPS/WATT when parallelizing DNNs. This is mainly because achieving high performance in accelerating a specific algorithm requires highly specialized hardware, whereas CPUs are designed with a general-purpose architecture, leading to more complex data control compared to GPUs.

GPUs are the most used architecture in training DNNs and, in some cases, they're used for inference as well. Among the various GPU manufacturers, Nvidia has made significant strides in optimizing both GPU hardware and software for DL. Most DL frameworks, such as PyTorch, TensorFlow, and Caffe, support execution on Nvidia GPUs. A major advantage is the cuDNN library, which offers a highly optimized set of primitives for DNNs. In addition to cuDNN, Nvidia provides an entire suite of libraries for DNN/ML, known as CUDA-X AI.

At the software level, several libraries have been developed to optimize Basic Linear Algebra Subroutines (BLAS) for both CPUs and GPUs. These libraries include essential operations like element-wise matrix multiplication, matrix-vector multiplication, and General Matrix Multiplication (GeMM). For neural networks, BLAS is particularly useful for optimizing the FC layer, which can be represented as either a vector-matrix multiplication or a matrix-matrix multiplication in batched computations.

Optimizing the computation of CONV layers is more complex. While operations between a weight kernel and subsets of the IFMAPS are simple, the memory access patterns are much more challenging since we need to access non-contiguous memory locations. Several computational transformations have been proposed to optimize the application of BLAS to CONV layers. Many software libraries map convolution operations into a General Matrix Multiplication (GeMM), as shown in figure 2.2. This method is highly efficient because the GeMM routine is extensively optimized. However, it requires duplicating the input data, as a result, this approach demands substantial memory for temporary storage.

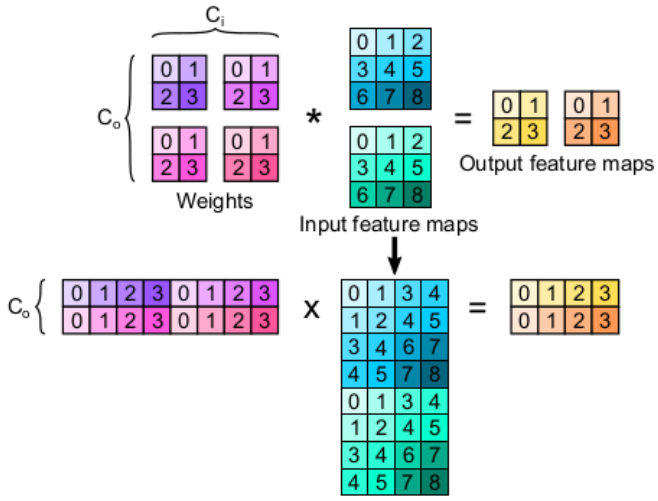

Figure 2.2: Convolution to GeMM transformation[4]

The GeMM-based approach for Convolution can be further optimized using the Strassen algorithm, which reduces the number of necessary multiplications by partitioning the matrices. This technique decreases the number of multiplications by 1/8 with each partition, though it introduces a higher number of additions as a trade-off. Another approach involves transforming both the IFMAPS and the weights from the spatial domain to the frequency domain using the Fast Fourier Transform (FFT) algorithm. In the frequency domain, the CONV operation simplifies to element-wise matrix multiplication. However, the FFT algorithm incurs significant computational overhead due to the domain transformation, and its efficiency has been demonstrated primarily for large weight kernels and unitary strides.

2.1.2 Spatial architectures

Spatial architectures, commonly implemented on FPGAs and Application Specific Integrated Circuits (ASICs), offer high performance but limited flexibility. Neural networks are particularly well-suited for this hardware due to the fixed and predictable sequence of operations in each layer, enabling specialized and optimized circuit designs.

Although the operations in neural networks are relatively simple, primarily consisting of multiply-and-accumulate (MAC) operations, there is a challenge related to the large volume of data that must be processed. The performance bottleneck is often the memory access bandwidth required to retrieve parameters. Each MAC operation involves reading three data elements (input, weight, and partial sum) and writing one (updated partial sum). DRAM accesses are particularly expensive in terms of power and latency, consuming roughly two orders of magnitude more energy than a MAC operation. The high energy cost associated with memory access has been noted in many modern DNN accelerators, such as DianNao [7].

A typical hardware architecture for a DNN accelerator includes the following components:

- **Off-chip memory:** The off-chip memory is generally implemented as DRAM and it stores the entire network's weights and activations, often occupying several GBs of data.
- **On-chip global buffer (GLB):** This on-chip buffer is large enough to hold one or multiple chunks of each layer's weights and inputs, which are fed to a certain subset of the PEs instantiated in the system. Accessing the GLB is significantly more energy-efficient, typically requiring two orders of magnitude less energy than DRAM.
- **Array of PEs:** It comprises hundreds of PEs, each equipped with an ALU to perform MAC operations in parallel. PEs usually have local storage in the form of Private Local Memories (PLMs), which offer even lower energy access costs compared to the GLB.
- **Network-on-Chip (NoC):** The array of PEs and the GLB are interconnected by a NoC, which is used to coordinate data movement across PEs based on the temporal scheduling and spatial distribution of operations. The NoC can be configured to work in different modes to handle different communication patterns.

Due to the high energy cost associated with off-chip DRAM accesses, state-of-the-art DNN accelerators focus on optimizing on-chip data reuse. This involves designing PE architectures, efficiently mapping the input dataset to multiple PEs,

and scheduling operations to maximize data reuse when stored in lower-level memories such as PLMs or the GLB.

Different layers of CNNs provide different opportunities for data reuse:

- **FC Layer:** An FC layer can be represented as a matrix-vector multiplication, offering an opportunity for input reuse. The input neuron vector is repeatedly multiplied with each row of the weight matrix.
- **CONV Layer:** The CONV layer provides three key data reuse opportunities:
  1. *Weight reuse:* A weight kernel is reused across multiple regions of the input feature map during the convolution process. Each kernel is reused $H_o \times W_o$ times for a given output feature map.
  2. *Input reuse:* The same input feature map is used C_o times to generate each of the output feature maps.
  3. *CONV reuse:* This leverages the sliding window mechanism, where adjacent output pixels share overlapping regions of the input feature map. The amount of overlap is determined by the kernel dimensions ($H_k \times W_k$) and the stride values (s_x, s_y). This reuse combines both weight and input reuse.

In a DNN accelerator, each PE performs a subset of MAC operations required to compute output feature maps. Multiple MACs are executed in parallel across the PE array, necessitating a spatial and temporal mapping of the operations. This mapping, known as dataflow, determines how data is loaded, stored, and moved within the memory hierarchy and across the NoC. The great number of MAC operations and the large PEs array lead to a wide space of possible mappings. This work navigates the mapping space with the goal of maximizing data reuse while minimizing memory accesses.

Different dataflows aim to optimize memory hierarchy efficiency by reusing data stored in lower-level memories, such as PLMs and the GLB, as much as possible. To classify dataflows based on their data reuse strategies, Chen et al.[8] introduced different categories:

- **Weight Stationary:** This dataflow focuses on reusing weights by keeping them stationary in the PLMs, while input data and partial sums are moved across the PE array. Examples include using input forwarding to maximize CONV reuse, as demonstrated by accelerators like nn-X and Google's Tensor Processing Unit (TPU) [9], which map FC layers to matrix multiplication units.
- **Output Stationary:** In this dataflow, the PEs accumulate partial sums locally, reducing the need for global memory accesses.

- **Row Stationary:** First introduced in Eyeriss[10], this dataflow maximizes the reuse of inputs, weights, and partial sums simultaneously. A PE keeps a row of weights stationary while input pixels are streamed in, with partial sums being accumulated along the columns of the PE array. This enables efficient 2D convolution operations by reusing data diagonally and vertically.

Each dataflow approach balances data reuse and memory access efficiency, with varying strategies depending on the layer types and operational needs of the PE.

2.1.3 The roofline model

We now introduce a useful model that helps understand the system-level behavior of complex architectures, particularly when dealing with mathematical functions that involve a large number of operations and transactions between the off-chip DRAM and the on-chip local buffer. The roofline model relates the total number of operations (OPS) of a given algorithm and the memory bandwidth to the performance achieved when executing the algorithm on the underlying hardware. We refer to the total number of byte transactions required by the algorithm from the off-chip DRAM as M_{tr}, and to the execution latency as T_{run}. The arithmetic intensity (AI) can be defined as:

$$AI = \frac{OPS}{M_{tr}} \tag{2.1}$$

The throughput (TH) is defined as:

$$TH = \frac{OPS}{T_{run}} = \frac{OPS}{N_{ClkCycle} \times T_{clk}} \tag{2.2}$$

The Memory Bandwidth (MBW) is calculated as:

$$M_{BW} = \frac{B}{s} \tag{2.3}$$

Each architecture provides a maximum achievable throughput dictated by the computation capabilities of its PEs, which we refer to as MAX_{TH} Depending on the operating region of the system, we can then express the throughput with the relationship:

$$TH = min(MAX_{TH}, AI \times M_{BW}) \tag{2.4}$$

.

By plotting throughput (TH) on the y-axis and arithmetic intensity (AI) on the x-axis using a log-log scale, we can identify a memory-bound region at low arithmetic intensities and a performance plateau at high arithmetic intensities when the available bandwidth is fully utilized. From this, we can draw a few important conclusions: if the workload has a low AI, it will place more stress on the memory bandwidth than on the PEs. Conversely, when the AI is high, it is beneficial to

parallelize across more PEs or enable the existing ones in the hardware to achieve higher performance.

Another key insight introduced by the roofline model is that the ideal operating point for the algorithm is the intersection between the memory-bound region and the computation-bound region. At this point, the algorithm is utilizing both the available bandwidth and the maximum computational capabilities of the PEs [11].

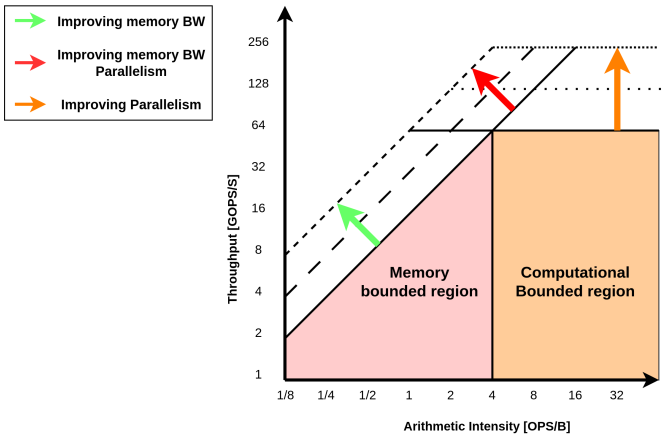

Figure 2.3: Qualitative representation of the roofline model

As shown in figure 2.3 we can see the memory-bounded region as well as the computational-bounded one. Moving up the ceiling of the computationally bounded part (orange arrow), we are increasing the throughput increasing the parallelism which can be effectively utilized due to the high AI of the algorithm. Increasing the memory bandwidth (green arrow) improves throughput in the memory-bounded region. Finally, we can increase both computational parallelism and memory bandwidth (red arrow), shifting the intersection point between the two regions upward.

2.2 Tiled Accelerators

Due to the high energy consumption and performance constraints associated with long array buses in monolithic accelerators, tiled architectures have emerged as a popular solution. Homogeneous tiled architectures enhance system-level performance by utilizing different parallelization strategies and/or specialized interconnections between tiles of the same type.[12, 13, 14, 15, 16, 17, 18, 19, 20].

In this section, more focus is dedicated to some major contributions in literature on tiled-based accelerators for DNN acceleration, particularly on Eyeriss and Google TPU. On top of the hardware architecture, some separate work use these

two accelerators as a building block for bigger architecture, optimizing mapping of networks on HW, focusing in particular on efficient data movement.

2.2.1 Eyeriss

Among the significant contributions to tile-based CNN accelerators is Eyeriss [10], which features a spatial architecture with 168 PEs organized into a four-level memory hierarchy. The structure of a PE is shown in figure 2.4. Data movement is prioritized through lower-cost levels, such as PE scratch pads and inter-PE communication, to minimize access to higher-cost levels, including the on-chi GLB and off-chip DRAM. Eyeriss introduces a CNN RS dataflow that adapts the spatial architecture to efficiently map computations for various CNN structures, optimizing energy usage. The design includes a NoC architecture that supports both multicast and point-to-point single-cycle data delivery to enable the RS dataflow.

Additionally, run-length compression (RLC) and PE data gating exploit the statistical occurrence of zero data in CNNs to improve energy efficiency.

The RS dataflow introduced by Eyeriss maximizes data reuse in terms of weights, IFMAPS, and partial sums. Each 2D convolution is organized into a set of independent 1D convolutions, which are executed in parallel and mapped onto the 12x14 matrix of PEs.

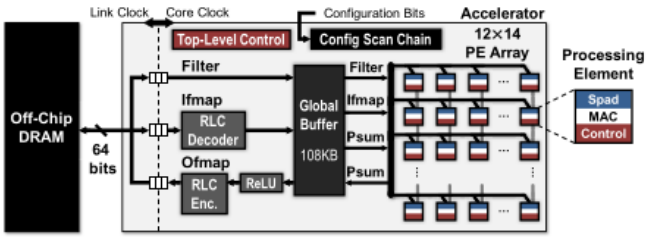

Figure 2.4: Eyeriss architecture [10]

The accelerator's configuration involves a scan chain of registers that serially reconfigure the hardware for executing a given layer. Each layer of the network is processed sequentially, and the accelerator can handle batches of IFMAPS from the same layer, which are processed consecutively. The total configuration time is approximately 100 μs.

During execution, the accelerator's goal is to optimize CONV reuse by reusing the same weight filter for a number of iterations equal to the output feature map dimensions and the same input pixel for a number of iterations equal to the kernel dimensions.

2.2.2 Google TPU

Another milestone in neural network acceleration is the Google TPU [9]. Unlike Eyeriss, which is specialized for CONV operations, the TPU utilizes matrix multiplication to support the execution of various types of layers with a unified dataflow. This application-specific hardware was developed to meet Google's need for a 10x improvement in cost-performance compared to GPUs. The architecture of the TPU, shown in figure 2.5, is similar to that of a floating-point unit, as the host server sends instructions to the hardware through a 256-byte-wide PCI bus.

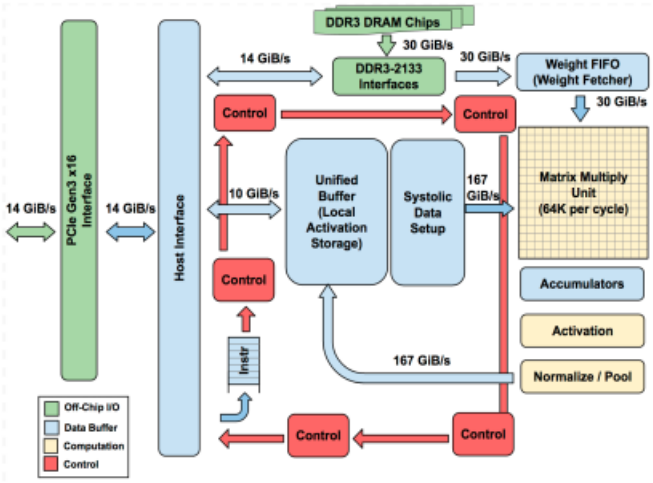

Figure 2.5: Google TPU architecture [9]

The matrix multiplication unit is the core of the architecture, it contains 256x256 MACs that execute 8-bit multiply and accumulate signed or unsigned integers. The products are connected to a 4 MB 32-bit accumulator that holds 4096, 256-element of 32-bit accumulation cells. The dimensions of the matrix of PEs was set to 4096 in order to be able to reach the peak performance on the roofline model accounting also for the use of double buffering. The peak performance can in fact be reached with half of the PEs working. In addition, the TPU is able to run half of the speed when the input features are 16-bit and the weights are 8-bit, or at 25% of the maximum speed handling 16-bit weight and INFMAPS. FC layers are mapped as vector-matrix multiplication or matrix-matrix multiplication when we are batching different inputs. As discussed in section 2.1.1 convolution can be mapped to a matrix-matrix multiplication paying the price of replicating some inputs.

Since the issuing of the instruction is relatively slow the architecture follows the CISC tradition having a number of Clock Per Instruction (CPI) in the range of 10-20.

The accelerator has several instructions, with the following five being the most

important:

1. **Read_Host_Memory**: Read the data from the CPU and store them in the Local buffer.
2. **Read_Weights**: Read the weight from the Memory into the Weight FIFO.
3. **MatrixMultiply/Convolve**: Perform the matrix-matrix multiplication with as a first operand a B*256 matrix with a 256x256 constant weight operand, computing the computation in B clock cycles.
4. **Activate**: Performs the activation operation: ReLU, Sigmoid, etc. The input is the Accumulator buffer and the output is the Local Unified Buffer.
5. **Write_Host_Memory**: The result are written back to the CPU host memory.

The performance of the TPU can be evaluated with the roofline model introduced in section 2.1.3, executing different algorithms. Starting from the assumption that the parameters necessariy to the execution of the networks can't fit the on-chip memory, the system is either memory-bounded or computationally bounded. The ideal point for the execution of an algorithm on a given hardware is the roofline knee, or the point where the memory bandwidth capabilities match the computational one.

In figure 2.6 are reported six different algorithms, we can see that MLPs and LSTMs are usually memory-bounded since the Arithmetic intensity is too low for exploiting efficiently the complete hardware capabilities. For the convolution the situation is different, we have in fact higher values of arithmetic intensity, and so the throughput is limited by the overall computational capabilities.

2.2.3 Tangram

Based on what we presented in the last two sections, we now introduce some important contributions that exploit the hardware architecture as a building block for a larger structure. In particular, Tangram addresses this problem by constructing a larger architecture using the Eyeriss accelerator as a foundational building block.

In the previous sections, we assumed a sequential execution of the layers. We now move to a more advanced paradigm. Specifically, we introduce intra-layer parallelism, which involves parallelizing the execution of the same NN layer across multiple accelerators, and inter-layer parallelism, which pipelines the execution of multiple layers across different tiles.

