PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

LEONARDO REZENDE JURACY

A FRAMEWORK FOR FAST ARCHITECTURE EXPLORATION OF CONVOLUTIONAL NEURAL NETWORK ACCELERATORS

Porto Alegre
2022

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

A FRAMEWORK FOR FAST ARCHITECTURE EXPLORATION OF CONVOLUTIONAL NEURAL NETWORK ACCELERATORS

LEONARDO REZENDE JURACY

Doctoral Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Fernando Gehm Moraes
Co-Advisor: Prof. Matheus Trevisan Moreira

Porto Alegre
2022

Ficha Catalográfica

J95f Juracy, Leonardo Rezende

A framework for fast architecture exploration of convolutional neural network accelerators / Leonardo Rezende Juracy. – 2022.
137 f.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Gehm Moraes.
Coorientador: Prof. Dr. Matheus Trevisan Moreira.

1. Convolutional Neural Networks. 2. Convolution Hardware Accelerator. 3. System Simulator. 4. PPA. 5. Design Space Exploration. I. Moraes, Fernando Gehm. II. Moreira, Matheus Trevisan. III. , . IV. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Loiva Duarte Novak CRB-10/2079

LEONARDO REZENDE JURACY

A FRAMEWORK FOR FAST ARCHITECTURE EXPLORATION OF CONVOLUTIONAL NEURAL NETWORK ACCELERATORS

This Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on August 5th, 2022.

COMMITTEE MEMBERS:

Profª. Cristina Meinhardt (PPGCC/UFSC)

Profª. Fernanda Gusmao de Lima Kastensmidt (PGMICRO/UFRGS)

Prof. César Augusto Missio Marcon (PPGCC/PUCRS)

Prof. Matheus Trevisan Moreira (PUCRS- Co-Advisor)

Prof. Fernando Gehm Moraes (PPGCC/PUCRS - Advisor)

AGRADECIMENTOS

Gostaria de deixar aqui, meu muito obrigado a pessoas que contribuíram para o desenvolvimento desta Tese.

Primeiramente, gostaria de agradecer e dedicar este trabalho aos meus pais, Anchieta e Rejane. Pode parecer clichê, mas sem eles, nada disso seria possível e essa Tese não existiria. Muito obrigado por tudo, amo vocês.

Gostaria de agradecer ao meu orientador Fernando Gehm Moraes, que aceitou me orientar a partir da segunda metade do Doutorado. Obrigado pela paciência, pelas horas gastas, e pelos ensinamentos. Se um dia eu tiver um cachorro, tenha certeza que ele terá só um nome.

Ao Alexandre de Morais Amory, que está comigo nessa minha jornada acadêmica há 10 anos. Obrigado pelas orientações durante a bolsa de iniciação científica, trabalho de conclusão de curso, mestrado, e doutorado. Muito obrigado por me orientar durante todos esses anos.

Ao Matheus Trevisan Moreira, que também está comigo desde o trabalho de conclusão de curso. Obrigado por todas as reuniões remotas, por todas as discussões que levaram a esta Tese, e pelas conversas não relacionadas sobre música e afins. Deixo aqui meu muito obrigado.

Agradeço aos professores do PPGCC pelas aulas durante a pós-graduação. Também gostaria de agradecer a secretaria do PPGCC por serem sempre atenciosos.

Queria também agradecer alguns amigos especificos que fiz durante essa jornada acadêmica. Ao Fochi e ao Caimi, pelos conselhos e conversas sobre futebol. Ao Korol, pelas conversas sobre a vida acadêmica e pelas trocas de artigos. Ao Walter Lau Neto, que trabalhou comigo durante a bolsa de iniciação científica, por ouvir as reclamações sobre o andamento do Doutorado. Ao Felipe Kuentzer, que trabalhou comigo e me ajudou muito durante o Mestrado. Ao Carlos Henrique, que me ajudou muito durante a graduação e me indicou para a bolsa de iniciação científica no GAPH. Ao Wachter, que junto com o Amory, foi uma das primeiras pessoas com quem trabalhei. Ao Sergio Johann, pelas conversas sobre guitarra e eletrônica.

Gostaria de deixar meu obrigado a várias pessoas que conviveram comigo durante meus anos de pesquisa, e que trabalharam comigo no GAPH e na DATACOM, e o pessoal do GSE: Castilho, Madalozzo, Augusto Erichsen, Thiago Mânica, L. Heck, G. Heck, Guilherme Medeiros, Guazzelli , Bortolon, Augusto Moraes, Felipe Lazzarotto, e Tanauan. Foram muitas pessoas, e provavelmente esqueci de alguém, mas deixo minhas sinceras desculpas e o meu muito obrigado a todos vocês.

Por fim, gostaria de agradecer à CAPES, que financiou todo esse trabalho.

Mais uma vez, muito obrigado a todos.

UM FRAMEWORK PARA EXPLORAÇÃO RÁPIDA DE ARQUITETURAS DE ACELERADORES PARA REDES NEURAIS CONVOLUCIONAIS

RESUMO

Aprendizado de Máquina (ML, do inglês, *Machine Learning*) é uma subárea da inteligência artificial que compreende algoritmos para resolver problemas de classificação e reconhecimento de padrões. Uma das maneiras mais comuns de desenvolver ML atualmente é usando Redes Neurais Artificiais, especificamente Redes Neurais Convolucionais (CNN, do inglês, *Convolutional Neural Networks*). As GPUs tornaram-se as plataformas de referência para as fases de treinamento e inferência das CNNs devido à sua arquitetura adaptada aos operadores da CNN. No entanto, as GPUs são arquiteturas que consomem muita energia. Um caminho para permitir a implementação de CNNs em dispositivos com restrição de energia é adotar aceleradores de hardware para a fase de inferência. No entanto, a literatura apresenta lacunas em relação às análises e comparações desses aceleradores para avaliar os compromissos Potência-Desempenho-Área (PPA, do inglês, *Power-Performance-Area*). Normalmente, a literatura estima PPA a partir do número de operações executadas durante a fase de inferência, como o número de MACs (do inglês, Multiplier-Accumulator), o que pode não refletir o comportamento real do *hardware*. Assim, é necessário fornecer estimativas de hardware precisas, permitindo a exploração do espaço de projeto (DSE, do inglês, *Design Space Exploration*) para implementar as CNNs de acordo com as restrições de projeto. Esta Tese propõe duas abordagens de DSE para CNNs. A primeira adota um simulador de sistema com precisão de ciclo de relógio e usa uma linguagem de alto nível para descrever o *hardware* de forma abstrata. Essa primeira abordagem, usa o TensorFlow como *front-end* para treinamento, enquanto o *back-end* gera estimativas de desempenho por meio da síntese física de aceleradores de *hardware*. A segunda abordagem, é um DSE rápido e preciso, usando um modelo analítico construído a partir dos resultados da síntese física de aceleradores de *hardware*. O modelo analítico estima a área de silício, desempenho, potência, energia e quantidade de acessos à memória. O erro médio do pior caso observado comparando o modelo analítico com os dados obtidos da síntese física é inferior a 8%. Embora a segunda abordagem permita obter resultados precisos e de forma rápida, a primeira abordagem permite simular um sistema computacional completo, considerando possíveis redundâncias na modelagem de aceleradores. Esta Tese avança o estado da arte, apresentando métodos para gerar uma avaliação abrangente de PPA, integrando estruturas de *front-end* (por exemplo, TensorFlow) a um fluxo de design de *back-end*.

Palavras-Chave: Redes Neurais Convolucionais, Acelerador de Hardware de Convolução, Simulador de sistema, PPA, Exploração do Espaço de Projeto.