Scaling the number of PEs within a monolithic accelerator is inefficient, primarily because it is difficult to achieve a high degree of resource utilization. The natural

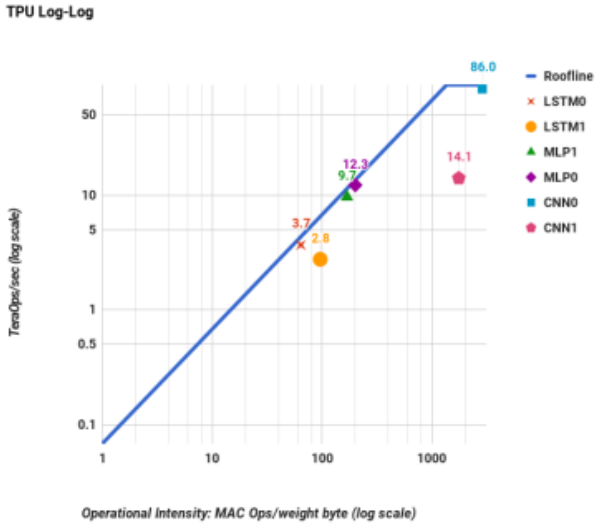

Figure 2.6: Google TPU roofline model [9]

solution is to introduce an array of monolithic accelerators, dividing parallelism into two different levels: a fine-grained parallelism within each accelerator, where the micro-architecture manages data movement between PEs, and a coarse-grained parallelism at the system level, leveraging a NoC for communication between different accelerators. However, with a large array of monolithic accelerators, the distance between memory and the internal PEs that perform MAC operations increases, which in turn raises the energy and latency costs. Tangram addresses this problem by introducing on-chip data movement that transforms the distributed local buffers within the accelerator into an idealized global buffer. First, we focus on the intra-layer parallelism implementation that uses the Buffer Sharing Dataflow BSD. In figure 2.7 we have an example of the data movement inside the system, in this example, the output parallelization of the convolution is used, this consists of the division of the weights across multiple accelerators that will calculate a subset of the output feature maps. Since accumulation is a commutative operation, the order of convolution execution does not matter. At T=1, the inputs are loaded into the three Eyeriss instances, and the partial outputs of the convolution are stored in their respective local buffers. Then, at T=2, the inputs are exchanged between the accelerators and another partial output is accumulated. In this scenario, an additional iteration is required for each of the three accelerators to complete its assigned output channels: specifically, O[0:2] for Engine 0, O[2:4] for Engine 1, and O[4:6] for Engine 2.

We can improve the data reuse across multiple accelerators using a two-dimensional array and move data across both axes, as shown in figure 2.8.

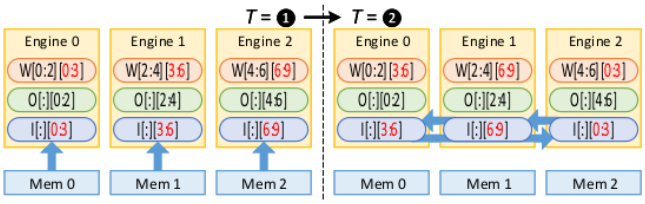

Figure 2.7: Tangram 1D BSD [13]

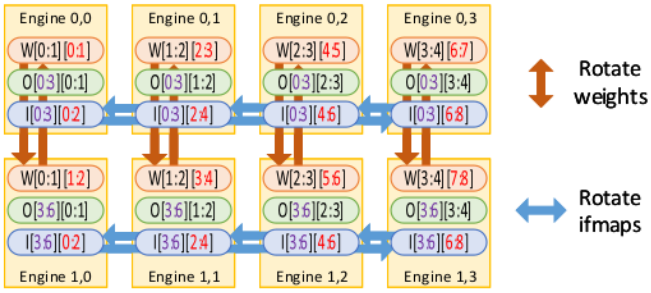

Figure 2.8: Tangram 2D BSD [13]

Focusing on inter-layer parallelism, Tangram proposes the Alternate Layer Loop Ordering (ALLO), which consists of spatially deploying multiple layers across the accelerator array in such a way that, as soon as some output features of the upstream layer are computed, they are sent to the downstream accelerator, allowing it to start its computation. This approach can only be applied to alternate pairs of layers because the output features produced by the downstream accelerator are continuously accessed.

In figure 2.9 we can see that L-3 needs to wait for the completion of L-2 in order to start his execution.

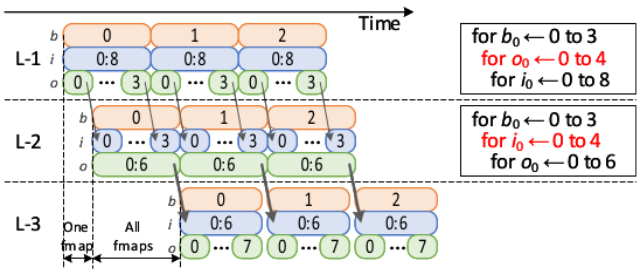

Figure 2.9: Tangram ALLO [13]

2.2.4 Planaria and the paradigm of multi-tenant execution

We focus now our attention on Planaria, a system that addresses the the problem of multi-tenancy on tiled-based SoC using as a building block a TPU-like architecture. Executing more than one network, sharing the resources of a single hardware between multiple layers, is a topic of crucial importance. With the increase of DNN applications, the need for data-center of flexible architecture able to execute more than a single layer at a time can improve the overall efficiency, in addition, new algorithms such as speech recognition or voice synthesis require multiple models deployed on the same hardware. PREMA[18] proposed a preemptive scheduler that can time multiplex the execution of multiple applications on the same hardware, but the idea in Planaria[14] is to design a specific accelerator for the multi-tenant execution.

The key points of a flexible DNN accelerator are first, the ability to split a monolithic array of PEs into sub-arrays that are effectively a new accelerator. Second, the implementation of the micro-architectural capabilities that can sustain such a structure and, finally a task scheduler able to leverage the introduced flexibility to maximize the throughput. Figure 2.10 highlights the ability to divide a monolithic array of PEs into different groups [14].

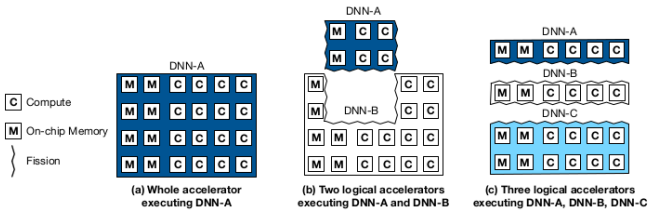

Figure 2.10: Planaria fissionable architecture [14]

In addition to the PEs fission the architecture should divide the local buffer capabilities, since the weight buffer is inside the PEs this comes for free, while the activation and the output buffer are implemented using a micro-architectural solution called Fission Pod. Figure 2.11 shows the complete Planaria architecture.

Planaria includes a compiler capable of generating binaries that can be executed by the architecture, along with tables that store the number of tiles involved in the layer execution as well as the corresponding latency. The proposed dynamic scheduling algorithm for a DNN accelerator leverages the dynamic architecture fission to optimize resource utilization and task co-location.

The steps followed by the scheduler are:

- Estimating Minimal Resources: The scheduler begins by identifying the minimum resources needed for each task to meet its Quality of Service (QoS) requirements using a task monitor and the configuration tables.

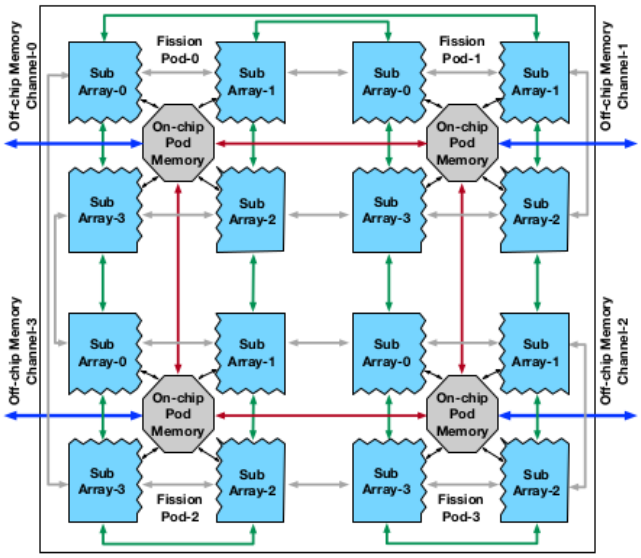

Figure 2.11: Planaria overall architecture [14]

- Resource Allocation:
  - If all tasks fit, the `ALLOCATE_FIT_TASKS` function assigns the necessary resources and distributes any remaining resources using a priority-based scoring function.
  - If only a subset of tasks can be co-located, the `ALLOCATE_UNFIT_TASKS` function handles competition using a scoring mechanism based on priority, slack time, and resource requirements.
- Tile-Based Scheduling: To minimize reallocation overheads, scheduling is performed at the tile level, and preemptions occur only when resource allocation changes, reducing memory requirements for storing intermediate results.

2.3 Embedded Scalable Platform as a Target Platform for Tiled Accelerators

The ESP is an open-source SoC designed for heterogeneous integration. It is the result of thirteen years of research and development at Columbia University, with contributions from other universities. Modern systems must handle a variety of algorithms, each with distinct hardware requirements, ranging from signal processing to artificial intelligence and cryptography. This broad range of applications has driven the semiconductor industry to transition from homogeneous multi-core

processors to heterogeneous SoCs, where traditional general-purpose processors are combined with highly specialized accelerators. ESP provides a scalable architecture that domain-specific designers can use to build complex SoCs, which can be implemented and tested on FPGA using an automated flow. Designers benefit from high flexibility when integrating new accelerators into the platform, supporting various design languages and synthesis tools. These include C/C++ with Xilinx Vivado HLS and Siemens Catapult HLS, SystemC with Cadence Stratus HLS and Siemens Catapult HLS, Keras TensorFlow, PyTorch, and ONNX with hls4ml, as well as Chisel, SystemVerilog, and VHDL for register-transfer-level design. ESP also supports the integration of third-party accelerators, such as the Ariane RISC-V core and the NVIDIA Deep Learning Accelerator [21].

2.3.1 The ESP Architecture

The ESP architecture is a tiled-based multi-core architecture where the designer can instantiate different types of tiles without sacrificing regularity, enabling an inherently scalable distributed system. The design environment includes a Graphical User Interface (GUI), shown in figure 2.12 that assists designers in interactive SoC floor planning, in a push-button manner for rapid FPGA prototyping.

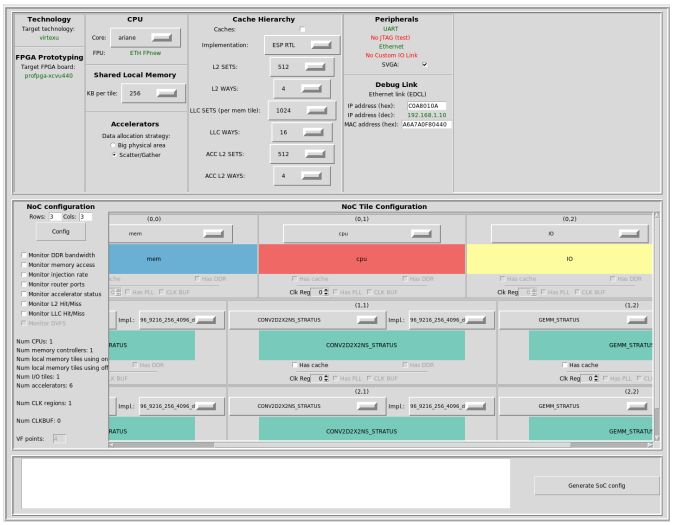

Figure 2.12: ESP GUI

A key architectural feature of ESP is its modular socket interface, which connects each tile to the NoC and provides platform services. This socket-based approach, which follows the latency-insensitive design paradigm, decouples the communication infrastructure from the internal component design, simplifying the integration of heterogeneous IPs for both IP designers and SoC architects. Such modularity

allows the designer to choose a set of services to be instantiated at design time or enable them at runtime in a simplified version [22].

2.3.2 Network on Chip

Every tile is connected with a packet-switched, multi-plane NoC that follows A MESI directory-based protocol and a 2D mesh topology. The NoC connects only with the socket and distributes messages across several physical planes to enhance accelerator performance while preventing protocol deadlocks. It supports coherence using three dedicated planes, ensuring deadlock-free communication. Two additional planes are allocated for handling Direct Memory Access (DMA) requests and responses between accelerator and memory tiles. Additionally, a separate plane manages IO/IRQ channels, primarily used for programming accelerators. Below is a breakdown of the message types handled by each plane:

1. **Coherence planes:**
    - Plane 1: Handles coherence requests from the CPU to the directory.
    - Plane 2: Manages forwarded coherence messages to the CPU.
    - Plane 3: Coherence bidirectional response messages from and to CPU.
2. **DMA planes:**
    - Plane 4: Manages bidirectional DMA responses to accelerators and coherent DMA requests from accelerators.
    - Plane 6: Handles bidirectional DMA requests from accelerators and coherent DMA responses.
3. **IO/IRQ planes:**
    - Plane 5: Remote Advanced Peripheral Bus (APB) interactions: Requests and responses between the processor and memory-mapped registers for configuration. Interrupt handling: Requests and remote acknowledgments between the processor and remote units. Advanced High-performance Bus (AHB) communications: Requests and responses to and from the processor's Debug Support Unit (DSU) for communication with the Debug Ethernet interface. Other AHB transactions: Communicate with additional peripherals of the IO tile, including UART 4 and a digital visual interface (DVI), enabling video output in a bidirectional manner.

The router architecture incorporates look-ahead dimensional routing, decoupling route computation from the critical path and executing it concurrently with

arbitration, ensuring each hop completes in a single clock cycle. This design is driven by the Transaction Level Modeling (TLM) abstraction, a high-level approach that simplifies inter-module communication by abstracting functional unit details. Functioning as a transparent communication layer, the NoC makes every SoC component appear as though connected directly to the local bus of the socket. This capability is enabled by proxy components within each tile, tasked with converting CPU and accelerator requests into packets suitable for NoC communication and restoring incoming NoC messages to their original form upon arrival. Each proxy is equipped with a buffering queue, respecting the latency-insensitive design principles crucial for shaping key aspects of NoC and socket interface design.

2.3.3 Latency insensitive design

Latency-insensitive design (LID) is the response to limitations within the synchronous design paradigm at the Register Transfer Level (RTL), which traditionally defines a digital system as a set of interacting modules composed of combinational logic and registers. These modules use input and internal state values each clock cycle to update their outputs. According to the synchronous paradigm, the computation within modules and the communication of computed values occur sequentially without overlap, known as the synchronous hypothesis.

This design approach faced challenges with the advancement of semiconductor technologies, particularly due to the impossibility of the global wire connecting modules to scale down as efficiently as local wires and logic gates. This discrepancy often turned interconnecting paths into critical paths due to increased resistive and capacitive delays, leading to numerous exceptions to the synchronous design principles. These exceptions required extensive modifications during CAD flow stages, without always achieving optimal results.

To address these challenges, LID relaxes timing constraints early in the design process before accurate global wire estimations are available. This methodology allows a system to operate correctly regardless of inter-module channel latencies, provided the functional modules are designed correctly. The key components of LID include:

- A **communication protocol** that makes inter-module channel communications independent of latency
- A **shell** interface that connects modules to latency-insensitive channels. This interface is adaptable to any module, ensuring flexibility.

In LID theory, two signals are considered latency-equivalent if they present the same sequence of valid data events, independent of the timing of these events. Additionally, if the system's functionality relies only on the order of these events, is

defined as a patient system. Latency-equivalent signals must maintain the sequence of informative events (valid data) despite the presence of stalling events (data packets without valid information).

Implementing this design involves creating a shell encapsulation around the module core. This shell includes a buffering stage and synchronization logic, allowing the module to interface with the latency-insensitive communication channel effectively. The system must ensure that the core is stallable, meaning it can pause to wait for new valid inputs indefinitely without losing its internal state.

The shell forwards valid data items from its input port to the core, allowing state updates and new outputs in the subsequent clock cycle. If an invalid data item arrives at an input port, the core stalls until a valid item arrives, while valid data at other ports is temporarily stored for later use. This methodology incorporates a latency-insensitive protocol using additional signals—a void bit and a stop bit—to manage data flow and backpressure within the NoC's various planes. The void bit differentiates valid from stalling data items, while the stop bit enables backpressure to manage buffer capacity effectively, preventing data overflow by halting valid data transmissions until the system stabilizes [23].

This design plays a critical role in the architectural decisions for ESP's NoC and socket interfaces.

2.3.4 ESP Tiles

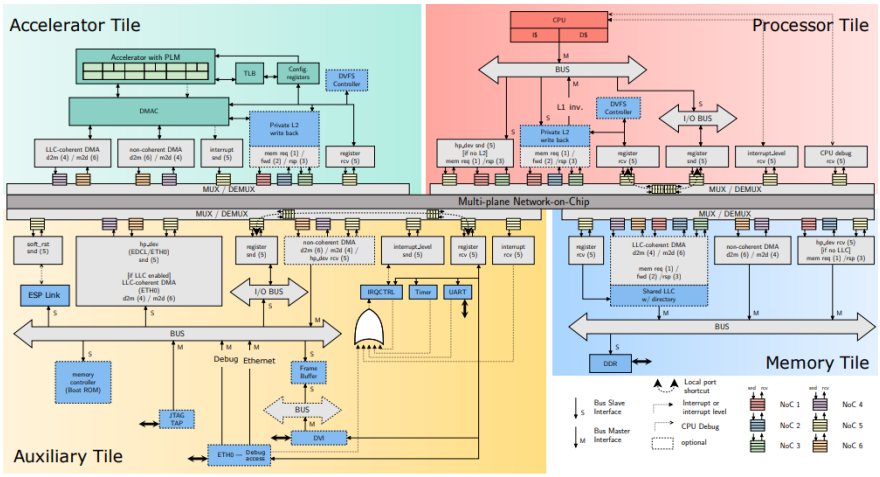

Figure 2.13: ESP Tiles [22]

In the ESP architecture, different types of sockets address different tasks, contributing to the versatility of the system. Figure 2.13 shows a detailed description of the tiles.

Accelerator Tile: Unlike many open-source RISC-V platforms, ESP assigns its accelerators their own dedicated tiles, giving them architectural parity with processors. These are known as loosely coupled accelerators. The classification of accelerators is primarily based on how they couple with the processor, impacting key behavioral trends such as local storage transparency, external memory access, and how the data are managed.

Processor-Centric Designs: Accelerators are tightly integrated within the processor pipeline, activated by control logic that decodes instructions from a specific ISA extension. This integration often limits the internal design due to space constraints, restricting storage capacity and the use of SRAM banks.

Co-processor Integration: As task complexity increases, designs may feature accelerators as co-processors, sharing significant resources with the processor but maintaining a distinct, tightly linked interface for data or instructions. High activity levels might necessitate direct bus system interfaces, utilizing mechanisms like DMA or cache coherency.

Loosely-Coupled Accelerators: Offering the highest flexibility, these accelerators operate independently of the processor, free from ISA constraints, and handle complex workloads with enhanced parallelism. This model facilitates the integration of external customized hardware into the system via the socket.

Loosely-coupled accelerators are the ones used in ESP, exploiting IP reuse across different SoCs and highlighting the platform's focus on flexibility and scalability. The accelerator tile in ESP supports various coherence protocols through its runtime selection capability:

Fully Coherent Model: Utilizes the existing memory hierarchy and an additional private L2 cache. Coherent-DMA Model: Operates without a private cache but maintains coherence with the system's other caches. LLC Coherent Model: Lacks a private cache and does not maintain coherence with the system. Non-coherent DMA Model: Entirely bypasses the cache hierarchy. The tile also houses a DMA controller that manages input and output requests under a latency-insensitive protocol and includes configuration registers for activating services without processor intervention.

Processor Tile: Each processor tile houses a core, selectable from a library that includes options like the RISC-V 64-bit Ariane and the SPARC 32-bit LEON3, each capable of booting Linux. These tiles integrate external IPs seamlessly, replicating the configurations of the accelerator tiles, with communication managed through designated bus systems (AHB for LEON3 and AXI for Ariane).

Memory Tile: Memory tiles serve as gateways to off-chip memory, each supporting a channel to external primary memory. The configuration of these tiles, from one to four, depends on the NoC's size, with each tile containing a portion of the last-level cache (LLC) and directory to support the MESI coherence protocol.

IO Tile: The IO tile manages off-chip communications and hosts essential

platform services, including the interrupt controller and interfaces for peripherals like Ethernet, which supports Secure Shell (SSH) logins and memory-mapped register access. A frame buffer for DVI video output, a UART console interface, and a timer for periodic tasks are also included.

2.3.5 The ESP Methodology

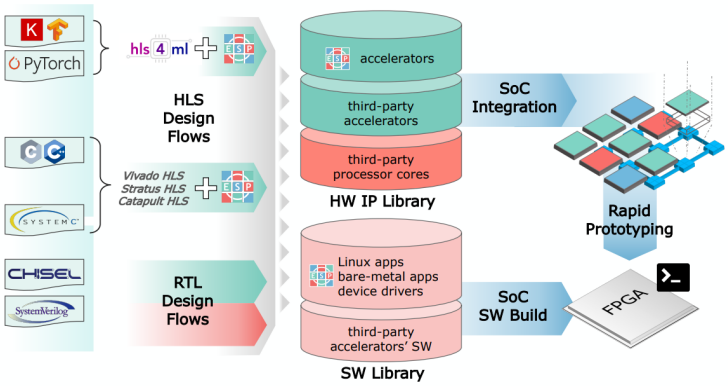

Figure 2.14: ESP design and integration flows[22]

The ESP methodology is driven by the need to sustain the semiconductor industry's advancement while managing the increasing complexity of SoC designs. A description of the flow is provided in figure 2.14. Currently, ESP supports multiple design flows, enabling designers at different abstraction levels to contribute to the platform's IP library. Specifically, ESP facilitates several ways for developing accelerators:

Traditional RTL Flow: Utilizes standard hardware description languages like Verilog, VHDL, or Chisel for cycle-accurate design. Commercial HLS Tools: Combined with ESP's in-house automation tools, these allow for the creation of accelerators from loosely-timed or untimed behavioral descriptions in C-like languages, primarily C++ and SystemC. Notable HLS tools supported include Xilinx Vivado HLS, Mentor Catapult HLS, and Cadence Stratus HLS. Open Source hls4ml Flow: This can be used to derive synthesizable accelerators from domain-specific libraries such as Keras, TensorFlow, and PyTorch. Among these, the HLS-based flow deserves particular attention as it shifts the focus from time-consuming, low-level RTL descriptions to higher-level system specifications. This transition is crucial as the complexity of modern SoCs can make features of the RTL flow lead to suboptimal designs, where significant changes are often required to optimize the architecture, increasing the risk of introducing bugs.

HLS leverages system-level synthesizable specifications to explore a vast design space with various micro-architectural configurations using a range of configuration knobs provided by advanced HLS tools. These include directives for function inlining, sharing, and loop pipelining. Timing constraints can also be applied to influence the number of states in the RTL code, unlike in synthesis where tighter timing constraints affect the choice of technology.

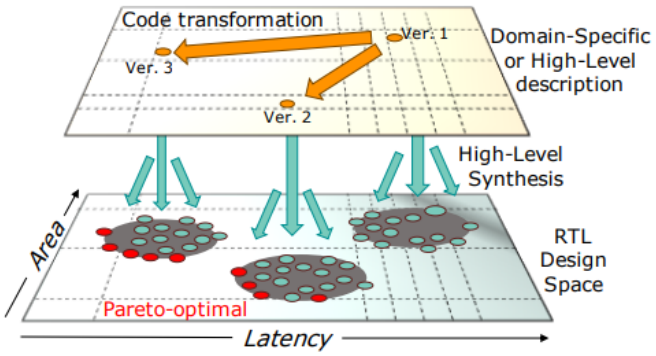

Figure 2.15: DSE with ESP [22]

With a high-level description, different settings passed to the HLS engine can generate multiple valid RTL implementations that correspond to distinct points in a multi-objective design space. It's crucial to note that these implementations are part of a latency-equivalent class and they can integrate within an ESP instance due to the latency-insensitive protocol of the tile sockets. Figure 2.15 shows the DSE leveraging HLS.

After choosing the best design option, a logic synthesis phase is needed to schedule and allocate resources for the hardware implementation, typically using the List-Scheduling algorithm. ESP plays an active role in this phase by providing accelerator templates to ease the design process and a fully functional accelerator skeleton based on specified parameters.

This HLS-based approach and its design space exploration at the application level are essential for designers looking to integrate an accelerator into an existing system with precise requirements. However, constructing an SoC from the ground up demands a wider consideration of parameters affecting the entire system's functionality, beyond just a single component. This challenge is addressed by ESP's FPGA-based rapid prototyping, enabling both hardware and software engineers to emulate and validate components within the complete system. This comprehensive approach supports hardware/software co-design, ensuring that all stages of SoC development are guided by the application-specific workload.

2.4 Existing approaches on CNN latency prediction

With the growing focus on using edge devices for time-critical application that uses deep learning algorithms, latency has become a critical factor when running DNNs. Accurately predicting the latency of DNN inference could be a very important feature, especially in scenarios where directly measuring latency on actual devices is impractical or costly. This is particularly important for tasks where an optimal resource allocation is crucial and profiling of all possible hardware configurations is impractical. Another important field where predicting latency is crucial is Neural Architecture Search (NAS) where ML is used for exploring the design space of the possible neural networks that can solve a specific task, choosing the optimal architecture given some constraints. In this scenario the execution latency will probably be one of the most important optimization goals, having a way of accurately estimating this parameter is a key point of the search. In this section, we analyze recent contributions in the prediction of latency of DNN inference on different platforms, in particular nn-Meter[24] that leverage Random Forest models to predict inference latency on different devices. Other works, such as PerfSAGE [25] and BRP-NAS [26] use Graph Neural Network to solve this problem.

2.4.1 nn-Meter

For addressing the problem of predicting latency on commercial edge devices such as CPU, GPU and, VPU, nn-Meter first of all divides the network graph in kernels. Each of these kernels represents either a single primitive or a fusion of multiple operations depending on the target device. Depending on the runtime optimization present on different devices some operations can be merged together or not. A concrete example is the Conv-Add that is fused on GPU, but not on CPU or VPU. This operation, along with nn-Meter architecture is shown in figure 2.16. First of all a dataset composed of 26,000 CNNs was created and then executed on different edge devices. The overall performance of the models is 99.0% for CPUs 99.1% for GPUs and 83.4% for VPUs.

The kernel detection is the first step performed by the system, it consists of the detection of primitives in the Direct Acyclic Graph (DAG), performed using Depth-first search (DFS), then using a set of rules that describe the fusion of the various operators in a kernel executed on hardware. All nodes of the graph have some properties that describe the type of computation (CONV, ADD, FC, Activations, etc.) and their topological connections to other nodes. In particular, each node has a single/multiple input/output edge. The fusion rules are influenced by the connection of the operators. If the DAG consists of two nodes with a single input and output and the backend supports the fusion, the two nodes will be fused

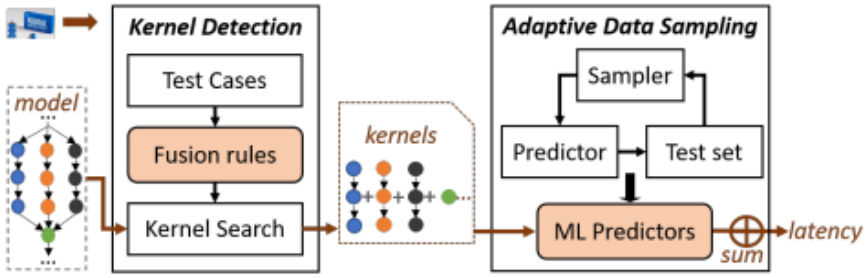

Figure 2.16: nn-Meter system architecture [24]

in a single kernel. This represent the simplest case, shown in figure 2.17.

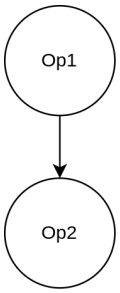

Figure 2.17: DAG with two nodes with single in/outbound

Another possible case can be when two nodes that have a single outbound are connected to a single node with two inbound. In this case, we can merge the node in two different ways (Op3, Op4) or (Op3, Op5) being careful to choose one out of two in order to avoid accounting two times for Op3. In this case, it could also happen that if Op5 is connected to Op4, we are breaking the DAG hypothesis and introducing a loop, this situation, shown in figure 2.18, must be avoided.

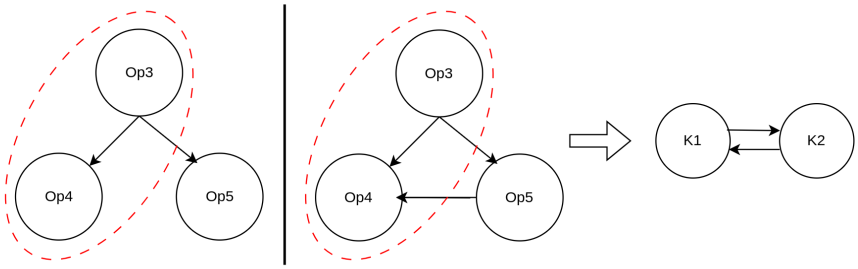

Figure 2.18: DAG with multiple in/outbounds

In figure 2.19 we can see the result of the primitive search and the integration in a DAG where nodes are a fused version that accounts for rules of integration specific of the target GPU.

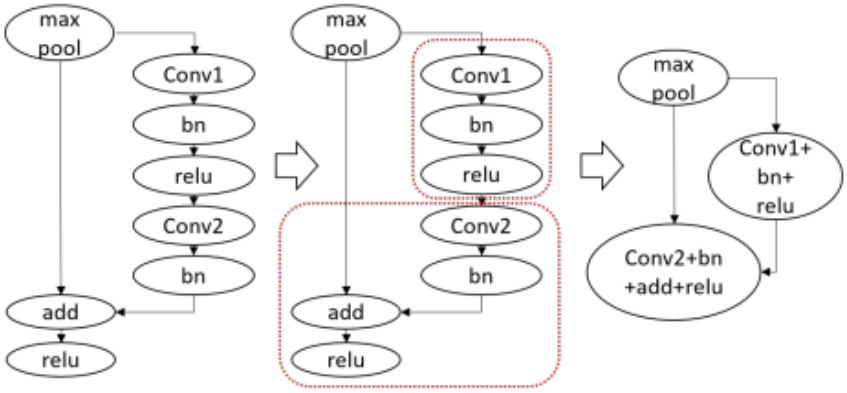

Figure 2.19: Kernel detection and fusing layer with a GPU as backend target. [24]

CONV and depthwise convolutional (DWCONV) kernels are the primary contributors to computational latency in CNN models. Analysis indicates that these kernels dominate the latency on various devices, accounting for approximately 94.2% on CPUs, 91.91% on GPUs, and 75.5% on Vision Processing Units (VPUs). FC and element-wise operations also contribute significantly to latency on VPUs.

Conv kernels have a vast number of possible configurations, primarily due to the wide range of input and output channel numbers (from 3 to 2160 in examined models). This results in an immense sample space of approximately 0.7 billion configurations, making exhaustive sampling impractical. Contrary to the assumption presented in some work of linearity between kernel configurations and latency, observations reveal non-linear and staircase patterns. Parameters like kernel size (K), input dimensions (HW), and channel numbers (Cin and Cout) do not have a straightforward relationship. This behavior reflects the underlying complex set of optimizations that influences the execution time in a nonlinear way. Random sampling configurations fail to capture critical data points, especially those reflecting hardware-specific optimizations. Missing these crucial configurations leads to inaccurate latency predictions.

In order to address these challenges nn-Meter uses an adaptive sampling between all the possible configurations in the design space, in particular analyzing state-of-the-art neural networks some configurations that are unlikely to be included in a model are removed from the high dimensional space. In addition, more sampling is performed around configurations that show higher prediction error.

To capture the non-linearities observed in the data, the model used is the Random

Forest Regression. Random Forests are ensemble methods based on decision trees that are very effective in solving this kind of problem. While some studies adopt XGBoost [27]—which requires tuning numerous hyperparameters—Random Forests are easier to optimize for high performance. For each kernel type, a different model is trained and saved, and his dedicated predictor will be used for inferring the latency.

Finally, the latency of each kernel is summed in order to find the overall latency of the network.

Chapter 3

Accelerator description

This section describes the CONV2D accelerator used in the experiments. This accelerator is an improved version of the CONV2D accelerator available in the ESP accelerator library, developed in the System Level Design group at Columbia University.

3.1 Convolution Accelerator

The CONV2D accelerator operates in 16-bit fixed point precision and it is designed in systemC and synthesizes with the high-level synthesis flow of ESP, leveraging Cadence Stratus HLS. The accelerator is loosely coupled [28], they are independent of the processor cores and they don't share any resources. Within a single invocation, the accelerator executes large workloads in a coarse-grained manner. The accelerator has a DMA interface designed to be latency-insensitive, as discussed in section 2.3.4. The accelerator has four main SystemC processes, they communicate with each other through a multi-bank and multi-port PLM.

- **Configure:** Configuration of the accelerator pipeline through memory-mapped registers.
- **Load:** leverage DMA interface for fetching data from memory, storing them in the PLMs.
- **Compute:** Core computation of the accelerator, reading data from input PLMs and storing them in the output PLMs.
- **Store:** Leverage the DMA interface for storing the computed results in the main memory.

The structure of the accelerator is described in figure 3.1. The accelerator exploits a double buffering technique using the so-called ping-pong memories. While a batch of inputs from the first PLM are computed the load process can fetch, from main memory, the next one and store the data in the second PLM.

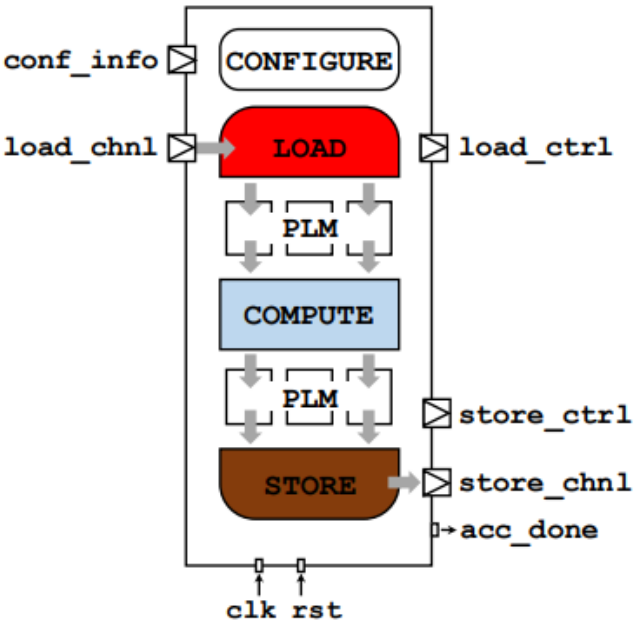

Figure 3.1: ESP accelerator structure[22]

The microarchitecture of the Conv2D accelerator is composed of three PLMs: one for IFMAPS, one for weights, and one for output feature maps. The IFMAPS and weights are fed into the accelerator's compute datapath two sets of registers equal in number to the MAC parallelism. The input collector features a Patch Extractor, while the weight collector retrieves weights from the weights PLM in parallel.

A multi-level dataflow is employed to maximize the reuse of both weights and input features. The accelerator adopts a weight-stationary-local-input-stationary (WS-LIS) dataflow [29]. This dataflow facilitates weight and input reuse, reducing both on-chip PLM and off-chip DRAM accesses. To enhance weight reuse, the accelerator loads part of the filter weights into the weight PLM and applies them across multiple IFMAPS, which are loaded in chunks. Once a filter chunk is processed, the next filter chunk is processed. The core of the computation relies on a MAC array that achieves maximum throughput, enabled by the number of ports in the PLMs to support parallel data access. The weights have a predictable access

pattern enabling the fetching of multiple filters weight per clock cycle. accessing IFMAPS in parallel is impractical without data duplication, so the Patch Extractor serially reads the required IFMAPS elements from the PLM into the input collector, also handling padding when necessary. To reduce the overhead from sequential IFMAPS accesses, the IFMAPS are reused across all filters within the current chunk for subsequent clock cycles.

The accelerator supports common configurations, in particular filter sizes 1, 3, 5, 7, strides 1, 2, and padding 1, 2, 3. Additionally, it integrates logic for in-place Batch Normalization, ReLU, and Downsampling (with max/average pooling for 2x2 or 3x3). This fusion of operations minimizes intermediate data storage needs and off-chip DRAM access, improving both performance and latency.

Chapter 4

Acceleration of CNNs on ESP

In this section, we focus on the CNN acceleration on ESP leveraging a varying number of available resources. Due to the high variations of the memory footprint of parameters and the number of operations across the layers of the network, it is crucial to optimize the deployment of each layer targeting the optimal combinations of CONV parallelization and hardware resources. Finding a balanced mapping that satisfies the memory and computational requirement is the task we will try to address in the remaining sections of the thesis. Figure 4.1 shows the memory footprint of weights and IFMAPS across the layers of ResNet18, together with the number of operations.