A FRAMEWORK FOR FAST ARCHITECTURE EXPLORATION OF CONVOLUTIONAL NEURAL NETWORK ACCELERATORS

ABSTRACT

Machine Learning (ML) is a sub-area of artificial intelligence comprehending algorithms to solve classification and pattern recognition problems. One of the most common ways to deliver ML nowadays is using Artificial Neural Networks, specifically Convolutional Neural Networks (CNN). GPUs became the reference platforms for both training and inference phases of CNNs due to their tailored architecture to the CNN operators. However, GPUs are power-hungry architectures. A path to enable the deployment of CNNs in energy-constrained devices is by adopting hardware accelerators for the inference phase. However, the literature presents gaps regarding analyses and comparisons of these accelerators to evaluate Power-Performance-Area (PPA) trade-offs. Typically, the literature estimates PPA from the number of executed operations during the inference phase, such as the number of Multiplier-Accumulators (MAC), which may not reflect the actual hardware behavior. Thus, it is necessary to deliver accurate hardware estimations, enabling design space exploration (DSE) to deploy CNNs according to the design constraints. This Thesis proposes two DSE approaches for CNNs. The former adopts a cycle-accurate system simulator and uses a high-level language to describe the hardware abstractly. This first approach uses TensorFlow as a front-end for training, while the back-end generates performance estimations through physical synthesis of hardware accelerators. The second approach is a fast and accurate DSE, using an analytical model fitted from the physical synthesis of hardware accelerators. The analytic model estimates area, performance, power, energy, and memory accesses. The observed worst-case average error comparing the analytical model to the data obtained from the physical synthesis is smaller than 8%. Although the second approach generate accurate results in a fast way, the first approach enables simulating a complete computational system, considering a possible accelerators modeling redundancy. This Thesis advances the state-of-the-art by offering methods to generate a comprehensive PPA evaluation, integrating front-end frameworks (e.g., TensorFlow) to a back-end design flow.

Keywords: Convolutional Neural Networks, Convolution Hardware Accelerator, System Simulator, PPA, Design Space Exploration.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ANN – Artificial Neural Networks

ASIC – Application Specific Integrated Circuit

BOP – Bit Operations Performed

CNN – Convolutional Neural Networks

DDDG – Dynamic Data Dependence Graphs

DMA – Direct Memory Access

DSE – Design Space Exploration

DSL – Domain-Specific Language

DNN – Deep Convolutional Neural Networks

FC – Fully Connected

FG – Fine-grained

GOPS – Giga Operations Per Second

GPU – Graphic Process Unit

HBM – High Bandwidth Memory

HLS – High-Level Synthesis

IFMAP – Input Feature Map

IOT – Internet of Things

IS – Input Stationary

MAC – Multiplier-Accumulator

ML – Machine Learning

NLR – No Local Reuse

NPU – Neural Processing Unit

NVDLA – NVIDIA Deep Learning Accelerator

OFMAP – Output Feature Map

OS – Output Stationary

PE – Processing Element

PPA – Power, Performance, and Area

RELU – Rectified Linear Unit

RTL – Register Transfer Level

SDF – Standard Delay Format

SIMD – Single Instruction Multiple Data

SOC – System-on-Chip

TLM – Transaction-Level Modeling

TOPS – Tera Operations Per Second

TPU – Tensor Processing Unit

VCD – Value Change Dump

WS – Weight Stationary

CONTENTS

1. INTRODUCTION

Machine Learning (ML) is a sub-area of artificial intelligence that contains a class of algorithms able to solve problems involving knowledge and "learning" characteristics from determined patterns. This allows decision capability [Goodfellow et al., 2016] and has re-emerged as a solution for problems of classification and pattern recognition. Many applications can use ML, such as computational vision, virtual reality [Facebook, 2022a], voice assistants [Google, 2022b], chatbots [ServiceNow, 2022], and self-driving vehicles [Tesla, 2022].

One of the most common ways to deliver ML nowadays is by using Artificial Neural Networks (ANN). ANNs are based on the human brain and perform data processing by mimicking synapses using thousands of neurons interconnected in a network. The synapses are composed of a data input sample plus a weight that works similar to a filter [Goodfellow et al., 2016]. Incoming synapses of a neuron are added up, and are the input to an activation function, which creates an output to be used in synapses of the next neurons [Haykin, 2009].

A common type of ANN is Convolutional Neural Networks (CNN). These networks gained popularity because they enable efficient computation of computer vision tasks, which became widespread in the last decade. CNNs have the advantage of having sparse connections, in contrast to fully connected ANNs, where all neurons of one layer are connected to all neurons of the next layer. This brings many computational benefits, such as less memory storage for weights of synapses, and there is more reuse of weights read from memory [Goodfellow et al., 2016]. Figure 1.1 illustrate a CNN and an classification application.

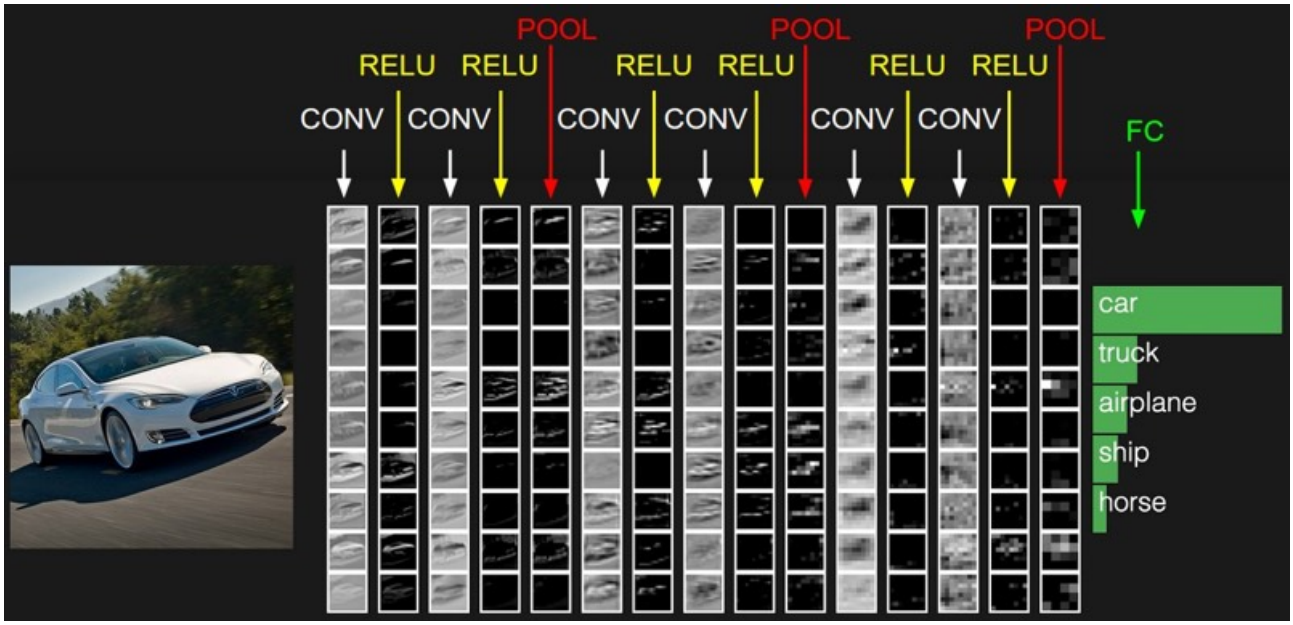


Figure 1.1: Example of a CNN and a Classification Application [CS231n, 2022].

A CNN contains four main layers:

1. convolutional layer (CONV in Figure 1.1), which is the CNN core and performs the synapses by multiplying and accumulating weights and input feature maps;
2. activation function (RELU in Figure 1.1), a nonlinear transformation sent to the next layer of neurons;
3. pooling layer (POOL in Figure 1.1), used to reduce the amount of data processed by the CNN;
4. fully connected layer (FC in Figure 1.1), used in the classification result.

The deployment of CNNs applications is typically divided into two phases [Haykin, 2009]:

1. training, which is the phase where the value of weights of synapses are defined;
2. inference, uses the weights previously computed during the training phase to classify or predict output values based on inputs. A well-trained CNN can correctly generate such classifications or predictions for new inputs, not used in the training phase.

The success of CNNs led to the development of frameworks that help developers to build their models by offering mechanisms required for training and inference. Examples of frameworks include Caffe [Caffe, 2022], Pytorch [PyTorch, 2022] and TensorFlow [TensorFlow, 2022]. These frameworks use a high-level approach to abstract the implementation of functions, such as convolution, and aid in implementing CNN applications. Also, these frameworks abstract the training phase by implementing functions like back-propagation algorithms. Usually, ANNs are trained on GPUs due to their parallelism capability [Chen et al., 2016b, Strom, 2015], reducing the time spent in training.

The inference is commonly executed in CPUs. However, CPU architectures, either based on Harvard or Von Neumann models, drastically affect the performance of the software executing inference of a CNN. For example, CPUs typically do not have multiply-accumulate (MAC) instruction. This kind of operation is usually split in n sums and multiplications instructions, which means that for each instruction, a memory fetch is performed, decreasing the performance. CNN applications like AlexNet [Krizhevsky et al., 2017] require billions of operations to process a single input, resulting in poor CPU performance. Even with optimized instruction set architectures, CPUs are inefficient in performance and energy.