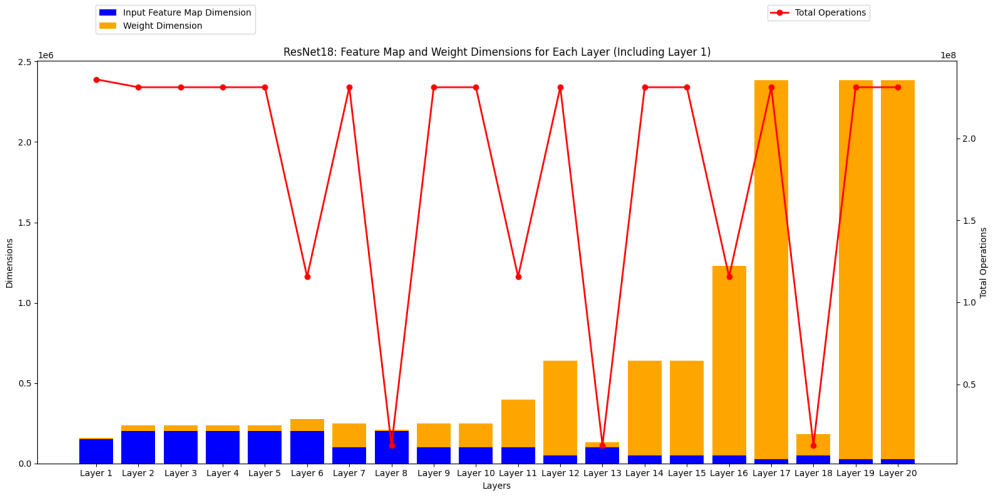

Figure 4.1: ResNet18 Parameters memory footprint and number of operations

4.1 Parallelization strategies

To parallelize the execution of the CONV layers on multiple accelerators, we leverage two different parallelization schemes: Output Parallelism (OUTP) and Input Parallelism (INPP). The main advantage of OUTP is that it splits the filters among the N convolution accelerators (N_{Conv2D}), resulting in each deployed accelerator producing N_{Filt}/N_{Conv2D} complete output features. The drawback is that convolving N_{Filt}/N_{Conv2D} filters requires duplicating the IFMAPS across each deployed accelerator as shown in figure 4.2 with the doubled orange feature map highlighted on the top left corner, which can lead to memory congestion as the number of accelerators accessing the same memory buffer region increases.

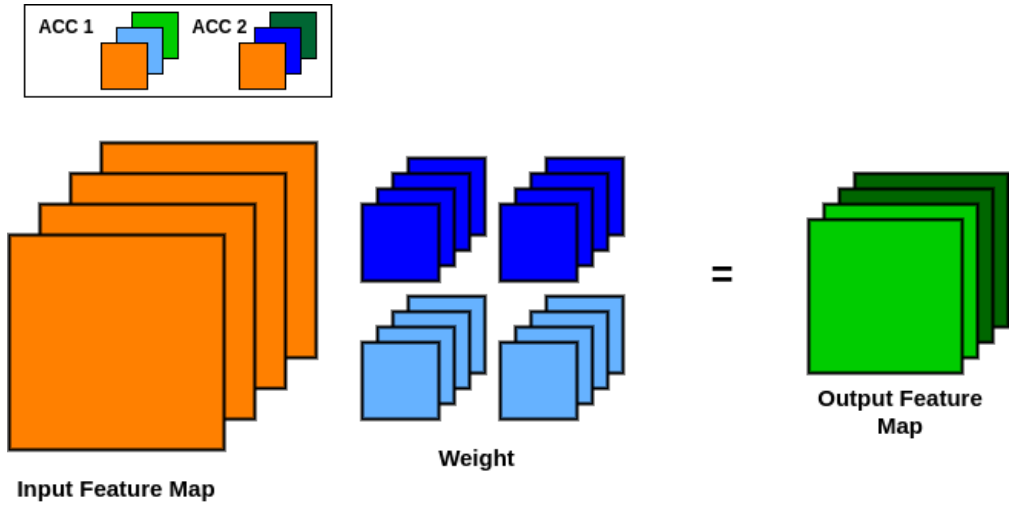

Figure 4.2: Output parallelism

On the other hand, INPP's main advantage is that the channels of the IFMAPS are divided among different accelerators. To convolve the filters with the reduced size of the IFMAPS, we also divide the number of kernels in each filter. To obtain the final convolution result, an element-wise addition must be performed on the partial sums produced by the N_{Conv2D} accelerators, as shown in figure 4.3. To speed up this operation, we employ a dedicated EWA. The element-wise addition is inherently an operation with very low AI, as it executes a single operation for each pair of transferred parameters. This property implies that, as the number of additions increases, the communication between the EWA and memory will be stressed, potentially saturating the memory bandwidth.

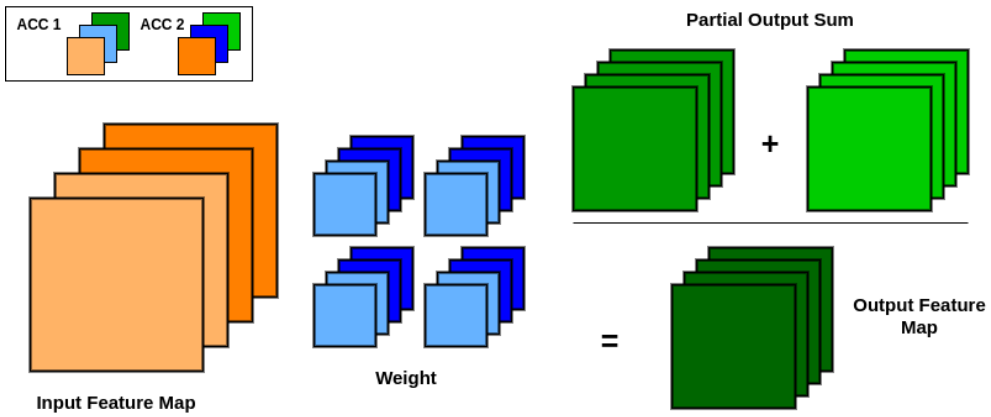

Figure 4.3: Input parallelism

4.2 Memory organization

In ESP the memory accesses are handled by the MT, the number of these systems can vary from a minimum of 1 up to 4, this has the direct effect of increasing the memory bandwidth since each MT is connected to a separate DDR controller. It is in our interest to maintain the functionality of the execution of our CNNs varying the numbers of memory tiles to better understand the impact of the memory bandwidth on the end-to-end inference performance. Depending on the number of MTs and CONV2D accelerators utilized, the weights and the IFMAPS are stored in different ways inside the addressable space. The top section of figure 4.4 shows the case where we have one single MT and one CONV2D accelerator. In this case, the weights of each layer are stored sequentially in the addressable memory. In the case of MT=2 and CONV2D=2, instead, we split the weight of each layer on two separate MT that will be accessed separately by two accelerators. In the last case we have MT=4 and CONV2D=4.

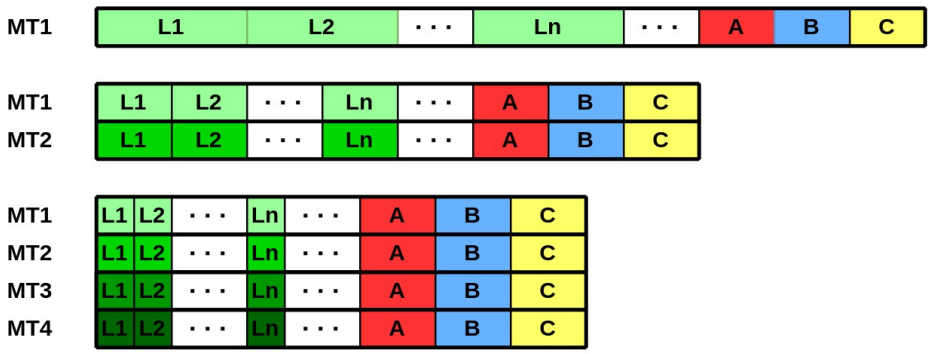

Figure 4.4: Memory organization

The intermediate feature maps are stored at specific memory addresses depending on the topology of the network being executed. In modern CNNs, it is common for the output of a layer to be forwarded to two separate layers, which then converge through an element-wise addition. To support the execution of this kind of network, in the addressable space of each memory tile three different offsets allow us to store the intermediate IFMAPS, we refer to them as A, B, C. When using more than one memory tile the memory organization is replicated over the addressable space handled by each MT, for both the weights and the IFMAPS. The offset between memory location A in the first MT and A in the second is indeed equal to the addressable space of a single MT. In figure 4.5 we can see a segment of ResNet18 where we parallelize the CONV layers on two CONV2D accelerators and the element-wise additions on two EWAs. In cases like this, using three separate offsets for intermediate features management guarantees that the same outputs can be temporally reused without being overwritten by intermediate operations. To clarify this, we highlight the 2 memory offsets to respectively load the input features and store the output feature for each operation in this example.

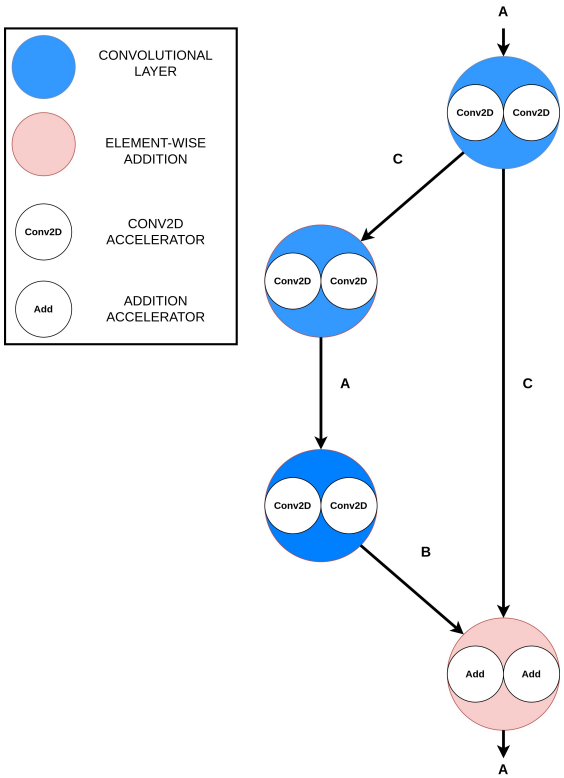

Figure 4.5: Partial IFMAPS allocation in the addressable space in ResNet18

4.2.1 Optimization of the partial sum addition

in order to optimize the memory bandwidth usage, the EWAs' accesses are equally balanced across multiple memory tiles. This crucial operation is necessary for two different operations:

- Reconvergent paths
- Partial Output Additions at the output of each layer executed by parallelizing across multiple accelerators with INPP strategy.

In the first case, we limit the number of EWAs to the maximum number of MTs in the SoC fabric. This happens because the element-wise addition in the reconvegent paths has the input operators stored in the addressable space handled by the same MT. We can better understand this concept by referring to figure 4.5 when our SoC fabric has 2 CONV2D accelerators, 2 MTs, and 2 EWAs. The EWAs that perform the addition have their inputs stored at the offsets B and C, in both the MTs in use, the first and the second EWA will leverage, to access the memory, respectively the first and the second MT independently. In the case of a partial output sum reduction, the maximum number of EWAs used is half of the MT number, as we aim to have a single memory transaction per MT by the EWAs, and the inputs of the addition are stored in the addressable space handled by two MTs. To provide a clearer understanding of how this last operation is performed, we show the memory accesses and the corresponding computations in Figure 4.6. Finally, in general, the workload of the element-wise addition in the reconvergent path is small compared to the partial sum reduction following the convolution when using INPP.

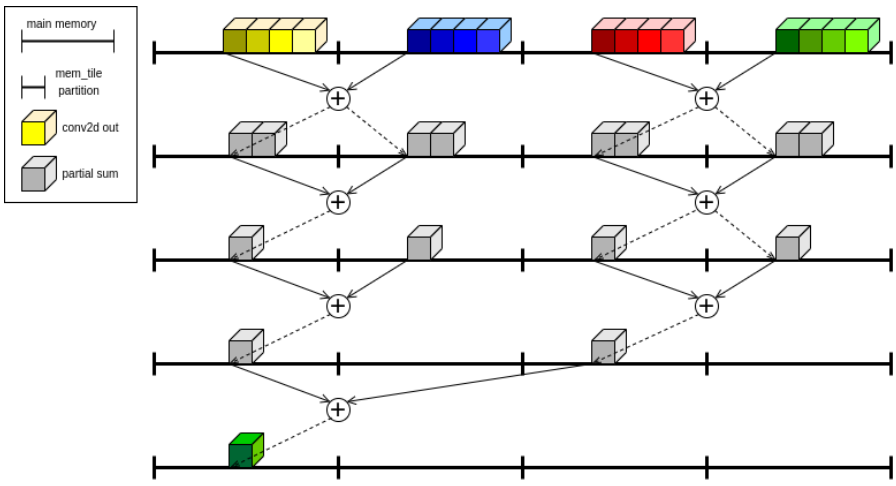

Figure 4.6: EWAs memory accesses

4.3 Software application

The application used to deploy the input model on a given SoC fabric implemented on FPGA is written in C and it leverages the software stack developed for ESP. First of all the inputs of the application are:

1. **Input Model**
2. **SoC Resources in Use (A Subset of the Resources Available in the SoCs fabric deployed on FPGA)**
   (a) Number of Conv2D_Acc
   (b) Number of Adders
   (c) Number of Memory Tiles
3. **Start and Stop Layer**
   (a) Start Layer
   (b) End Layer

Once we have set the number of accelerators we are using we calculate how many accelerators will access each MT and we proceed to the allocation of the model with the parallelization strategies among the two introduced in section 4.2. Then we proceed to sequentially configure all the accelerators involved in the execution of the current layer. Once we have configured the accelerators we leverage the function `esp_run` from the ESP API to invoke the accelerators.

Algorithm 1 Linux ResNet18 Application - Part 1

```
1: Read input parameters
2: Declare: accPerMemTiles
3: if numAccelerators ≥ numMemTiles then
4:     accPerMemTiles = numAccelerators/numMemTiles
5: else
6:     accPerMemTiles = 1
7:     numMemTiles = numAccelerators
8: end if
9: Init configuration Parameters of the Network ▷ numChannels, numFilters, etc.
10: Init MemOffset  ▷ The value of the offset is the addressable space of a single
    memory tile
11: Calculate A, B, C Offsets
12: Calculate INPP partial sum reduction Offsets
13: Allocate space for model Parameters
14: for i = start_i to end_i − 1 do
15:     if Parallelism[i] = INPP then
16:         Fill the accelerators-reserved buffer region with INPP policy
17:     else if Parallelism[i] = OUTP then
18:         Fill the accelerators-reserved buffer region with OUTP policy
19:     end if
20: end for
```

Algorithm 2 Linux Application ResNet18 - Part 2

```
for i = start_i to end_i − 1 do
    Common Parameters Calculation
    if Parallelism[i] = INPP then
        Write Accelerators Configuration registers(INPP Params)
    else if Parallelism[i] = OUTP then
        Write Accelerators Configuration registers(OUTP Params)
    end if
    if i = start_i then
        Allocate Inputs
    end if
    esp_run(Accelerators)
    if Parallelism[i] = INPP then
        Partial Sum reduction Configuration
        esp_run(Adders)
    end if
    if Reconvergent path then
        Element Wise Addition Configuration
        esp_run(Adders)
    end if
end for
Output checking
if Output OK then
    Test Passed
else
    Test Failed
end if
```

4.4 Evaluation of the performance

To better understand the trade-offs involved in complex SoCs with multiple Conv2D accelerators, EWA units, and MTs, we analyze the performance of relevant computational kernels using the roofline model. These kernels are selected from the neural networks we aim to deploy on the ESP. In particular, we want to evaluate the performance trends as we vary key parameters such as the number of Conv2D accelerators, the PLM dimensions, the MAC parallelism, and the number of MTs.

The AI of the computational kernels varies depending on the chosen parallelization scheme, the number of accelerators utilized in the computation, and the size of the weight PLMs. Given the weight-stationary nature of our architecture, we

observe that when the weight PLM is smaller than the size of the weights, the IFMAPS must be accessed multiple times. This leads to an increase in the total number of memory accesses, impacting overall performance. Figure 4.7 shows the evolution of the performance of Layer 2 of ResNet34 when executed with a variable number of CONV2D accelerators and with different parallelisms.

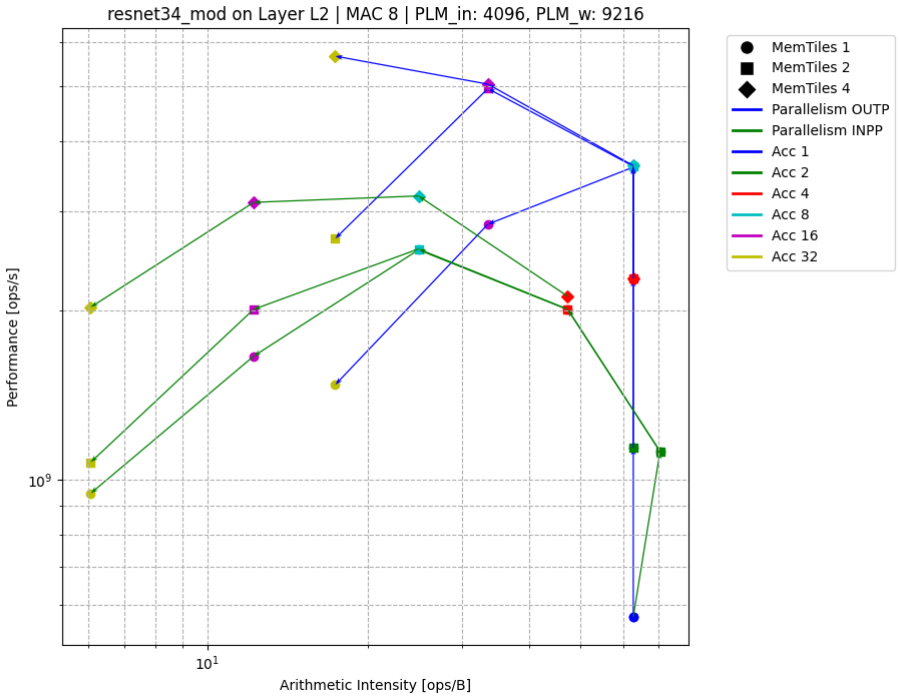

Figure 4.7: Evolution of the throughput with the variation of Conv2D accelerators and parallelism on the Performance vs AI plane

We can see in figure 4.7 that, in general, the arithmetic intensity tends to reduce with the increase in the number of accelerators. This is because the total number of memory transactions is increasing, while the number of operations of a single CONV layer remains the same. When we are far from the memory bandwidth limit imposed by the MTs instantiated in the SoC, the throughput increases with an increase in the number of CONV2D accelerators. This is expected behavior since we are splitting the total number of operations onto multiple compute elements, although we see diminishing returns in increasing the parallelization. For instance, the performance increase from 1 to 2 accelerators is greater than the increase from 4 to 8. When we reach the memory-bound limit, we can see that we are degrading the throughput by increasing the number of accelerators. In this case, we are

increasing the area and worsening the performance, which is indeed the worst-case scenario.