Thus, GPUs became the reference platform for training and inference due to their tailored architecture to the CNN operators. The main GPU drawback is its considerable energy consumption. Considering energy-constrained applications, such as Internet of Things

(IoT), autonomous driving, and wearable devices, the adoption of specialized hardware for computing inference became a trend in the inference phase.

CNN *hardware accelerators* are a suitable replacement for CPUs and GPUs for the inference phase [Dally et al., 2020]. CNN accelerators can reduce power dissipation and/or improve throughput [Chen et al., 2016b, Andri et al., 2017, Shivapakash et al., 2020]. Also, consumer products are increasingly receiving these blocks [Hsiao et al., 2020, Spagnolo et al., 2020, Hsiao and Chang, 2020]. Most of these accelerators are application-specific and can focus only on one characteristic to optimize, such as power, performance, or area [Tesla, 2019, Apple, 2022].

It is necessary to model the following components to implement a CNN hardware accelerator:

1. input buffers, used to store the CNN values;
2. MAC array, the unit that processes the convolution operation. The MAC array can be a matrix (2D architecture) or a vector (1D architecture);
3. activation function, such as Sigmoid, Rectified Linear Unit (ReLU), leaky ReLU [Keras, 2022];
4. output control logic, it is used to communicate with the accelerator and the output memory.

Besides these components, the literature presents hardware accelerators using different approaches to access the memory, called *dataflows types*. The most common dataflow types are weight stationary (WS), input stationary (IS), and output stationary (OS). The main difference between these architectures is how accelerators access data (input feature map and weight tensors) and compute the output (output feature map tensor).

However, the literature presents gaps regarding analyses and comparisons of these accelerators. Even with a representative number of accelerators using different implementations, there is a lack of works exploring the trade-offs between implementations. For example, Eyeriss proposes a comparison between different accelerators but lacks performance or area trade-offs evaluation [Chen et al., 2016b]. Also, some works compare accelerators considering different technology nodes, resulting in an unfair analysis [Das et al., 2020].

Literature shows works focused on frameworks to analyze and perform design space exploration (DSE) of CNN hardware accelerators. These works are based on analytical approaches to estimate the power, performance, and area (PPA) of these accelerators [Heidorn et al., 2020, Zhao et al., 2020] to a given hardware constraint. System simulators [Parashar et al., 2019, Muñoz-Martínez et al., 2020] are important tools for executing DSE. These simulators are typically described in high-level abstraction languages, like Python and C++, reducing the design time and providing PPA evaluation. However, both analytical and

simulator approaches present as drawbacks the PPA accuracy, typically estimated from the number of executed operations, as MAC [Parashar et al., 2019, Wu et al., 2019]. Despite the efforts to increase the abstraction level for accelerators using high-level synthesis (HLS) [Giri et al., 2020, Venkatesan et al., 2019], this approach also has challenges related to performance and power estimation, and performing DSE.

1.1 Thesis Statement

It is possible to execute fast and accurate design space exploration (DSE) for machine learning accelerators, considering different CNN architectures models using standard frameworks. The DSE flow must be comprehensive in terms of power, performance, and area (PPA) estimation. Providing PPA enables the designer to select the most relevant parameters (according to the literature) to design a hardware accelerator.

1.2 Objectives

The strategic objective of this Thesis is to propose a method to perform a *fair* design space exploration in a *fast* and *accurate* way to enable the estimation related to the costs of selecting hardware accelerator parameters. Hardware parameters include the number of accelerators in parallel, accelerator type (1D, 2D), and dataflow (WS, IS, OS).

The following specific goals must be fulfilled to attain the strategic goal:

1. **CNN framework integration** (Chapter 3). Integration of a high-level framework (as TensorFlow) with an accelerator library. The goal is to define how to integrate high-level models with low-level data, obtained from the physical synthesis. To fulfill this goal, third-party accelerators are used, being the focus of this goal the framework and optimization related to the hardware cost, as data and weights quantization;

2. **CNN hardware accelerator design** (Chapter 4). Design of different CNN hardware accelerators, including 1D and 2D arrays, and WS, IS, and OS dataflow types. The goal is to build a library of accelerators to allow the extraction of PPA values;

3. **Comparison method** (Chapter 5). Propose an approach to compare different accelerators types. The goal is to define the method to compare the accelerators using the same parameters, such as technology node, frequency, and memory type;

4. **CNN hardware accelerator physical synthesis** (Chapter 6). Definition and execution of a physical synthesis flow for the hardware accelerator library. The goal is to have a flow that enables to extract automatically accurate PPA;

5. **PPA extraction method** (Chapter 6): Define a method to extract PPA data, using inputs from an CNN application. The goal is to characterize the accelerators using actual switching activity to produce accurate power estimations.

6. **DSE method** (Chapters 3 and 6): Execute the DSE in the high-level framework. The goal is to execute a *fair* DSE, in a *fast* and *accurate* way.

1.3 Original Contributions

We can state that this Thesis presents 4 original contributions in relation to the state-of-the-art:

1. Adoption of TensorFlow as a front-end framework to perform DSE for CNN hardware accelerators. The originality is to use data from the physical synthesis of the complete hardware accelerators, not only from basic components (as MACs).

2. A library of CNN hardware accelerators, considering different array styles and dataflow types. The originality resides in detailing the hardware architecture, considering the accelerator core, the control logic, and the memory interface.

3. A method to fairly compare different CNN hardware accelerators. This method makes it possible to compare accelerators with different dataflows, considering the same characteristics, such as the technology node, frequency, and memory type. It is worth mentioning that the hardware accelerators use as input values extracted from TensorFlow, resulting in accurate power estimations.

4. An analytical method to perform DSE. Using as reference the physical synthesis flow, a set of equations integrated into TersorFlow enables to estimate area, performance and power. The analytical model, integrated into TensorFlow, is the key to obtain a fast and accurate DSE.

1.4 Thesis Structure

Figure 1.2 graphically presents how the text structure is organized. This Thesis contains 7 Chapters:

- Chapter 2 presents the state-of-the-art regarding CNN hardware accelerators, system simulators, and DSE frameworks. Also, this chapter present the contribution of this Thesis compared to the literature;

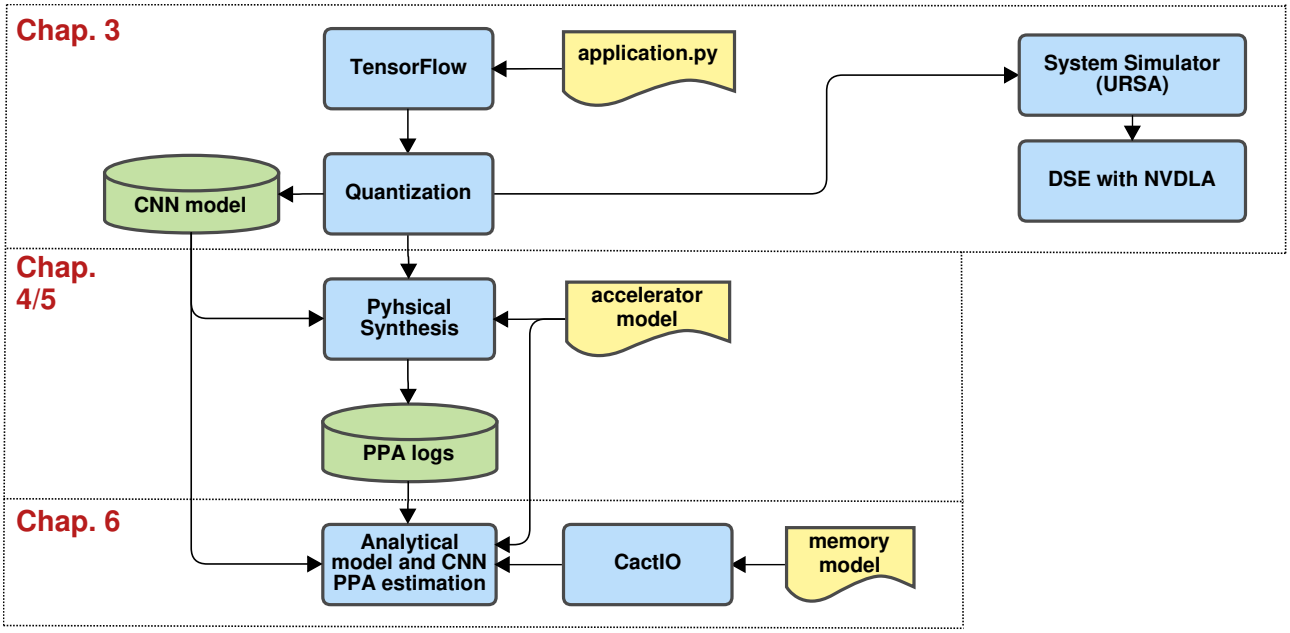


Figure 1.2: Thesis Structure.

- Chapter 3 presents the usage of the TensorFlow framework, and a quantization method to reduce the memory requirements. A third-party accelerator (NVDLA) helps on defining the DSE flow by integrating the front-end (TensorFlow and URSA simulator) to the back-end (physical synthesis);
- Chapter 4 details the RTL implementation of the hardware accelerators used to generate the main results of this Thesis;
- Chapter 5 presents the obtained results based on the hardware described in Chapter 4;
- Chapter 6 describes the DSE method and shows the obtained results;
- Chapter 7 shows the conclusion and directions for future works.

2. STATE OF THE ART

This Chapter presents hardware solutions to accelerate Convolutional Neural Networks (CNNs), as well as the bottlenecks for CPU (performance) and GPU (power). This Chapter describes dedicated and industrial solutions for the bottlenecks, and also works focused on the analyses related to PPA through DSE. This Chapter is organized as follows:

- Section 2.1: describes concepts required to understand the state-of-the-art works;
- Section 2.2: presents descriptions and analyses of academic and industrial CNN hardware accelerators;
- Section 2.3: presents a description and analyses of DSE frameworks and simulators for CNN hardware accelerators;
- Section 2.4: details the contributions of this Thesis and its original contributions.

2.1 Basic Concepts

The CNN concept emerged to deal with problems of visual pattern recognition area [Goodfellow et al., 2016]. For example, the first convolutional network, called LeNet [Al-Jawfi, 2009], was developed to recognize handwritten numbers. Figure 2.1 illustrates a general architecture of a CNN, which contains the followings components:

- Convolution Layer: the core of the CNNs. It executes multiplications and sums of the input values. The convolution uses filters, limiting these multiplications and sums to matrix windows. The filters contain a set of weights, also called parameters. The convolution also uses a variable called stride, which is the number of positions that the filter slides over the input matrix;
- Activation Function: a non-linear function used to help the classification process. This function is applied at the end of a convolution. The most common activation functions are hyperbolic, exponential, and Rectified Linear Unit (ReLU);
- Pooling Layer: this layer has the property to reduce the amount of data to be processed. Unlike the convolutional layer, the pooling layer does not have parameters but is composed of an operation. The more classic operations are the Average Pooling and the Max Pooling. Average Pooling executes an average calculation of the values of a window, while Max Pooling gets the most significant value in a set of values limited by a window;

- Fully Connected (FC)/Dense Layer: the FC Layer is used at the end of a CNN, where all previously output layers are connected with each input for the FC. The output of this layer provides the classification result.

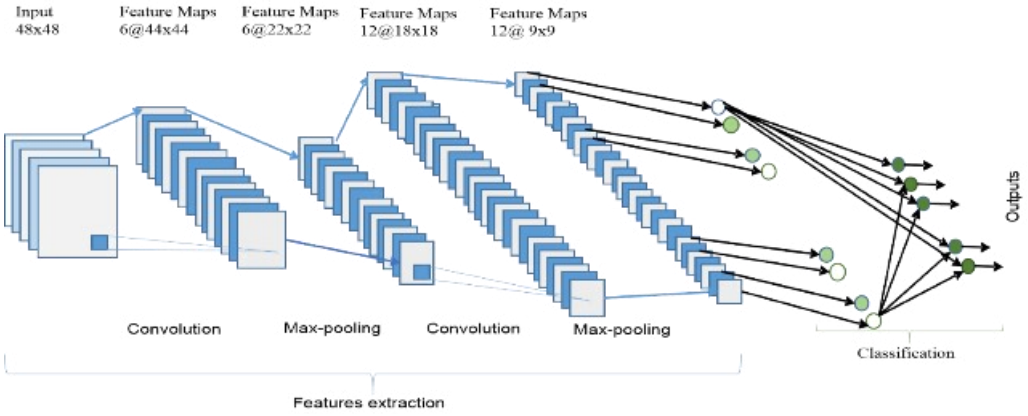


Figure 2.1: Convolutional Neural Network general architecture [Alom et al., 2018].

A metric used to evaluate a CNN is the accuracy, a percentage value representing the amount of data that was classified or recognized correctly. Moreover, the deep learning concept with the Deep Convolutional Neural networks (DNN) emerged together with the CNN [Goodfellow et al., 2016]. It was observed that the increase in the number of convolution layers improved the accuracy metric. Thus, it is possible to solve more complex problems, but at the cost of the increase in the network parameters and, consequently, in memory usage.

As mentioned before, the convolution layer is the core of a CNN and is the main target of the hardware accelerators. Figure 2.2 illustrates a convolution operation. Multiplications are executed between the weights and the input feature map (IFMAP) values that come from an RGB image and accumulate the generated partial value of each operation to generate a complete convolution value. Also, a bias value is added to the sum of the accumulated value, and it is applied to the activation function to generate the output feature map (OFMAP).

Equation 2.1 formally describes the process to obtain one OFMAP result. The convolution receives as inputs the IFMAP tensor (each map may also be called a channel) and another tensor of filters. Then, each filter window is convolved with (and slid across) its respective input channel forming a new set of feature maps. Next, a vector of bias is added to the feature map, generating the final OFMAP tensor (**O**).

$$\mathbf{O}[f][x][y] = \mathbf{B}[f] + \sum_{k=0}^{C-1} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} (\mathbf{I}[k][S_x + i][S_y + j] * \mathbf{WF}[f][k][i][j]) \tag{2.1}$$

where: f, x and y are the current output channel, the horizontal and the vertical position, respectively; C is the total number of input and filter channels, W and H corresponds to the

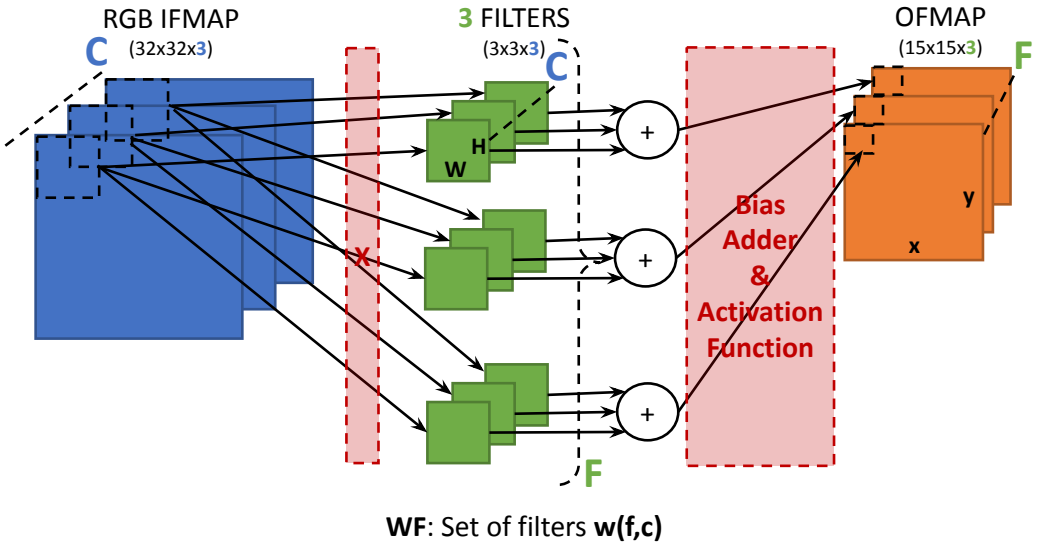


Figure 2.2: Example of a convolution operation: 32x32x3 IFMAP, 15x15x3 OFMAP, 3x3 filter size, stride equal to 2.

filter size; *S* is the stride, and **O** is the output, **I** the input, and **WF** the filter tensors and **B** the bias vector.