The following formulas are used to compute the arithmetic intensity based on memory transactions and the number of operations.

Outputs Dimension:

$$O_{dim} = O_H \times O_W \times N_{filt} \tag{4.1}$$

Weights Dimension:

$$W_{dim} = K_H \times K_W \times N_{ch} \tag{4.2}$$

Inputs Dimension:

$$IN_{dim} = I_H \times I_W \times N_{ch} \tag{4.3}$$

Memory Transactions:
First of all, we calculate the weight dimension per accelerator

$$W_{PA} = \frac{W_{dim}}{N_{CONV2D}} \tag{4.4}$$

For different levels of parallelism, the total memory transactions vary, we differentiate the calculation for OUTP and INPP. In the following computation, we refer to the data type as dt, expressed in bytes.

- **OUTP**: The input feature map per accelerator is equal to the INFMAPS

$$IN_{PA} = IN_{dim} \tag{4.5}$$

The total memory transaction in the case OUTP follow equation 4.6.

$$M_{tr} = \left(W_{PA} + \left\lceil \frac{W_{PA} \times \mathrm{dt}}{plm_w} \right\rceil \times IN_{PA} + \frac{O_{dim}}{N_{CONV2D}} \right) \times \mathrm{dt} \times N_{CONV2D} \tag{4.6}$$

- **INPP**:
For Input Parallelism the input features maps are split over the accelerators

$$IN_{PA} = \frac{IN_{dim}}{N_{CONV2D}} \tag{4.7}$$

and the total number of memory transaction is calculated as:

$$M_{tr\_conv} = \left(W_{PA} + \left\lceil \frac{W_{PA} \times dt}{plm_w} \right\rceil \times IN_{dim} + O_{dim} \right) \times dt \times N_{CONV2D} \tag{4.8}$$

In addition, for input parallelism calculation we have to account for the memory transactions relative to the partial sums in output of the CONV accelerators:

$$M_{tr\_add} = \sum_{i=0}^{\log_2(N_{CONV2D})-1} \frac{N_{CONV2D}}{2^i} \times O_{dim} \times N_{filt} \times dt \tag{4.9}$$

The total memory transactions for Input Parallelism are:

$$M_{tr} = M_{tr\_conv} + M_{tr\_add} \tag{4.10}$$

For calculating the Arithmetic Intensity we need to calculate the total operations of the CONV layer:

$$TotOps_{conv} = 2 \times C_o \times H_o \times W_o \times K_H \times K_W \times C_{in} \tag{4.11}$$

At this point, we can calculate the Arithmetic Intensity of a given layer deployed on an SoC configuration having considered system-level parallelism, CONV parallelization, and micro-architectural properties:

Arithmetic Intensity

$$AI = \frac{TotOps_{conv}}{M_{tr}} \tag{4.12}$$

4.4.1 Experimental Results - Variable Parallelization Strategies

The modern CNN's structures have a very high imbalance in feature maps and weight dimensions. The usual trend that can be observed is that the inputs have usually a higher memory footprint in the early layers while the weights footprint is bigger in the late layers. What we can observe is that usually OUTP performs better on early layers, while INPP has better results on final layers. This behavior introduces the need of a mixed parallelism approach. In order to be able to use different parallelism on subsequent layers we used a common load and store policy to be able to choose the best deployment. Figure 4.8 shows the performances obtained by executing each layer of Resnet18 on FPGA with different CONV parallelization strategies while keeping a constant number of CONV2D accelerators equal to 8.

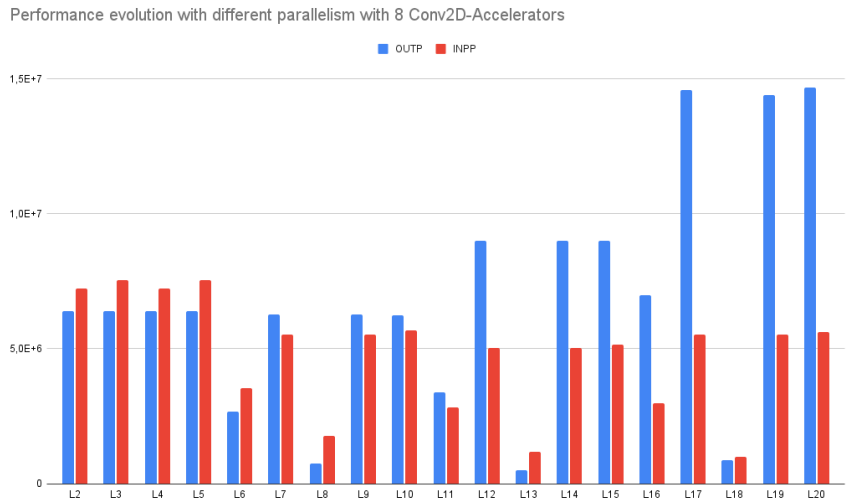

Figure 4.8: Evolution of the throughput with the variation of parallelism ResNet18

Figure 4.9 shows the total latency of ResNet18 in different deployment conditions: when executed utilizing no parallelization strategy, in a single accelerator configuration, the whole network leveraging OUTP and INPP and the mixed parallelization

strategy approach. These results are provided for 4, 8, and 16 accelerators.

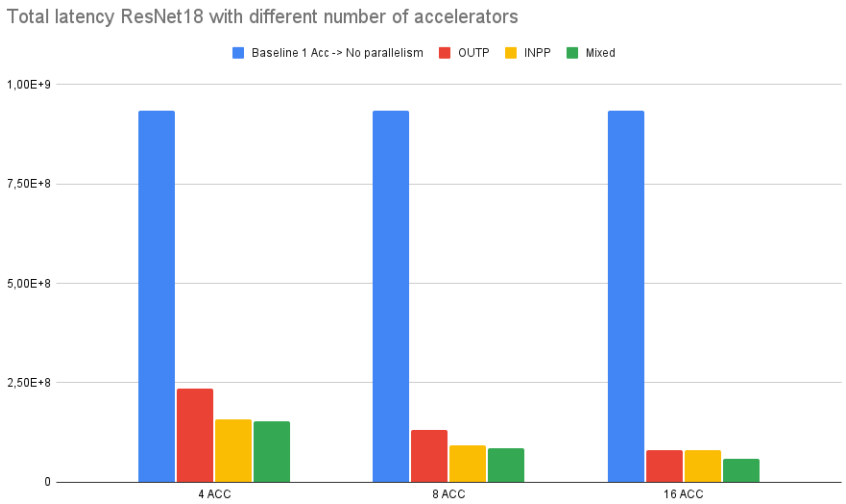

Figure 4.9: Evolution of the throughput with the variation of the number of CONV2D Accelerators ResNet18

4.4.2 Experimental Results - Variable Memory Tiles

The variability of memory tiles changes the available memory bandwidth since every memory tile is connected to a dedicated DDR controller. As mentioned in section 2.1.3 the memory bandwidth increase improves the performance especially when we are dealing with low arithmetic intensity kernels. In figure 4.10 we show a plot of the roofline model built with data collected on FPGA. In particular, each point is a layer of ResNet34 deployed on different SoCs fabrics, where the number of CONV2D accelerators is fixed to 8 while we vary the MT utilization. Each layer has an overall arithmetic intensity that follows the equations in section 4.4. We can clearly see that the points with optimal performance within each arithmetic intensity interval reflect the properties of the roofline model introduced in section 2.1.3. In particular, for low values of arithmetic intensity, we are in the memory-bounded region. As explained in section 2.1.3, when the execution is memory-bounded, the performance benefits from the increase of the memory bandwidth, in our case this corresponds to the deployment of more MTs. When the execution is compute-bound, the performance no longer depends on the memory bandwidth but only on the accelerator parallelism.

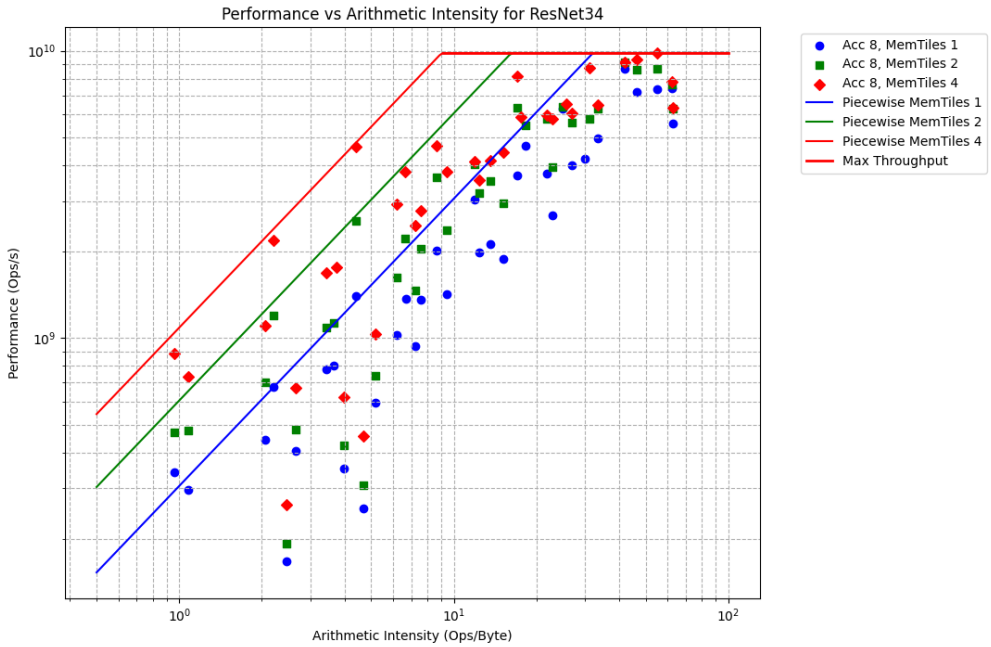

Figure 4.10: Roofline Model for ResNet34

Chapter 5

Experimental setup and dataset creation

When dealing with modern SoCs that contain multiple complex tiles interconnected by a sophisticated NoC, simulating their behavior becomes very impractical since it requires a very long execution time and a high-performance hardware setup. An alternative approach for prototyping and testing is deployment on FPGA. This is the most suitable approach for the ESP since it has an automated flow that allows the user to generate bitstreams, deploy them on FPGA, and test real applications in bare-metal and on top of Linux.

5.1 Experimental setup

For testing and evaluating the flexible infrastructure presented in section 4 we synthesize 18 different bitstreams targeting two different boards, the profpga-xcvu440 and the profpga-xcvu19p. Each bitstream has a self-explaining name that describes the features of the CONV2D accelerators in the SoC. The first field is the number of MAC units, of each accelerator instantiated in the system, concatenated with a code indicating the PLM configuration. Specifically, SI/BI stands for Small/Big input/output PLM, while SW/BW stands for Small/Big Weights PLM. Finally, 16FP/32FP indicated the datatype utilized in the accelerator design, namely 16 fixed-point and 32-fixed point. Each of the 18 bitstreams overviews is shown in table 5.1 highlighting the number of CONV2D Acceleratotors and the dimension of their PLMs, the number of EWAs, and the target board. In order to test the correctness of the applications that we run, we use a Linux application, to verify each configuration. Leveraging Linux system calls is extremely useful during debugging applications since we can handle files and folders, as well as using `secure copy` between the FPGA and a remote machine. On the other hand, evaluating the

Bitstream Name	N CONV2D	PLM I	PLM W	PLM O	N EWA	Target Board
8-SI-SW-32FP	16	4096	9216	4096	4	profpga-xcvu440
8-BI-SW-32FP	16	16384	9216	16384	2	profpga-xcvu440
8-SI-BW-32FP	16	4096	32768	4096	4	profpga-xcvu440
16-SI-SW-32FP	16	4096	9216	4096	4	profpga-xcvu440
16-BI-SW-32FP	16	16384	9216	16384	2	profpga-xcvu440
16-SI-BW-32FP	16	4096	32768	4096	4	profpga-xcvu440
32-SI-SW-32FP	16	4096	9216	4096	1	profpga-xcvu440
32-BI-SW-32FP	16	16384	9216	16384	1	profpga-xcvu440
32-SI-BW-32FP	16	4096	32768	4096	4	profpga-xcvu440
8-SI-SW-16FP	32	4096	9216	4096	4	profpga-xcvu19p
8-BI-SW-16FP	32	8192	9216	8192	2	profpga-xcvu19p
8-SI-BW-16FP	32	2048	18432	2048	4	profpga-xcvu19p
16-SI-SW-16FP	32	4096	9216	4096	4	profpga-xcvu19p
16-BI-SW-16FP	32	8192	9216	8192	1	profpga-xcvu19p
16-SI-BW-16FP	32	2048	18432	2048	2	profpga-xcvu19p
32-SI-SW-16FP	32	4096	9216	4096	4	profpga-xcvu19p
32-BI-SW-16FP	32	8192	9216	8192	1	profpga-xcvu19p
32-SI-BW-16FP	32	2048	18432	2048	2	profpga-xcvu19p

Table 5.1: Bitstreams under test

real performance of our hardware infrastructure on top of Linux can be challenging. When running the operating system, numerous processes are executed concurrently, which may result in the process invoking the accelerator not being scheduled immediately or assigned a high priority. Given the substantial size of the Linux operating system and the relatively limited efficiency of the Ariane core, the overall execution speed is not optimal. Furthermore, invoking the accelerator triggers a context switch from user space to kernel space, which introduces significant overhead. Additionally, when multiple accelerators are invoked using the `esp_run` function, the `pthread` library is employed for thread management, further increasing overhead due to its inherent complexity. For these reason, we decided to use a bare-metal application to evaluate the end-to-end execution performances, while keeping the Linux-based deployment flow only for functional validation. The main idea is to generate some binary files with the configurations of the various accelerators and the numbers of invocations. The bare-metal application will read these binary files and execute the network. In order to profile the execution time, we leverage `esp monitors` that are registers dedicated to counting the number of clock cycles that the program will take to execute. At the end of the execution of the application, the result of the profiling is written using a simple `printf` function call.

5.2 Batch evaluation

Our goal is to evaluate the performance of our system by using all the available knobs introduced so far. Each of the 18 bitstreams under test is configured with a fixed number of tiles. Our goal is to explore all possible combinations of execution mappings, which results in a very high number of combinations to be profiled.

To manage this, we developed a testing infrastructure, the structure of which is summarized in the directory tree below:

```
TEST
|-- README
|-- bitstreams.txt
|-- batch_eval.sh
|-- scripts
|   |-- bs_eval.sh
|   `-- utils.sh
|-- devcom
|   |-- check.sh
|   |   |-- parser.py
|   |-- csv
|   |   |-- resnet18.csv
|   |   |-- resnet34.csv
|   |   |-- resnet50.csv
|   |   |-- squeezenet10.csv
|   |   |-- squeezenet11.csv
|   |-- layer_struct
|   |   |-- resnet18.csv
|   |   |-- resnet34.csv
|   |   |-- resnet50.csv
|   |   |-- squeezenet10.csv
|   |   `-- squeezenet11.csv
|   |-- log
|   |   |-- resnet18.log
|   |   |-- resnet34.log
|   |   |-- resnet50.log
|   |   |-- squeezenet10.log
|   |   `-- squeezenet11.log
|   `-- validation
|       |-- resnet18_validation.log
|       |-- resnet34_validation.log
|       |-- resnet50_validation.log
```

```
|       |-- squeezenet10_validation.log
|       `-- squeezenet11_validation.log
`-- transcript
```

- **README**: Contains instructions and an overview of the testing infrastructure, guiding users on how to set up and execute the experiments.
- **bitstreams.txt**: A text file listing all the bitstreams to be tested. Each line specifies the bitstream name, number of CONV2D accelerators, and EWAs
- **batch_eval.sh**: The main script that orchestrates the batch evaluation of different bitstreams and configurations. It automates the entire testing process by sequentially programming the FPGA, compiling applications, and managing the execution flow.
- **scripts/**
  - **bs_eval.sh**: A script used to evaluate individual bitstream configurations. It handles the execution of all possible combinations of accelerator deployments for a given bitstream.
  - **utils.sh**: A script containing utility functions and common routines used in the testing infrastructure. This includes preprocessing steps like setting up the FPGA environment and compiling applications.
- **devcom/**: Directory containing information related to the communication between the local machine and the device.
  - **log/**: Directory containing log files from the execution of tests. These log files capture the output from the FPGA during the execution of the tests for each network.
  - **parser.py**: A Python script used to parse log files and extract relevant information from the test outputs.
  - **layer_struct/**: Directory containing CSV files that describe the layer structures of the neural network models. These files are used by the parser in assigning each layer latency to a given layer structure.
  - **validation/**: Directory containing validation files. When the parser extracts the information from the log it generates these files that keep track of the inconsistencies between the expected result of the test and the one present in the log.
  - **check.sh**: A script that checks for outliers in the latency data. If a measurement is below an execution time threshold it raises an error.

  - **csv/**: Directory containing CSV files with parsed results from the tests. These files contain the results for each neural network model tested.

- **transcript**: A file containing a complete transcript of the execution process, including messages and outputs from the scripts and the FPGA. It serves as a comprehensive log for debugging and verification purposes.

Each profiling task consists of executing a full CNN inference on a given SoC configuration that utilizes a portion or all of the resources available on the bitstream. Executing all the tests sequentially by waiting for the maximum execution time before launching the next test is impractical due to the significant variation in latency across different workloads and deployments.

To efficiently execute the batch of experiments, we built an infrastructure that synchronizes the various machines involved in the process. The infrastructure comprises two primary machines:

- **Machine A**: This is the machine where the ESP repository lives, together with all the bitstreams, software applications, and the testing infrastructure. It is responsible for programming the FPGA, compiling the bare-metal application, generating the binary files necessary for execution, and loading these onto the FPGA. It also initiates the execution.

- **Machine B**: This machine is physically connected to the FPGA through the UART, and runs a script that monitors the UART output from the FPGA.