The deployment of CNNs comprises two phases: training and inference [Haykin, 2009]. The training phase defines the weight values. The inference phase uses the weights previously computed to classify or predict output values using unknown inputs, which results in the accuracy. The advantages of CNNs regarding classification issues led to the development of frameworks that help developers to build their models by offering mechanisms required for training and inference. Examples of frameworks include Caffe [Caffe, 2022], Pytorch [PyTorch, 2022] and TensorFlow [TensorFlow, 2022]. These frameworks provide libraries to implement Machine Learning applications, including CNNs, which allow performing training and inference phases in a simplified approach based on high-level program languages like Python. Also, these frameworks support the most common CNN functions, such as convolution, max pooling, and ReLU [Keras, 2022].

2.2 Hardware Accelerators

This Section analyses dedicated (mostly academic proposals) and industrial CNN hardware accelerators. For academic approaches, a taxonomy is proposed to classify the accelerators. Figure 2.3 shows the proposed taxonomy, based on [Moolchandani et al., 2021] and extended with other accelerator approaches found in the literature. The proposed taxonomy comprises the following classes: (*i*) array style; (*ii*) convolution techniques; (*iii*) accelerator goal; (*iv*) dataflow type; (*v*) word size; (*vi*) data format.

Array style is the array structure, which can be a vector (1D) or a matrix (2D), systolic or not. Convolution techniques can be executed classically, with multiplications and

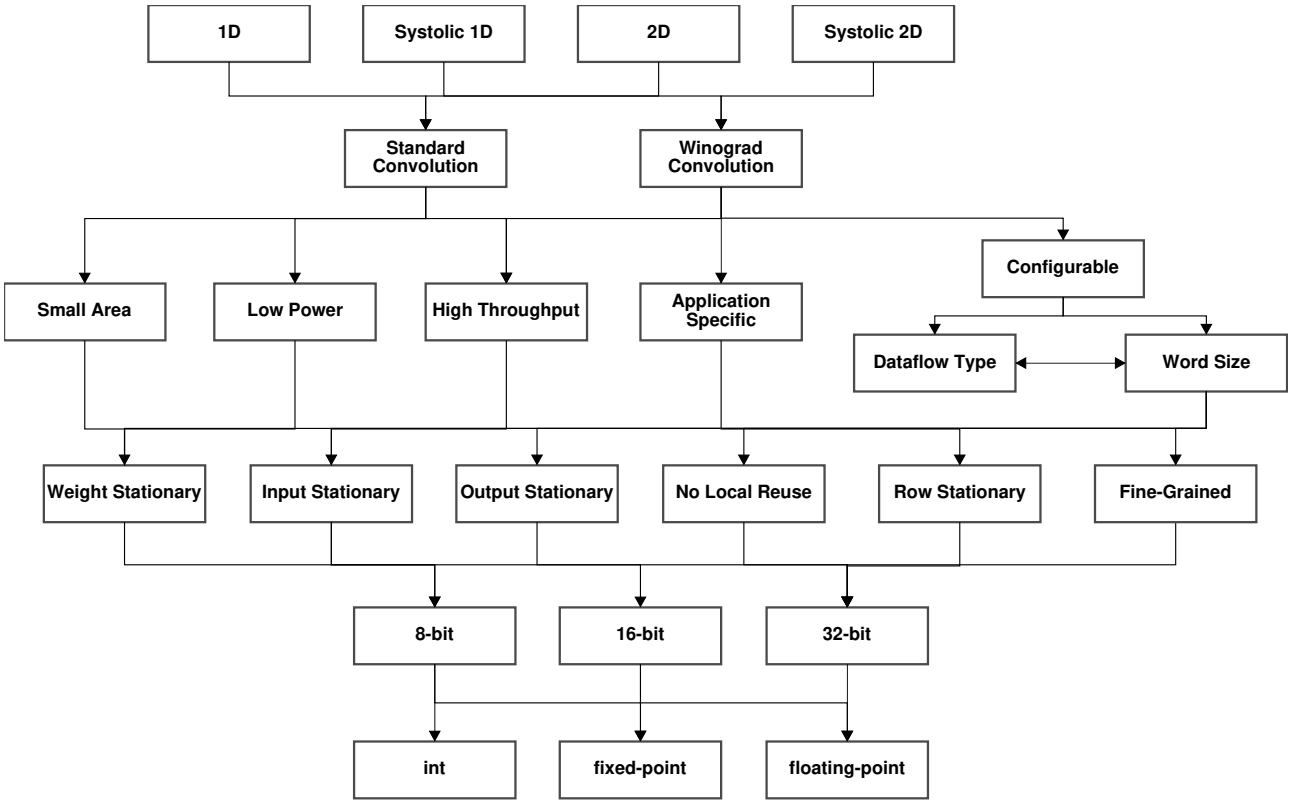


Figure 2.3: Proposed taxonomy for CNN hardware accelerators. This taxonomy is based on [Moolchandani et al., 2021].

accumulations, or using mathematical equivalences, like the Winograd algorithm [Park and Chung, 2020]. The accelerator goal class comprises the focus of the accelerator, like small area or low power. Word size is the input data length, and data format includes integer or float-point values.

Dataflow regards the approach to load the values on the accelerator internal buffers. The state-of-the-art review identified 6 dataflows types: Weight Stationary (WS), Input Stationary (IS), Output Stationary (OS), No Local Reuse (NLR), Row Stationary (RS), and Fine-Grained (FG) [Moolchandani et al., 2021, Xiang et al., 2018].

- The WS dataflow stores the weights in an internal accelerator buffer, aiming their reuse. Thus, each weight value is read once from the input memory, and the convolution is performed using stationary values for weight with values read from memory for the IFMAP window.
- The IS dataflow registers an IFMAP window in an internal accelerator buffer to provide its reuse. The window size is equal to the filter size. Similar to WS, the IFMAP values are read once, and the weight values are read from memory.
- The OS dataflow is based on registering the partial values generated on the convolution. The OS does not present buffers to store the inputs, and each convolution fetches the IFMAPs and weight values in the memory.

- NLR is a dataflow that does not store input and outputs in internal buffers. Thus, each convolution fetches the IFMAPs and weight values in the memory and stores the partial outputs values in the memory.
- RS is a dataflow where the rows of the weight matrix are stored in a processing element (PE). It is similar to the WS dataflow but regards the entire filter row.
- FG is a dataflow that divides the IFMAP and weight matrix into small matrices and stores them into accelerator internal buffers. This dataflow is a mix of WS and IS dataflow.

Some of the proposed taxonomy classes have few examples or can not be applied to this work. Thus, some exclusion criteria were used to define the work scope. The first exclusion criterium was the publication year. We adopted a period from 2015 to 2022, which can be considered relevant to show the proposed work's originality. However, some exceptions are considered due to the relevance or citation number, as Diannao [Chen et al., 2014].

Also, some categories that have few representative works are excluded. For example, accelerators that use methods such as Winograd [Park and Chung, 2020, Ahmad and Pasha, 2020]. Similar occurs to the systolic 1D convolution array type, which has one representative work from 2014 [Gokhale et al., 2014]. The same criterium is used for the dataflow type, where FG has only one example.

The literature presents accelerators focused on networks with binary weights. This kind of accelerator allows replacing complex elements such as float-point MACs with smaller components composed of simple components such as AND gates, which reduce area and power dissipation [Andri et al., 2017, Xian et al., 2020]. Although the claimed advantages of binary architectures, they need to be trained regarding binary values [Courbariaux et al., 2016], and frameworks such as TensorFlow do not support a native binary training. Thus, binary networks are out of the work's scope.

Another approach to design accelerators is HLS. Literature shows works focused on using HLS to design accelerators or proposals that improve the HLS methods [Giri et al., 2020, Venkatesan et al., 2019, Ye et al., 2021, Zacharopoulos et al., 2022, Gerogiannis et al., 2022]. HLS has as an advantage the higher abstraction level to describe accelerators. However, this approach has challenges related to performance and power estimation once the goal of HLS is to generate an RTL description as output. Also, HLS can take a long time to develop a high-performance architecture due to the many design choices at a higher level, requiring more design time [Sohrabizadeh et al., 2021]. Thus, HLS also is out of the scope of this work.

The proposed taxonomy is not applied to industrial accelerators. The goal of analyzing industrial approaches is to show the relevance of the development and analysis of

hardware accelerators for CNN, showing the relevance of this Thesis. Also, industrial accelerators do not detail their designs, preventing comparing these accelerators to our proposal.

2.2.1 Dedicated Accelerators

1D Array Style

Jiao et al. [Jiao et al., 2020] propose a 1D array style programmable neural processing unit (NPU) for data center scenarios, based on WS and IS dataflows. They improve the convolution efficiency and deliver program flexibility by using a large SRAM in the design. Figure 2.4 show the NPU general architecture. It is composed of a local memory (LM), a constant buffer (CB), a tensor engine (TE), a pooling engine, a memory-copy engine (ME), and internal buffers (A-buffer and W-buffer). Multipliers and accumulators compose the TE. CB is used to store values used in operations such as normalization and quantization. ME is used to copy data internally and perform matrix transposition. A command processor controls the communication between the external world and the NPU. A-buffer and W-buffer allow data reuse to reduce memory access. The NPU was fabricated in TSMC 12nm at 700MHz, and results show a throughput of 825 TOPS using an 8-bit integer data format, with an area of $709mm^2$. Using ResNet50-v1 as a study case, the NPU reaches a throughput of 78,563 images per second, with a power efficiency of 500 images per second per watt.

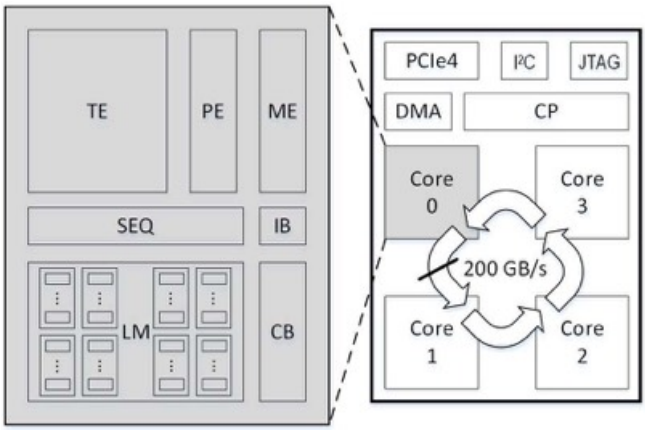


Figure 2.4: NPU general architecture [Jiao et al., 2020].

Hsiao et al. [Hsiao et al., 2020] propose a 1D array configurable accelerator that can execute some DNN operations, like convolution, supporting IS, WS, and OS dataflow types. Figure 2.5 shows the proposed accelerator. The accelerator presents 16 parallel PEs, composed of 8 adder tree multipliers and an adder tree to accumulate them. Thus, it is possible to process in parallel 16 results of 8 input data from 8 input for a convolution operation. A control block manages the operations, and an LPDDR3 DRAM is used as

external memory. The design was synthesized using TSMC 40nm and 28nm technologies, both at 200MHz. The results in a 40nm technology achieved an energy efficiency of 527.8 giga operations per second (GOPS) per watt, while 28nm achieved 1055.7.

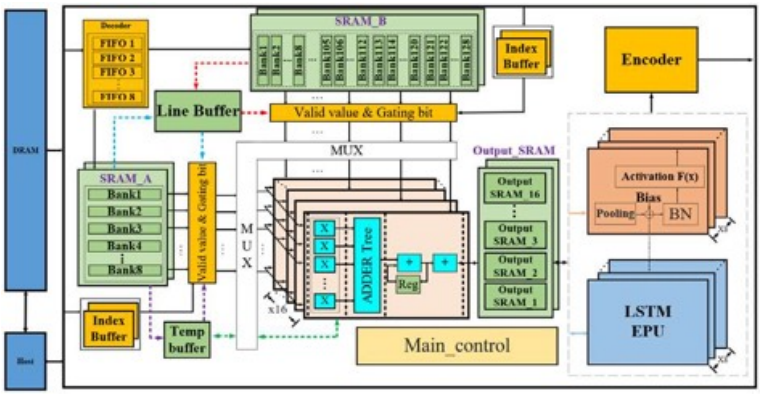


Figure 2.5: Sparsity-aware accelerator architecture [Hsiao et al., 2020].

In another work, Hsiao and Chang [Hsiao and Chang, 2020] extend the accelerator proposed in [Hsiao et al., 2020] to reduce memory accesses and power dissipation. The Authors added an extra bit to flag if the data is zero or not (two-symbol Huffman coding). Also, it is proposed a gating scheme to reduce the switching power that disables the multipliers when a value is zero. The results show a power dissipation of 101.16mW to process one layer of VGG-16 and 89.82mW to process the fully-connected layer. These results mean a decrease of about 25% and 18%, respectively, compared to the execution without exploiting sparsity.

Chen et al. [Chen et al., 2014] propose a 1D array style accelerator focusing on high throughput. The accelerator uses a fixed-point format with 6-bit for the integer and 10-bit for the fractional parts. Figure 2.6 illustrates the accelerator architecture. The first two stages (NFU-1 and NFU-2) operate as normal pipeline stages, and the third stage (NFU-3) is activated after all additions from NFU-2. The accelerator is implemented using OS dataflow, where NBout buffer is used as a circular buffer to store the partial sums and output buffer. NFU-1 and NFU-2 are active every cycle for classifier and convolutional layers, achieving 496 fixed-point operations. The architecture was synthesized using 65nm technology at 0.98GHz, using dual-port SRAM for the buffers. Results show that the accelerator achieves 452 GOPS, with an area of $3.02mm^2$ and 485mW power dissipation. Also, the accelerator can be 117.87 times faster than a 128-bit 2GHz single instruction multiple data (SIMD) processor, reducing the total energy by 21.08 times.

Zhang et al. [Zhang et al., 2015] propose an analytical method based on the roofline model to find the best performance and lowest FPGA resource requirement for an accelerator. The model has a quantitative analysis of throughput and required memory bandwidth as parameters. Figure 2.7 shows the FPGA-based accelerator, which is a 1D array style. It contains PEs, an on-chip buffer, a DDR3 DRAM external memory, and an

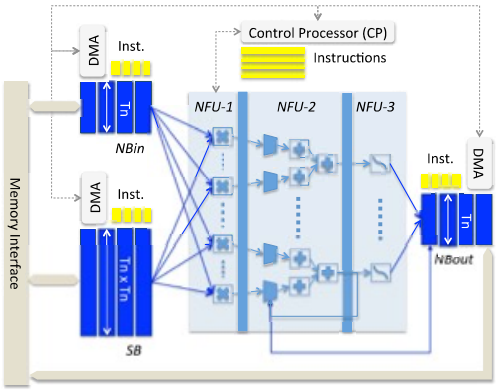


Figure 2.6: DianNao accelerator architecture [Chen et al., 2014].

AXI4 interconnection. A PE has multipliers followed by an adder tree with a 32-bit float-point data format. The data are first stored in the on-chip buffers before being fed to PEs. The accelerator is implemented based on OS dataflow, once partial output sums are reused to reduce memory accesses. The Authors implemented a CNN accelerator on a VC707 FPGA board that reached a peak performance of 61.62 GFLOPS at 100MHz.

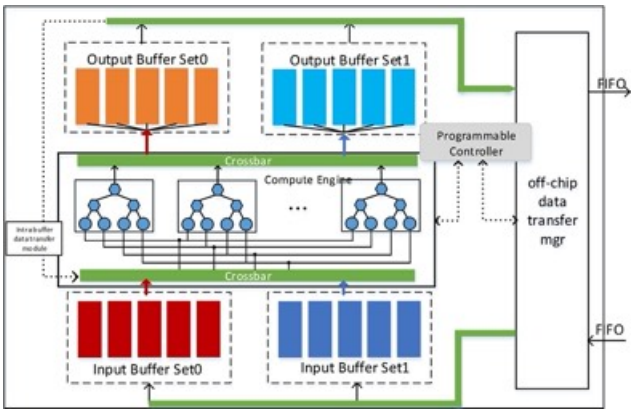


Figure 2.7: FPGA-based Accelerator architecture [Zhang et al., 2015].

Spagnolo et al. [Spagnolo et al., 2020] present a reconfigurable convolution 1D array style architecture designed to support different weights and feature maps at runtime, controlled by a set of multiplexers. The accelerator is implemented using OS dataflow and has four parallel modules, with eight multipliers, eight MACs, and an adder tree, as illustrated in Figure 2.8. The accelerator was prototyped in a Xilinx Ultrascale XCZU9EG SoC, with an 8-bit integer data format. VGG-16 and VGG-S were used as case studies. Results show a throughput of 350.4 GOPS and power dissipation of 145mW at 195MHz.