We implemented a synchronization mechanism between Machine A and Machine B. After Machine A launches the execution, the testing script pauses, waiting for a token indicating that the inference has been completed. Machine B continuously monitors the log file generated by `minicom`, which captures the messages printed by the application over UART. Once Machine B detects a specific keyword signaling completion, it sends a token back to Machine A, allowing it to resume and launch the next inference. Since the machines are on different servers, we use an empty file transferred via the secure copy command (`scp`) as the synchronization token. This procedure is repeated for each inference. After covering all possible configurations for a bitstream, we reprogram the FPGA with a new bitstream and repeat the process for all the networks under test. Figure 5.1 shows the interaction between Machine A, B and the FPGA.

Several issues can prevent the test execution from completing successfully, such as:

- **Programming Faults**: Errors during the programming of the FPGA can halt the testing process.

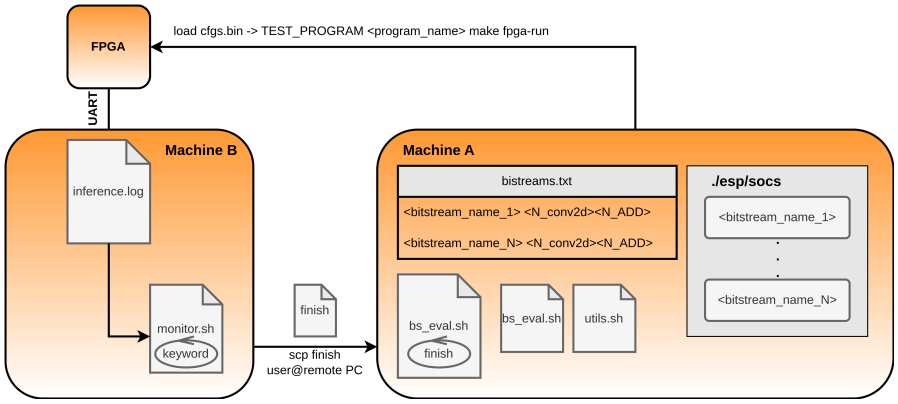

Figure 5.1: Diagram of the automated testing infrastructure

- **Load Errors**: Error during the load of the binaries file leveraging `esp_link` that is the communication protocol that enables the communication between and external machine and ESP.
- **Execution Failures**: The application may fail to complete execution due to the stress of consecutive FPGA programming cycles.

To mitigate these issues, we implement timeout mechanisms for the programming step and the execution phase. If a timeout occurs or an error is detected, the board is reprogrammed, and the test resumes from the last successful point.

Algorithms 3, 4 and 5 summarize the steps explained in this section.

5.2.1 Post processing of data

In order to prepare the dataset for the learning models that we will discuss in section 6.2.2, we need to post-process the collected data into an appropriate format. When we profile the execution of a CONV layer that is followed by an element-wise addition, the total latency result in terms of clock cycle is the sum of the two contributions. However, to provide a significant figure estimating the performances of the convolution kernel itself, we depurate the total latency of the contribution of the adder. Another important task in the post-processing is related to the addition of the total memory transaction and the arithmetic intensity, using formulas of section 4.4.

5.2.2 Dataset

The networks considered in this work are ResNet18/34/50 and SqueezeNet1.0/1.1, deployed on the 18 bitstreams of table 5.1 testing all the possible combinations of

Algorithm 3 CNN Batch evaluation

Input: Bitstream under test in the file 'bitstreams.txt'
Output: Execution of FPGA configurations with logging
Clean the design folder from the previous execution Initialize log_file Initialize $number\_of\_layers$ Initialize $home\_dir$
for each $bitstream$ in $bitstreams\_file$ **do**
  Extract $bitstream$, n_conv2d_acc, n_adder_acc from the $line$
  Extract mac, plm_in, plm_w, d_type from $bitstream$
  Log the current bitstream and MAC, PLM, and data type details
  Change to the directory '$bitstream$'
  Source the environment setup script
  if $bitstream = last\_bitstream$ **then**
    Set $last\_bitstream\_flag \leftarrow 1$
  else
    Set $last\_bitstream\_flag \leftarrow 0$
  end if
  Source and execute the preprocessing script with parameters:
    mac, plm_in, plm_w, d_type, $resnet\_raw$, log_file
  Source and execute evaluation script 'bs_eval.sh' with parameters:
    n_acc, n_add, $number\_of\_layers$, mac, plm_in, plm_w, d_type, $last\_bitstream\_flag$, log_file
  Return to $home\_dir$
end for

Algorithm 4 Bitstream Preprocessing ('utils.sh')

Input: mac, plm_in, plm_w, d_type, $resnet\_raw$
Output: FPGA bitstream and header setup for the evaluation
Set bitstream format string $bitstream \leftarrow$ "mac-plm_in−plm_w-d_type"
Use a `case` statement to assign the correct header based on mac and d_type
Substitute the string in $cnn\_application$ with the header information
Select the target FPGA
  1. Compile $cnn\_application$
  2. Compile $cnn\_config\_application$
  3. Program target FPGA

Algorithm 5 Bitstream Evaluation ('bs_eval.sh')

Input: p_acc, n_adder, $number\_of\_layers$, mac, plm_in, plm_w, d_type, $last\_configuration$, log_file
Output: Log of configuration testing, FPGA inference results
Compute $log2\_acc$ and $log2\_add$ (log base 2 of $conv2d\_acc$ and n_adder)
Initialize $last\_inference \leftarrow 0$
for $par = 0$ to 1 **do** ▷ Parallelism 0: OUTP, Parallelism 1: INPP
    for $i = 0$ to $log2\_acc$ **do**
        Calculate $p\_acc \leftarrow 2^i$
        for $j = 0$ to $log2\_add$ **do**
            Calculate $n\_add \leftarrow 2^j$
            for $mem\_tiles = 1$ to max_mem_tiles **do**
                if last configuration and last bitstream **then**
                    Set $last\_inference \leftarrow 1$
                end if
                Generate configuration for p_acc, n_add, mem_tiles
                Run inference on FPGA with timeout mechanism
                if timeout occurs **then**
                    Reprogram the board and retry inference
                end if
                Wait for inference to complete or timeout
            end for
        end for
    end for
end for
if $last\_bitstream = 1$ **then**
    Wait for the final log file and copy the results
end if

resources utilization described in 4. Leveraging the testing infrastructure described in this chapter we collected up to 170.000 datapoints. Table 5.2 shows the total number of observations for each network.

Network	Number of observations
resnet18	20977
resnet34	38641
resnet50	57409
squeezenet10	27601
squeezenet11	27601

Table 5.2: Dataset dimension

Chapter 6

Neural Network Mapping Tool for SoC Architectures Leveraging ML Algorithms

In this chapter, we discuss the goals and structure of a Mapping Tool for CNNs on tile-based SoCs. Our goal is to optimize the CNN deployment on a given SoC configuration, finding the best mapping for each layer of the network. A *mapping* is a deployment of a CONV layer on a given SoC configuration utilizing a Parallelization Scheme (PS) = {#MT, #`CONV_2D`, #`N_ADD`, `PARALLELISM` = {`OUTP`, `INPP`}} that leverages all or part of the total resources available in the input SoC configuration. The infrastructure introduced in Section 4 is very flexible, and the space of possible deployments is indeed vast. Having such flexibility, while maintaining the correctness of the algorithm, is generally advantageous, as it allows tailoring the deployment of a layer on the SoC with finer granularity. On the other hand, when performance varies significantly across different deployments and depends on memory and computational capabilities, finding the best mapping can be challenging. Memory bandwidth capabilities depend on the number of MTs in use providing off-chip DRAM access, while memory requirements depend on the dimensions of the on-chip PLMs distributed inside the accelerator tiles, the number of CONV2D accelerators in use, and the PS. All of these parameters are encapsulated in the arithmetic intensity of the layer mapped using a given PS. Computational capabilities depend on the number of CONV2D accelerators in use and the MAC parallelism inside each accelerator. This heterogeneous aggregation of micro-architectural and system-level parameters determining end-to-end inference performance introduces multiple trade-offs. To explore the space of possible mappings, we chose to leverage a data-driven approach by training machine learning models on the data collected with the test infrastructure introduced in

Section 5. The goal of the model is to predict the latency of a CNN layer execution based on: the CNN layer structure, the SoC configuration, the parallelization strategy, the arithmetic intensity, and the total memory transactions. The set of possible PS on a given SoC configuration represents our mapping space. Leveraging a Latency Predictor to find the best mapping of a network involves predicting the latency for all possible PS for all the sequential layers of the network and finding the minimum.

6.1 Mapping Tool Overview

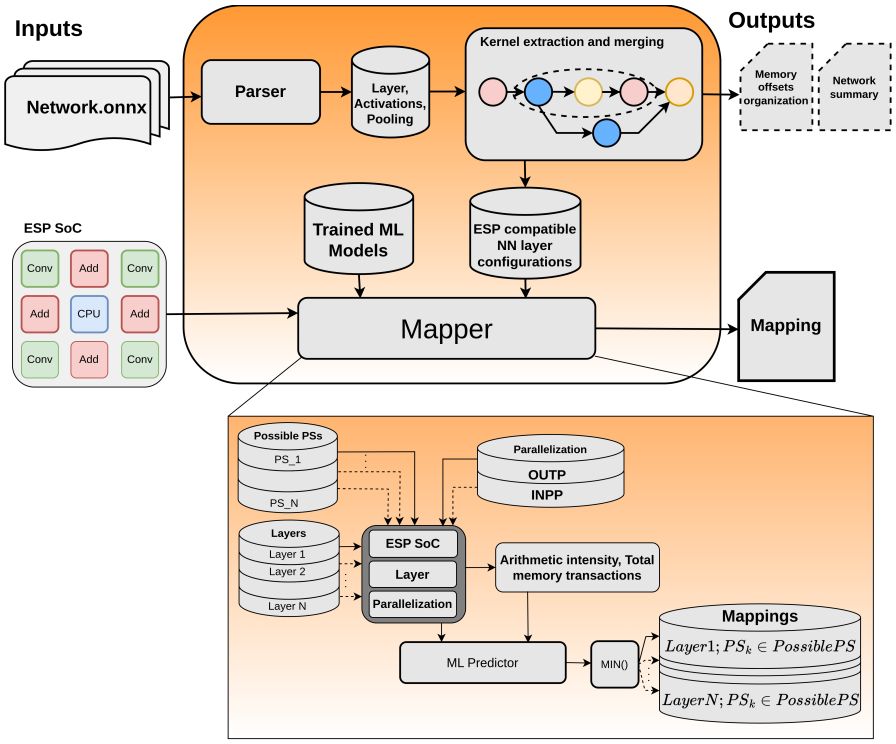

Figure 6.1: Mapping Tool

The inputs of the mapping tool are two. First, the .onnx standard format of a Network we wish to map, contains all the information on the topology, the layers structure, and the operators that are used. The second is an instance of an ESP SoC, that has a given number of Conv2D Accelerators, along with his PLMs dimensions, the number of adders, and the data type used in the calculation (16FP or 32FP).

The first step of the mapping flow is highlighted in the top section of figure 6.1 and consists of parsing the layers in the target network and extracting all the

structural information. Once we collect all the Conv layers, the Additions, the activations, Batch Normalization (BN), and Pooling, we merge all of them in a way that is proper to the underlying hardware. As we discussed for other platforms in section 2.4, CONV accelerator usually performs operations such as ReLu, BN, or pooling within the computation of the CNN layer [24]. The CONV2D accelerator presented in section 3 also performs these operations internally. Once we have built a graph that reflects the network structure as it will be executed on ESP, the tool guides the user in efficiently handling memory offsets organization, in particular in the presence of reconvergent paths. After the parsing step, the tool also provides an informative table of the Network summary in a dedicated `log` file. An example of this graph reorganization is shown in figure 6.2, and an instance of the informative table is displayed in 6.1

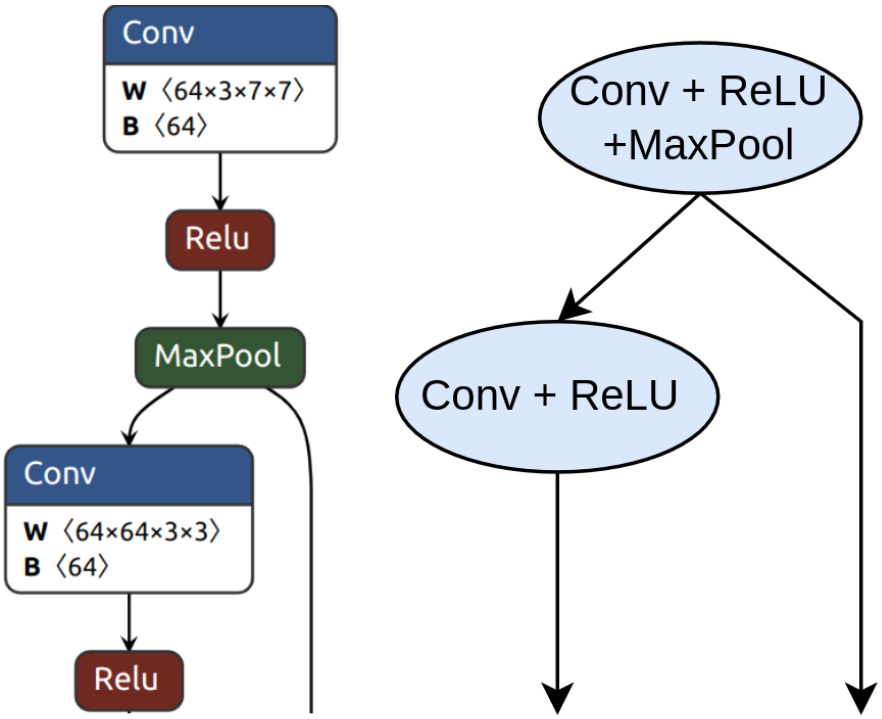

Figure 6.2: Graph integration of Conv, Activation and Pooling from ResNet18

Since ResNets have reconvergent paths composed of at most two CONV layers, the tool provides a memory offset organization in order to store partial input features map without overwriting useful data in main memory. This process reflects the concepts introduced in section 4.2. The output of this step is provided by the tool under the form of a table in the `log` file, an example for ResNet 18 is provided in 6.2.

Layer	Layer Type	Input Shape	Output Shape	Kernel Size	Padding	ReLU	Pooling	Batch Norm
Layer 0	Conv	[3, 224, 224]	[64, 56, 56]	[64, 7, 7]	1	1	1	2
Layer 1	Conv	[64, 56, 56]	[64, 56, 56]	[64, 3, 3]	1	1	0	2
Layer 2	Conv	[64, 56, 56]	[64, 56, 56]	[64, 3, 3]	1	0	0	2
Layer 3	Add	[64, 56, 56]	[64, 56, 56]	-	-	1	0	1
Layer 4	Conv	[64, 56, 56]	[64, 56, 56]	[64, 3, 3]	1	1	0	2
Layer 5	Conv	[64, 56, 56]	[64, 56, 56]	[64, 3, 3]	1	0	0	2
Layer 6	Add	[64, 56, 56]	[64, 56, 56]	-	-	1	0	1
Layer 7	Conv	[64, 56, 56]	[128, 28, 28]	[128, 3, 3]	1	1	0	2
Layer 8	Conv	[128, 28, 28]	[128, 28, 28]	[128, 3, 3]	1	0	0	2
Layer 9	Conv	[64, 56, 56]	[128, 28, 28]	[128, 1, 1]	1	0	0	2
Layer 10	Add	[128, 28, 28]	[128, 28, 28]	-	-	1	0	1
Layer 11	Conv	[128, 28, 28]	[128, 28, 28]	[128, 3, 3]	1	1	0	2
Layer 12	Conv	[128, 28, 28]	[128, 28, 28]	[128, 3, 3]	1	0	0	2
Layer 13	Add	[128, 28, 28]	[128, 28, 28]	-	-	1	0	1
Layer 14	Conv	[128, 28, 28]	[256, 14, 14]	[256, 3, 3]	1	1	0	2
Layer 15	Conv	[256, 14, 14]	[256, 14, 14]	[256, 3, 3]	1	0	0	2
Layer 16	Conv	[128, 28, 28]	[256, 14, 14]	[256, 1, 1]	1	0	0	2
Layer 17	Add	[256, 14, 14]	[256, 14, 14]	-	-	1	0	1
Layer 18	Conv	[256, 14, 14]	[256, 14, 14]	[256, 3, 3]	1	1	0	2
Layer 19	Conv	[256, 14, 14]	[256, 14, 14]	[256, 3, 3]	1	0	0	2
Layer 20	Add	[256, 14, 14]	[256, 14, 14]	-	-	1	0	1
Layer 21	Conv	[256, 14, 14]	[512, 7, 7]	[512, 3, 3]	1	1	0	2
Layer 22	Conv	[512, 7, 7]	[512, 7, 7]	[512, 3, 3]	1	0	0	2
Layer 23	Conv	[256, 14, 14]	[512, 7, 7]	[512, 1, 1]	1	0	0	2
Layer 24	Add	[512, 7, 7]	[512, 7, 7]	-	-	1	0	1
Layer 25	Conv	[512, 7, 7]	[512, 7, 7]	[512, 3, 3]	1	1	0	2
Layer 26	Conv	[512, 7, 7]	[512, 7, 7]	[512, 3, 3]	1	0	0	2
Layer 27	Add	[512, 7, 7]	[512, 7, 7]	-	-	1	0	1

Table 6.1: Network Layer Summary

Layer	Layer 0	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6
Name	Conv0	Conv1	Conv2	Add0	Conv3	Conv4	Add1
In/Out	a→c	c→a	a→b	c, b→a	a→b	b→c	a, c→b

Layer	Layer 7	Layer 8	Layer 9	Layer 10	Layer 11	Layer 12	Layer 13
Name	Conv5	Conv6	Conv7	Add2	Conv8	Conv9	Add3
In/Out	b→a	a→c	b→a	c, a→b	b→a	a→c	b, c→a

Layer	Layer 14	Layer 15	Layer 16	Layer 17	Layer 18	Layer 19	Layer 20
Name	Conv10	Conv11	Conv12	Add4	Conv13	Conv14	Add5
In/Out	a→b	b→c	a→b	c, b→a	a→b	b→c	a, c→b

Layer	Layer 21	Layer 22	Layer 23	Layer 24	Layer 25	Layer 26	Layer 27
Name	Conv15	Conv16	Conv17	Add6	Conv18	Conv19	Add7
In/Out	b→a	a→c	b→a	c, a→b	b→a	a→c	b, c→a

Table 6.2: Layer Memory organization for ResNet18

After extracting the topology of the network and reorganizing the kernels according to the hardware support, we focus on the mapping task itself. We define an SoC Configurations as SOC_CFG=$\{N_{MT}$, N_{ADD}, N_{CONV2D}, DATA_TYPE, N_{MAC}, PLM_IN(OUT), PLM_W$\}$. We remind that a mapping is a deployment of a CONV Layer on the SoC utilizing a PS = {#MT, #CONV2D, #N_ADD, PARALLELISM = {OUTP, INPP}}. It is important to notice that when #CONV2D is equal to one, no parallelization scheme is applied. In addition, #MT, #CONV2D and #N_ADD must always satisfy the following rules:

$$MT_{PS} \leq CONV2D_{PS} \tag{6.1}$$

$$CONV2D_{PS} \leq N_{CONV2D} \tag{6.2}$$

$$CONV2D_{PS} = 2^k \quad \text{with} \quad 0 \leq k \leq \log_2(N_{CONV2D}) \tag{6.3}$$

$$N\_ADD_{PS} \leq N_{ADD} \tag{6.4}$$

$$N\_ADD_{PS} = 2^k \quad \text{with} \quad 0 \leq k \leq \log_2(N_{ADD}) \tag{6.5}$$

The space of the possible PS has the dimensionality of

$$PS_{space} = 2 * \log_2(N_{ADD}) * \log_2(N_{CONV2D}) * \log_2(N_{MT}) \tag{6.6}$$

where we account for the parallelization strategy multiplying by the factor 2 in front. A model capable of predicting the latency of a layer deployed with a given PS can be used to explore the entire space of possible PS and pick the one with the expected minimum latency for each layer. Algorithm 6 summarizes how the latency predictor is used to find the best PS for each network layer.