2D Array Style

Eyeriss hardware accelerator [Chen et al., 2016a, Chen et al., 2016b] focuses on optimizing the energy efficiency of the system, including off-chip DRAM. Eyeriss also allows

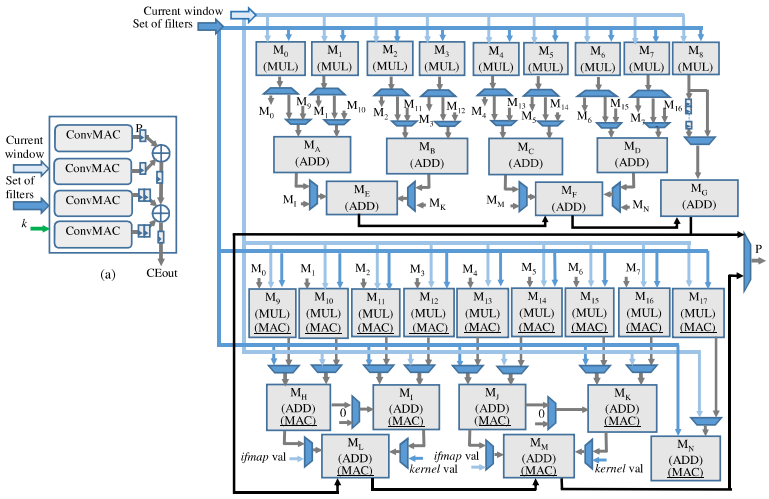


Figure 2.8: Accelerator overview and MAC detailed architecture [Spagnolo et al., 2020].

accelerating many CNN convolution operations. The Authors adopt a RS dataflow, with 168 PEs. Data compression and data gating are used to improve energy efficiency. Figure 2.9 shows the proposed architecture. First, the Eyeriss performs a logical mapping, which maps all the 1D convolution operations. Next, Eyeriss performs a physical mapping, which maps the logic mapping in the physical space. Eyeriss loads in each PE a row from the convolution operation to perform the processing. The IFMAPs is loaded in a horizontal direction, while the weights are loaded in a diagonal direction. Thus, it is possible to perform the multiplication operation of the convolution, and the sum can be performed in the vertical direction. Results show a throughput of 35 frames/s and 0.0029 DRAM access/MAC at 278mW for AlexNet and a throughput of 0.7 frames/s and 0.0035 DRAM access/MAC at 236mW for VGG-16.

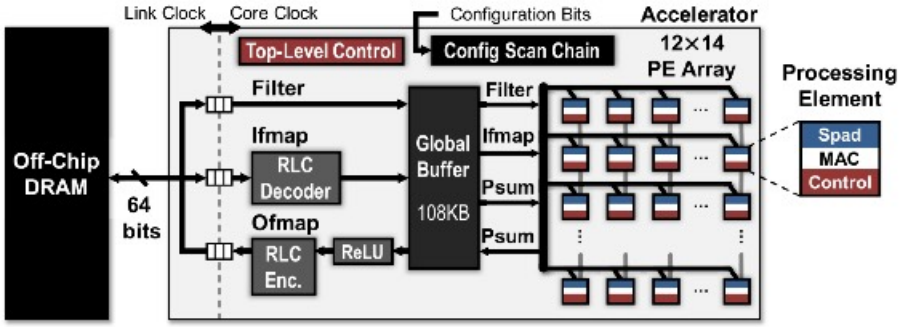


Figure 2.9: Eyeriss general architecture [Chen et al., 2016b].

In another work, the Authors propose Eyeriss v2, an accelerator targeting mobile applications [Chen et al., 2019], also implemented using the RS dataflow. It is a DNN accelerator architecture for sparse DNNs. Also, the work introduces a flexible on-chip network, called hierarchical mesh, that can adapt to allow data reuse and bandwidth requirements of different data types. Figure 2.10 shows the mesh interconnection, global buffers, and the PE clusters. Results show that Eyeriss v2 is 12.6 times faster and 2.5 times more energy-efficient than the original Eyeriss using the MobileNet as a case study.

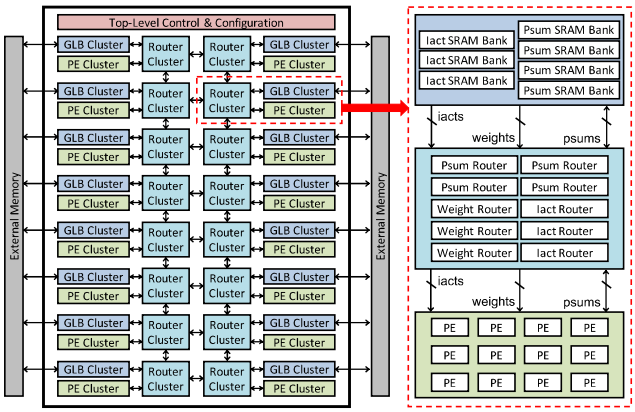


Figure 2.10: Eyeriss v2 general architecture [Chen et al., 2019].

Tavakoli et al. [Tavakoli et al., 2020] present two 2D array style targeting FPGA devices. The first accelerator, Single-Layer Convolution Processor (SLCP), performs the convolution layer by layer. The second accelerator, Multi-Layer Convolution Processor (MLCP), works processing more than one layer in parallel. Both accelerators combine WS and IS dataflow to perform data reuse. Figure 2.11 show both architectures. Each processor unit (PU) has a 3x3 MAC matrix. SCLP uses BRAM to store the data from external DRAM, while MLCP does not have BRAMs. MLCP architecture is similar to SCLP once both have PUs, accumulators, and max pooling modules. However, MLCP is arranged in a pipeline form to allow parallelism. MLCP allows reducing the access to on-chip memory and memory bandwidth. According to the results, MLCP accelerator achieves 12.9 GOPS, 2.6 times faster than SLCP. The accelerators were prototyped on a Xilinx Zynq XC7Z020 chip at 200MHz.

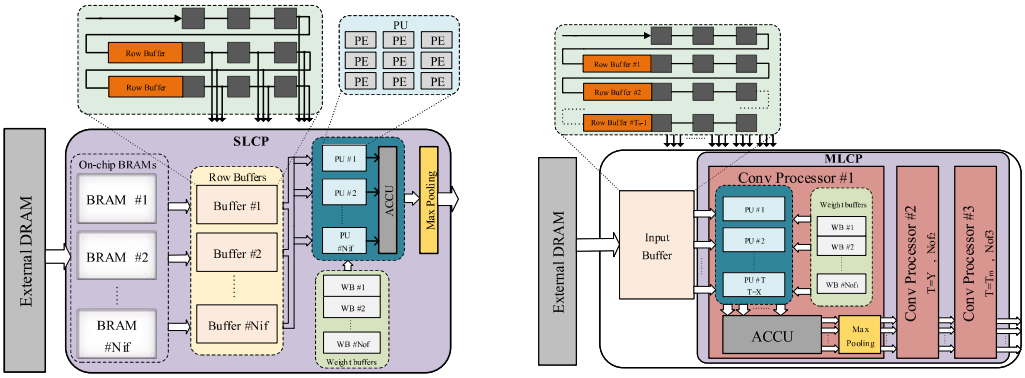


Figure 2.11: SLCP and MLCP accelerator architectures [Tavakoli et al., 2020].

Shivapakash et al. [Shivapakash et al., 2020] propose a power-efficient multi-bit 2D array style accelerator, using a truncating technique for the partial sum results provided by the previous layer of a neural network. Figure 2.12 shows the accelerator architecture. The IFMAP scratchpad memory is implemented in a 4KB SRAM, and the filter scratchpad memory is implemented using 11 32-bit registers. The PE array adopts a RS dataflow. The PE array contains an 11x11 MAC matrix. The $2N$ bits multiplication output is truncating to N bits based on the fixed point q-format. The proposed architecture is prototyped in a

KINTEX-7 KC705 FPGA, and results show that it is possible to preserve accuracy when using 12-bit or more for the word length. Results are based on 8-bit to 20-bit word length. The proposed truncation reduces the FPGA resources and presents a power reduction of 50% when compared to a 32-bit architecture.

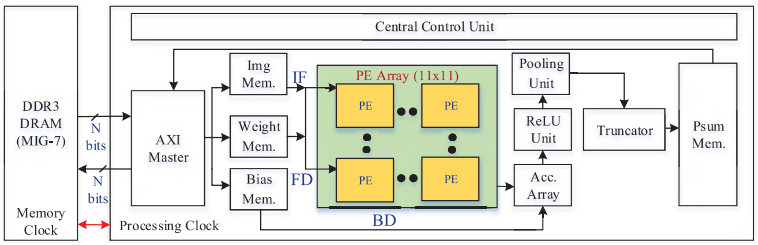


Figure 2.12: Multi-bit accelerator architecture [Tavakoli et al., 2020].