The output of the mapping tool is a report that provides a list of suggested mappings for a given network and a given SoC configuration. We show an example input SoC configuration in table, and the corresponding mapping tool output for ResNet18 in table 6.3.

Algorithm 6 Find Layer Mapping

```
1: procedure FindLayerMapping(layer, soc)
2:     ▷ Input: layer, soc
3:     ▷ Output: best_soc_cfg, min_latency
4:     ▷ Initialization
5:     min_latency ← ∞
6:     ▷ Execution
7:     for i ← 0 to log2(soc['p_acc']) do
8:         p_acc ← 2^i
9:         for j ← 0 to log2(soc['n_add']) do
10:            n_add ← 2^j
11:            for w ← 0 to log2(soc['mem_tiles']) do
12:                mem_tiles ← min(2^w, p_acc)
13:                for parallelism ← 0 to 1 do
14:                    soc_cfg ← {
15:                        'mem_tiles': mem_tiles,
16:                        'n_add': n_add,
17:                        'p_acc': p_acc,
18:                        'data_type': soc['data_type'],
19:                        'mac': soc['mac'],
20:                        'plm_in': soc['plm_in'],
21:                        'plm_w': soc['plm_w']
22:                    }
23:                    latency ← latency_predictor.predict(layer, soc_cfg, parallelism)
24:                    if latency < min_latency then
25:                        min_latency ← latency
26:                        soc_cfg['parallelism'] ← parallelism
27:                        best_soc_cfg ← soc_cfg
28:                    end if
29:                end for
30:            end for
31:        end for
32:    end for
33:    return best_soc_cfg, min_latency
34: end procedure
```

mem_tiles	n_add	p_acc	data_type	mac	plm_in	plm_w
4	2	32	2	16	4096	9216

mem_tiles	n_add	p_acc	data_type	mac	plm_in	plm_w	parallelism	predicted_latency
4	1	16	2	16	4096	9216	OUTP	3,691,866
4	1	16	2	16	4096	9216	OUTP	3,692,269
4	1	16	2	16	4096	9216	OUTP	3,691,866
4	1	16	2	16	4096	9216	OUTP	3,692,269
4	2	8	2	16	4096	9216	INPP	1,658,939
4	2	32	2	16	4096	9216	OUTP	2,492,627
2	1	2	2	16	4096	9216	OUTP	594,519
4	2	32	2	16	4096	9216	OUTP	2,492,224
4	2	32	2	16	4096	9216	OUTP	2,492,627
4	2	16	2	16	4096	9216	INPP	1,635,571
4	1	16	2	16	4096	9216	INPP	3,386,103
2	1	2	2	16	4096	9216	INPP	842,182
4	1	16	2	16	4096	9216	INPP	3,385,037
4	1	16	2	16	4096	9216	INPP	3,386,103
4	2	32	2	16	4096	9216	INPP	3,185,959
4	2	32	2	16	4096	9216	INPP	3,980,185
4	2	4	2	16	4096	9216	INPP	992,670
4	1	16	2	16	4096	9216	INPP	3,997,516
4	2	32	2	16	4096	9216	INPP	3,980,185

Table 6.3: SoC configuration on top and tool's mapping output for ResNet18

6.2 Latency predictor

In this section, we describe the steps we followed in the creation of a latency predictor model leveraging ML and evaluate the performances after integrating it in the mapping tool introduced in section 6.1

6.2.1 Dataset

Our goal is to train a model that can predict the latency of CONV layers mapped on tile-based SoCs with acceptable error. The model's inputs include the structure of a CNN layer, the SoC configuration, the convolution parallelization, the layer's arithmetic intensity, and the total memory transactions. An example of the information contained in each observation of the dataset is shown in Table 6.4.

CNN Layer Structural Information

n_channels	n_filters	feature_map_height
64	64	56
feature_map_width	filter_dim	stride
56	3	1
do_relu	pool_type	out_height
1	0	56
out_width	weight_dim	infmap_dim
56	36864	200704
tot_ops_conv		
231211008		

SoC Configuration

mac	plm_in	plm_w
8	4096	9216
mem_tiles	data_type	n_con2D
1	2	1
n_ewa		
1		

Additional Parameters

arithmetic_intensity	total_memory_transactions	parallelism
62.72	3686400	OUTP

Latency (Target)

tot_cycle_convs
40473908

Table 6.4: Information in a datapoint grouped by CNN Layer Structure, SoC Configuration, Additional Parameters, and Latency

6.2.2 Learning Models

To implement our latency predictor we now focus on the exploration of well-known models, starting with Random Forest (RF), Extreme Gradient Boosting (XGB), and Multi-Layer Perceptron (MLP). In order to easily explore different possibilities, a versatile API was implemented to enable the user to choose the target model to experiment with, as well as apply pre-training scaling and/or Principal Component Analysis (PCA). In this section, we highlight the results obtained by exploring different model configurations with a particular focus on RF and XGB, which displayed the most promising performances. Algorithm 7 shows the high-level description of the API implemented.

Algorithm 7 Model Training and Evaluation (Part 1)

Input: Training dataset, Testing dataset, Model choice (MLP, Random Forest, XGBoost)
Output: Trained model, Evaluation metrics, Plots
Step 1: Dataset Preparation
Load pre-processed datasets into DataFrames: 'resnet18', 'resnet34', etc.
Step 2: Configuration Setup
Set configurations for `scaler_choice`, `model_choice`, and flags for analysis options.
Step 3: Dataset Partitioning
Define training and testing set choices for each dataset.
Iterate through all possible combinations of datasets using `itertools.product`.
for each valid partition (non-empty training and testing) **do**
    Create new training and testing datasets by concatenating the DataFrames.
    Remove unnecessary columns and split them into features and target variables.
    Step 4: Data Pre-processing
    if StandardScaler is selected **then**
        Scale features using `StandardScaler`
    else if MinMaxScaler is selected **then**
        Scale features using `MinMaxScaler`
    else
        No scaling is applied.
    end if
    Step 5: Dimensionality Reduction
    if PCA is applied **then**
        Apply PCA transformation to datasets
    end if
    Step 6: Model Selection
    if model_choice is 'MLP' **then**
        Build MLP model
    else if model_choice is 'Random Forest' **then**
        Build Random Forest model
    else if model_choice is 'XGBoost' **then**
        Build XGBoost model
    else
        Raise error for invalid model choice
    end if
    Perform model training and evaluation on each partition.
end for
Output results to log file and display on the console.

Algorithm 7 Model Training and Evaluation (Part 2)

```
if model is 'MLP' then
    Compile and train the model for 50 epochs with batch size 32
    Save model
else if model is 'Random Forest' then
    Fit the Random Forest model and extract feature importances
    Save model
else if model is 'XGBoost' then
    Fit the XGBoost model
    Save model
end if
Log training results and generate training and validation plots.
```

Random forest

Random forest is an ML method that combines multiple decision tree to create a robust learning model. The dataset is randomly sampled, during a process called *bagging*, and the samples are used to train a set of decision trees. By creating a random subspace, each tree can learn different patterns within the same data, leading to more stable and accurate predictions. As usually done with regression problems, each tree provides a decision and the final prediction is obtained with a weighted average of such decisions. Some important hyperparameters in the model configurations are:

- `max_depth`: this parameter sets the max depth of the individual tree. Setting this value to a small number creates shallow trees that are able to capture less complex patterns in the data but also prevent overfitting. It is a good practice to set a `max_depth` if the dataset contains irrelevant features or if it is noisy.
- `max_features`: this is the specify the number of features that are considered in the splitting of a node into each decision tree.
- `n_estimator`: this is the number of total trees that will be included in the Forest ensemble.
- `min_sample_split`: this parameter sets the required sample to split a node in a given decision tree. By default set to 2.
- `min_sample_leaf`: this is the minimum number of samples required to be present in a leaf node of a decision tree. By default set to 1.[30]

In order to evaluate the results of a given model in predicting the execution latency of a Network on a given SoC, we treated each network's dataset as independent and explored all the possible combinations of networks to build the

training set, while evaluating the R-squared score for the target network. This approach results in a comprehensive overview of the performance of the models in real conditions. In fact, each network under test is completely excluded from the training and validation set. We use R-squared as a metric for the evaluation of the model. This metric evaluates how far the scatter data points are from the fitted regression line and is expressed between 0% and 100%[31]. The left scatter in figure 6.3 has a low R-squared score, while the right has a high one.

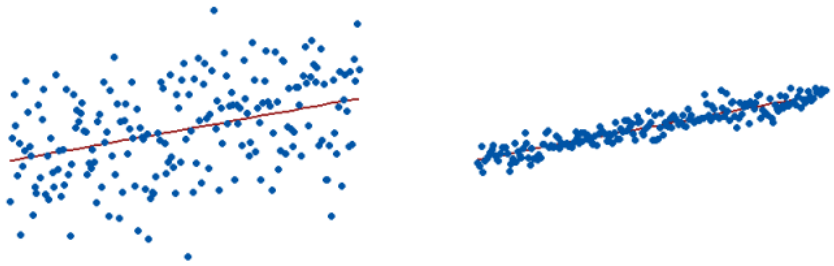

Figure 6.3: Representation of a low and high R-squared score, respectively left and right. [31]

Listing 6.1 shows the configuration of the `RandomForestRegressor` used in this first class of experiments.

Listing 6.1: Random forest model

```
# Configuration parameters
scaler_choice = ''
model_choice = 'random_forest'
apply_pca = False

# . . .

# Model Cfg
def build_random_forest():
    model = RandomForestRegressor(
        max_depth=22,
        n_estimators=25,
        min_samples_leaf=1,
        min_samples_split=2,
        max_features=6,
        oob_score=True,
        random_state=10,
        verbose=3
    )
    return model
```

The heatmap in figure 6.4 highlights the results of this batch of training on multiple datasets for three different networks: ResNet18, Squeezenet10, and ResNet50. The first two networks obtain overall a good score, this is because the range of the latency of execution of the target network is within one of the networks used in the training set. Tree-based models tend to struggle with predictions outside their training range, which limits our ability to map networks that may exhibit higher latencies than those used for training. In real condition of use of the mapping tool, the natural choice of the model's training dataset consists in using the data from all the available networks to build the training set. In figure 6.4, we highlighted in blue the set of training networks that provide the overall best result and in yellow the results of the set that contains all the available networks. In ResNet18 we excluded from the possible choice all the dataset that include ResNet34 in the training set. This is because the two networks share all the layers and so in this condition, the input of the testing network will be equal to the training one and the model will achieve results that are too optimistic. Notice that this could happen in real conditions, in fact, ResNet18 and ResNet34 are formally two separate networks. Nevertheless, we preferred to exclude it from the evaluation for a fair evaluation.

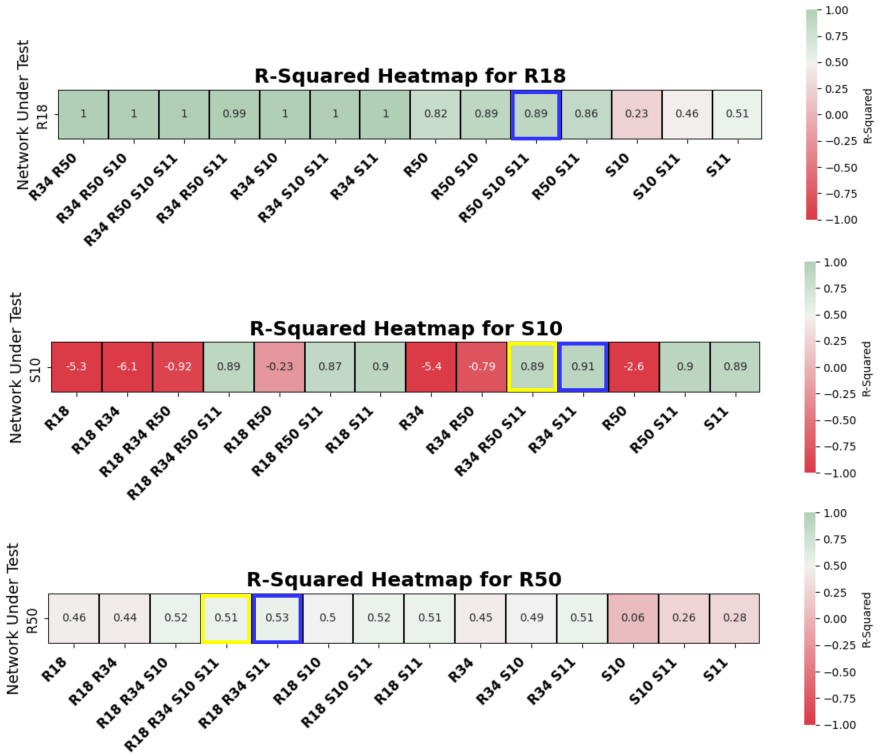

Figure 6.4: Heatmaps of ResNet18, Squeezenet1.0 and ResNet50 with different training sets leveraging Random Forest Regressor

Extreme Gradient Boosting

In this section, we discuss the results obtained from training and testing an *xgboost* model with the same combinations used in the previous section.

Listing 6.2: XGBoost model

```
# Configuration parameters
scaler_choice = ''
model_choice = 'xgboost'
apply_pca = False

# . . .

def build_xgboost():
    model = XGBRegressor(objective='reg:squarederror',
                         n_estimators=600,
                         learning_rate=0.008,
                         max_depth=11,
                         random_state=42)
    return model
```

As done for Random Forest, we evaluate which are the best training sets when we test the model, as shown in figure 6.5. The results of the two models are in general comparable, with XGBoost almost always scoring higher results.

In real conditions of use of the mapper is impossible to know in advance which combination of available networks, used in the training dataset, is the most suited to train a model that maps a given network. The straightforward choice that doesn't need any prior knowledge is to include all the available networks in the training set. In figure 6.6 and 6.7 we evaluate the difference between the training dataset that includes all the available networks and the best-performing training dataset. The results of the two are in all cases very similar, concluding that choosing all the available networks is in general a reasonable choice. An additional feature that the mapper could implement is a metric that enables the evaluation of which subset of available networks in the training set is most likely to generate a well-performing model. Given the correlation between model size and latency, this metric could leverage statistical tools to predict whether the execution time of the target network will fall within the range of the networks used in the training set. Some preliminary explorations have been made in this direction, but it remains an open problem. To evaluate the best achievable performance of the tool, we used the best-performing XGBoost regressor as the latency predictor for each mapped network, noting that the performance difference between this model and the one trained on all available networks is small and should not significantly degrade the mapper's overall performance.

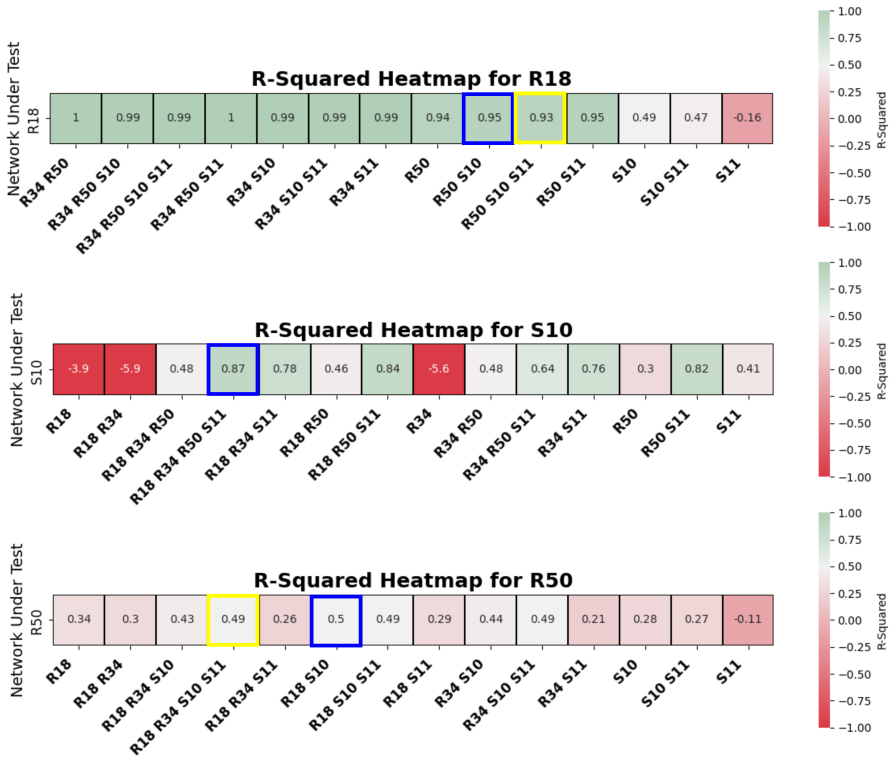

Figure 6.5: Heatmaps of ResNet18, Squeezenet1.0 and ResNet50 with different training sets leveraging XGBoost regressor

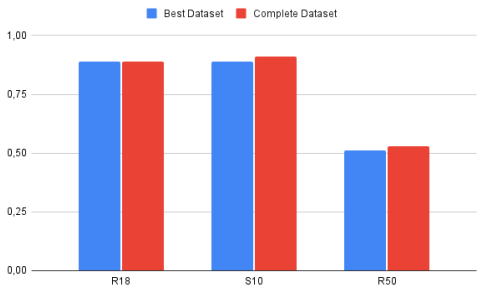

Figure 6.6: R-sqared random forest: best Dataset vs Complete dataset

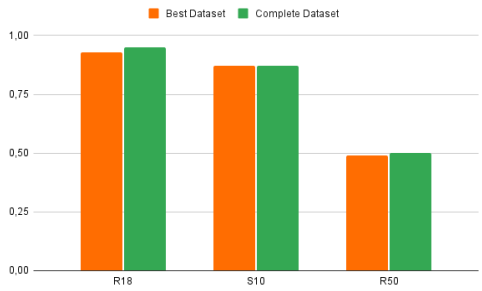

Figure 6.7: R-sqared xgboost: best Dataset vs Complete dataset

Multi-layer perceptron

Several experiments were conducted by building various MLP models and training them on different combinations of input networks. To reduce the dimensionality of the dataset, we applied PCA in combination with a standard scaler. Other

approaches explored alternative scaling techniques, such as MinMax scaling. However, none of these methods achieved results comparable to those obtained using ensemble techniques, which proved to be highly effective for this type of regression problem.

6.3 Mapper performance

We now aim to assess the performance of the mapping tool in addressing the problem of finding the best PS for a given network deployed on a given SOC_CFG. For each CONV layer, we identify the PS that has the predicted minimum latency, referring to this set of PS as the `Predicted Best Mapping` (PBM). The predicted latency of the PS for each layer is derived from the predictor's model inference output, and differs from the actual latency measured on FPGA during the data-set collection. Therefore, we evaluate the Percentage Error (PE) and the Absolute Error (AE) relative to the same PBM measured on FPGA.

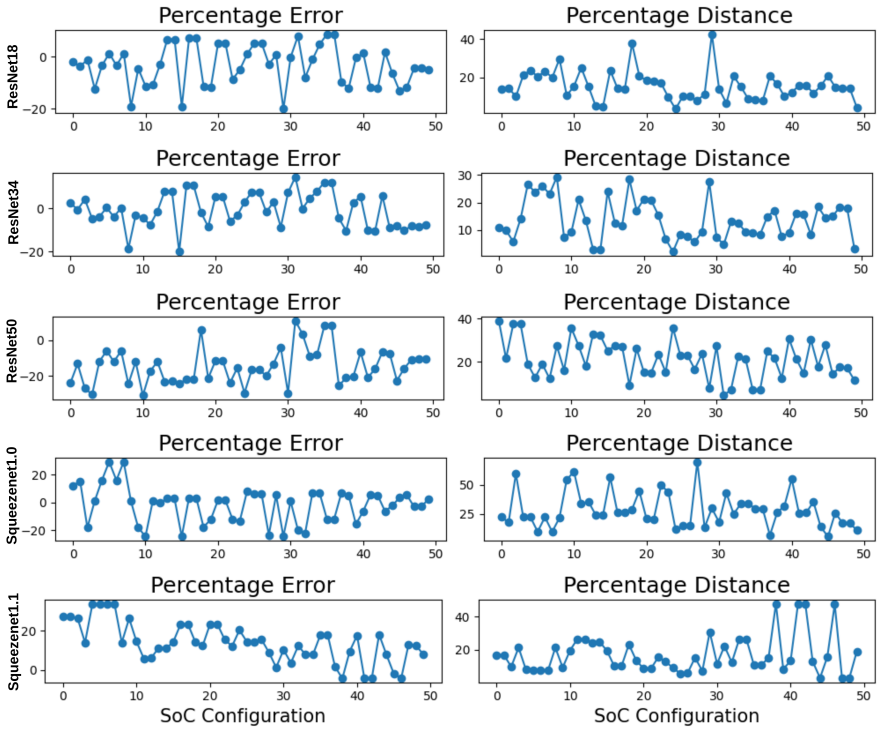

Figure 6.8: Percentage Error and Distance over the selected 50 SOC_CFG

Next, we need to assess how far we are from the `Measured Best Mapping` (MBM), which is the set of PS that provides the lowest measured latency when deployed on the SOC_CFG under test on FPGA. To this end, we introduce the

Percentage Distance (PD) and the Absolute Distance (AD). To evaluate the performances of a significant batch of SOC_CFGs, we selected 50 SOC_CFGs from the possible subsets of resources in our 18 bitstreams. The total dimensionality of this space of SOC_CFGs is given by the sum of the $SOC\_CFG\text{space} = \log_2(N_{\text{ADD}}) \times \log_2(N_{\text{CONV2D}}) \times \log_2(N_{\text{MT}})$ for each bitstream. For all these configurations we evaluate the MBM and we choose the 50 best-performing SOC_CFG. In figure 6.8, we show the evolution of the PE and the PD over the selected 50 SOC_CFG.
To provide a more compact representation of these results, we define the Mean Percentage Error (MPE), the Mean Percentage Absolute Error (MAPE), and the Mean Absolute Error (MAE), as follows:

$$\text{MPE} = \frac{1}{n}\sum_{i=1}^{n}\frac{\hat{y}_i - y_i}{y_i} \times 100 \tag{6.7}$$

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{\hat{y}_i - y_i}{y_i}\right| \times 100 \tag{6.8}$$

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \tag{6.9}$$

where:

- n is the number of SoCs
- y_i is the measured latency
- $\hat{y}_i$ is the predicted latency

To assess the mapping capabilities of the tool, we introduce the Mean Percentage Difference (MPD) and the Mean Absolute Difference (MAD) defined as:

$$\text{MPD} = \frac{1}{n}\sum_{i=1}^{n}\frac{\hat{m}_i - m_i}{m_i} \times 100 \tag{6.10}$$

$$\text{MAD} = \frac{1}{n}\sum_{i=1}^{n}|\hat{m}_i - m_i| \tag{6.11}$$

where:

- n is the number of SoCs
- m_i is the measured latency of the Measured Best Mapping
- $\hat{m}_i$ is the measured latency of the mapping suggested by the tool

We don't need to introduce the Mean Absolute Percentage Difference (MAPD) since the distance from the MBM will always be greater or equal to zero. In particular, it will be zero when PMB and MBM are the same or, in other words, when the mapper is able to identify all the best-performing mappings across the network's layers.

The results of the evaluation of the batch of SOC_CFG are provided in table 6.5. Overall the mapper shows an accuracy in predicting the latency of the PBM that oscillates between 6.6% MAPE for ResNet34 and 14.2% for Squeezenet1.1. The MPD, provided in table 6.6, is instead 13.6% for ResNet34 and 21.2% for ResNet50 that is, as expected, the worst performing network. A final note is on Squeezenet1.0, which shows good results in terms of MPE and MAPE, with a significant degradation in MPD and MAD. This happens because the model fails to find the best mapping for few layers that impact significantly on the final performance due to the smaller dimension of the model. A solution to overcome this problem could be to keep track of these critical layers and include more measurements of similar configurations in the training set.

Target Network	MPE (%)	MAPE (%)	MAE
ResNet18	-3.105	9.605	5672445
ResNet34	-0.646	6.648	7676807
ResNet50	-14.930	16.109	30132143
Squeezenet1.0	1.648	6.680	1724217
Squeezenet1.1	13.502	14.252	2116897

Table 6.5: Mean Inference Error

Target Network	MPD (%)	MAD
ResNet18	14.075	7094909
ResNet34	13.684	13500032
ResNet50	21.276	31893812
Squeezenet1.0	20.167	6104921
Squeezenet1.1	16.912	2284314

Table 6.6: Mean Inference Distance

Chapter 7

Conclusion and future works

In this section, we discuss possible extensions of the mapping tool leveraging the versatility of the latency prediction model that we implemented. Finally, we provide the conclusions of the thesis.

7.1 Design Space Exploration Tool

In section 6.1 we implemented an infrastructure that leverages ML models to optimize the deployment of a CNN on a given input SOC_CFG. Here, we propose a possible alternative to this flow to create a constrained DSE tool. The goal is to provide the user with a Pareto curve with different possible SoCs that are within the boundaries of the given constraints, and how these SoCs will perform when deployed on FPGA.

The first step of this different structure, shown in figure 7.1, is to generate a set of synthetic SoC from the available constraints and proceed to find the PBM for each of them. Once we have the predicted latency for all of them we calculate the area of each SoC, based on the information provided by the FPGA logic synthesizer summing the contribution of each tile independently, then selecting the ones that are Pareto optimal. To reach this goal, we implemented a simple version of the DSE tool in figure 7.1, setting the following constraint:

- Number of CONV2D=32
- Number of EWA=2
- Number of MTs=4

We excluded the micro-architectural parameters inside the CONV2D accelerators, namely PLMs dimensions, MAC parallelism, and data type, fixing them to PLM_IN:

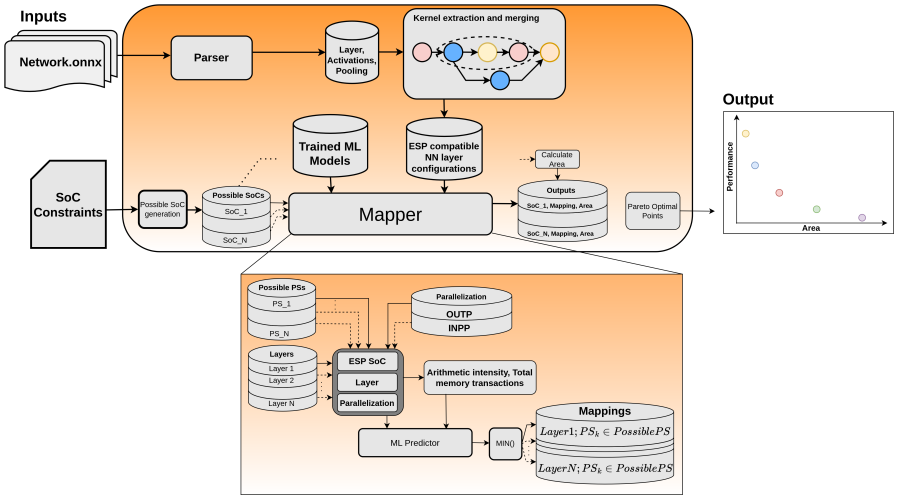

Figure 7.1: Design Space Exploration Tool

4096, PLM_W: 9216, MAC: 8, DATA_TYPE: 2, from the possible SoCs in the DSE. The results of this proof of concept are highlighted in figure 7.2.

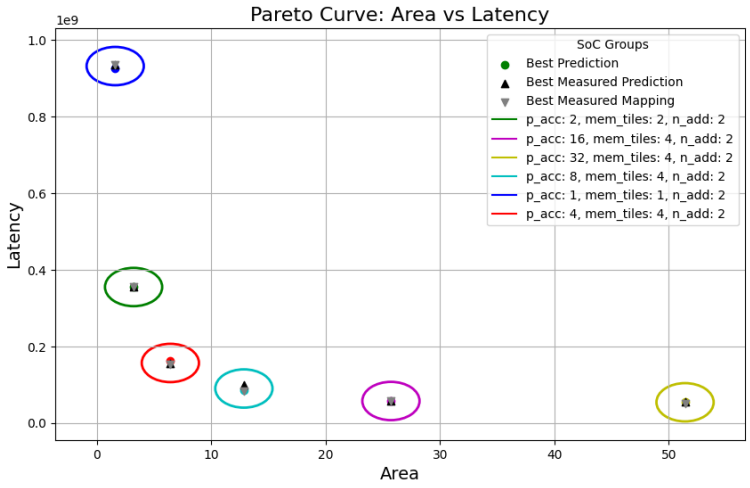

Figure 7.2: Pareto Curve in Output of the DSE tool

7.2 Conclusions

In conclusion, we demonstrate the possibility of mapping popular CNNs on complex tile-based SoCs leveraging a data-driven approach. As opposed to intensive system modeling and development of heuristics, traditionally employed to deploy computational kernels onto hardware efficiently, Machine Learning offers an empirical model that learns specific patterns within raw data. The versatility of a latency predictor could be leveraged by moving towards a structure as the one in 7.1, integrating different metrics in ranking the SoCs and focusing on the micro-architectural parameters. Another interesting possibility could be to model the power performance during FPGA execution and train a model able to predict the power consumption of a given deployment, creating, along with SoC area and CNN execution latency, a multi-objective DSE tool. While our focus was on FPGA platforms, this methodology could be extended to profile other technologies, making it relevant also for ASIC designs. Given the high cost of producing ASIC and the rapid evolution of software applications, especially in the context of machine learning, this approach could be instrumental in mapping new applications efficiently onto existing ASIC.

Bibliography

[1] *AI vs. Machine Learning*. Accessed: Month Day, Year. Columbia University. URL: https://ai.engineering.columbia.edu/ai-vs-machine-learning/ (cit. on p. 1).

[2] Fei-Fei Li, Andrej Karpathy, and Justin Johnson. *Neural Networks Part 1: Setting up the Architecture*. n.d. URL: https://cs231n.github.io/neural-networks-1/ (visited on 10/02/2024) (cit. on p. 1).

[3] GeeksforGeeks. *Multi-Layer Perceptron Learning in TensorFlow*. Accessed: 2024-10-02. 2021. URL: https://www.geeksforgeeks.org/multi-layer-perceptron-learning-in-tensorflow/ (cit. on p. 2).

[4] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. «Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead». In: *IEEE Access* 8 (2020), pp. 225134–225180. DOI: 10.1109/ACCESS.2020.3039858 (cit. on pp. 3, 6–8).

[5] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791 (cit. on p. 3).

[6] Chris Kevin. *Feature Maps*. Medium Article. 2019. URL: https://medium.com/@chriskevin_80184/feature-maps-ee8e11a71f9e (visited on 10/02/2024) (cit. on p. 3).

[7] Yunji Chen et al. «DaDianNao: A Machine-Learning Supercomputer». In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014, pp. 609–622. DOI: 10.1109/MICRO.2014.58 (cit. on p. 9).

[8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. «Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators». In: *IEEE Micro* 37.3 (2017), pp. 12–21. DOI: 10.1109/MM.2017.54 (cit. on p. 10).

[9] Norman P. Jouppi et al. «In-Datacenter Performance Analysis of a Tensor Processing Unit». In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246. URL: https://doi.org/10.1145/3079856.3080246 (cit. on pp. 10, 14, 16).

[10] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. «Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks». In: *IEEE journal of solid-state circuits* 52.1 (2016), pp. 127–138 (cit. on pp. 11, 13).

[11] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, and Markus Püschel. «Applying the roofline model». In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014, pp. 76–85. DOI: 10.1109/ISPASS.2014.6844463 (cit. on p. 12).

[12] Swagath Venkataramani et al. «SCALEDEEP: A scalable compute architecture for learning and evaluating deep networks». In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 13–26 (cit. on p. 12).

[13] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. «Tangram: Optimized coarse-grained dataflow for scalable nn accelerators». In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 807–820 (cit. on pp. 12, 17).

[14] Soroush Ghodrati et al. «Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks». In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 681–697 (cit. on pp. 12, 18, 19).

[15] Zhanhong Tan, Hongyu Cai, Runpei Dong, and Kaisheng Ma. «NN-Baton: DNN workload orchestration and chiplet granularity exploration for multichip accelerators». In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 1013–1026 (cit. on p. 12).

[16] Young H Oh et al. «Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling». In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 584–597 (cit. on p. 12).

[17] Jingwei Cai, Yuchen Wei, Zuotong Wu, Sen Peng, and Kaisheng Ma. «Inter-layer scheduling space definition and exploration for tiled accelerators». In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 2023, pp. 1–17 (cit. on p. 12).

[18] Minsoo Rhu Yujeong Choi. «PREMA: A Predictive Multi-task Scheduling Algorithm For Preemptible Neural Processing Units». In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 220–233 (cit. on pp. 12, 18).

[19] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. «Sigma: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training». In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 58–70 (cit. on p. 12).

[20] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. «A Multi-Neural Network Acceleration Architecture». In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 940–953 (cit. on p. 12).

[21] NVIDIA. *NVIDIA Deep Learning Accelerator (NVDLA)*. www.nvdla.org. 2017 (cit. on p. 20).

[22] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. «Agile SoC development with open ESP». In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD '20. Virtual Event, USA: Association for Computing Machinery, 2020. ISBN: 9781450380263. DOI: 10.1145/3400302.3415753. URL: https://doi.org/10.1145/3400302.3415753 (cit. on pp. 21, 23, 25, 26, 32).

[23] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. «Theory of latency-insensitive design». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.9 (2001), pp. 1059–1076. DOI: 10.1109/43.945302 (cit. on p. 23).

[24] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. «nn-Meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices». In: *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '21. Virtual Event, Wisconsin: Association for Computing Machinery, 2021, pp. 81–93. ISBN: 9781450384438. DOI: 10.1145/3458864.3467882. URL: https://doi.org/10.1145/3458864.3467882 (cit. on pp. 27–29, 58).

[25] Yuji Chai, Devashree Tripathy, Chuteng Zhou, Dibakar Gope, Igor Fedorov, Ramon Matas, David Brooks, Gu-Yeon Wei, and Paul Whatmough. *Perf-SAGE: Generalized Inference Performance Predictor for Arbitrary Deep Learning Models on Edge Devices*. 2023. arXiv: 2301.10999 [cs.LG]. URL: https://arxiv.org/abs/2301.10999 (cit. on p. 27).

[26] Łukasz Dudziak, Thomas Chau, Mohamed S. Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D. Lane. *BRP-NAS: Prediction-based NAS using GCNs*. 2021. arXiv: 2007.08668 [cs.LG]. URL: https://arxiv.org/abs/2007.08668 (cit. on p. 27).

[27] Tianqi Chen and Carlos Guestrin. «XGBoost: A Scalable Tree Boosting System». In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785 (cit. on p. 30).

[28] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. «An Analysis of Accelerator Coupling in Heterogeneous Architectures». In: *Proceedings of the Design Automation Conference (DAC)*. DAC'15. San Francisco, California: ACM, June 2015, 202:1–202:6. ISBN: 978-1-4503-3520-1. DOI: 10.1145/2744769.2744794. URL: http://doi.acm.org/10.1145/2744769.2744794 (cit. on p. 31).

[29] Rangharajan Venkatesan et al. «MAGNet: A Modular Accelerator Generator for Neural Networks». In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942127 (cit. on p. 32).

[30] Deniz Gunay. *Random Forest Analysis*. https://medium.com/@denizgunay/random-forest-af5bde5d7e1e. 2024 (cit. on p. 66).

[31] Jim Frost. *Interpreting R-squared in Regression Analysis*. https://statisticsbyjim.com/regression/interpret-r-squared-regression/. Accessed: 2024-10-07. 2024 (cit. on p. 67).