Bai et al. [Bai et al., 2020] propose a scalable neural network 2D array architecture for image segmentation. Image segmentation uses two operations: convolution and deconvolution, and both of them use the same structure based on MACs. The Authors also propose an optimization to reduce the memory accesses. Figure 2.13 shows the general architecture, which adopts a WS dataflow. The line buffer is responsible for storing the data from an external DDR memory. Features and weights are quantized for 8-bit. The process engine array has a 3x3 multiplier array followed by an adder tree. Also, the process engine array presents a shift register to control the features and perform the convolution strides. The accelerator was implemented using Simulink and the HDL Coder tools, prototyped in a Xilinx ZC706, and shows a throughput of 151.5 GOPS for convolution.

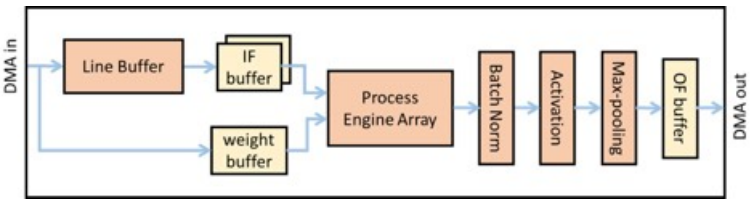


Figure 2.13: Unified convolution and deconvolution accelerator architecture [Bai et al., 2020].

Udupa et al. [Udupa et al., 2020] propose an accelerator for 2D operations based on a z-first storage architecture. The z-first architecture uses several 2D arrays to deliver parallelism for convolution operations, followed by an adder tree and a reuse scheme based on IS dataflow. The architecture is compared with a baseline implementation, which does not perform data reuse, both with 8-bit 16x16 MACs. The architectures were synthesized using Synopsys Design Compiler, using 10nm technology at 800MHz. The power estimation is done using the Spyglass power estimation tool. The results show an improvement in power dissipation of about 1.46 times for convolution operations and about 1.89 times for pooling operations compared to the baseline implementation. Also, the throughput is improved by four times. Memory access is reduced by 1.72 times.

Chen et al. [Chen et al., 2020] propose a dataflow that reuses data to perform stride and uses a mathematical method to reduce memory access using a workload and storage mapping scheme. The accelerator combines both WS and IS dataflow in its implementation. Figure 2.14 shows the accelerator architecture. The 2D array style accelerator has 16×16 PEs, and each PE has a 16-bit fixed-point MAC and registers to store partial sums. It is possible to configure the accelerator to support different convolutional layer dimensions. The accelerator is synthesized in 65nm technology at 500MHz. It also uses a DRAM as external memory and uses the Memory Compiler tool to generate the GBufs and PrimeTime tool for power evaluation. Results are compared to Eyeriss [Chen et al., 2016a] using VGGNet-16 network and show that the proposed dataflow reduces 43.3% memory access than Eyeriss without input compression, and 6.7% compared with input compression.

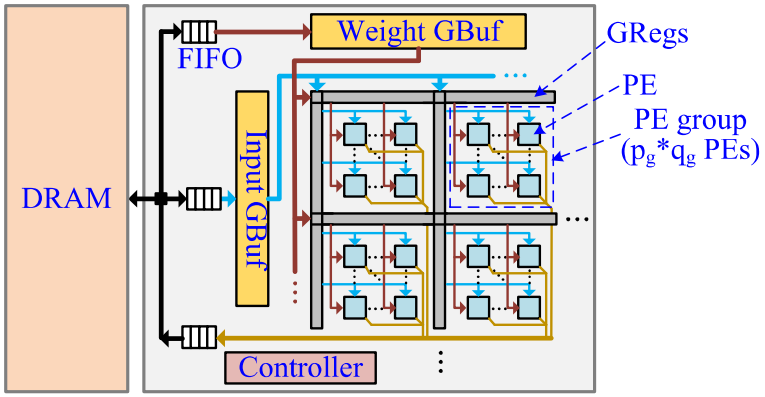


Figure 2.14: Unified convolution and deconvolution accelerator architecture [Chen et al., 2020].

Lu et al. [Lu et al., 2017] propose a new 2D array style architecture called FlexFlow. This architecture supports IS, OS, and WS dataflows. Figure 2.15 shows the accelerator architecture, where each PE has a MAC, and the adders inside of each PE of the same row are connected to build an adder tree. The PE architecture and its interconnection are designed to allow multiple dataflow types. The accelerator has 16x16 16-bit fixed-point PEs, an external DRAM memory, and was synthesized using 65nm technology at 1GHz. When compared to Eyeriss [Chen et al., 2016a], the architecture presents an area reduction of 4 times and a reduction in memory access of 1.22 times.

Du et al. [Du et al., 2015] propose a 2D array accelerator to be used with sensors, without external memory. The accelerator adopts the OS dataflow. Figure 2.16 shows the accelerator architecture, with 8x8 PEs, each PE with a MAC. The input image comes from the sensors, and the accelerator uses 16-bit fixed-point arithmetic. It was synthesized with Synopsys Design Compiler and IC Compiler, using the TSMC 65nm. Results show that it is possible to achieve 60 times more energy-efficient than the previous state-of-the-art accelerators once it does not have external memory, plus the exploration of data access patterns from the sensors. The results also show an improvement of about 50, 30, and

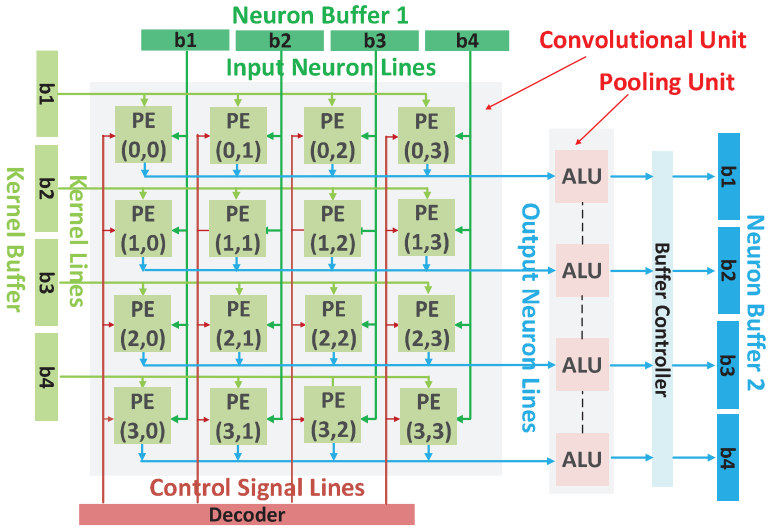


Figure 2.15: FlexFlow accelerator architecture [Lu et al., 2017].

1.87 times in performance compared to Intel Xeon E7-8830 CPU, NVIDIA K20M GPU, and DianNao accelerator [Du et al., 2015].

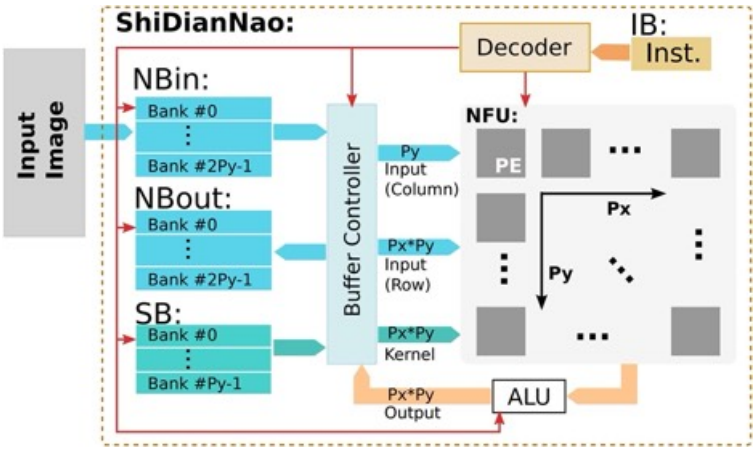


Figure 2.16: ShiDianNao accelerator architecture [Lu et al., 2017].

2D Systolic Array Style

Das et al. [Das et al., 2020] propose a 2D systolic array style accelerator focused on improving energy efficiency and scalability. The accelerator contains vector units, presented in Figure 2.17, for increased power efficiency. It contains an external DDR memory, a global buffer using SRAM, a support logic, and an array. The PE array has 4096 8-bit fixed-point MACs and uses the OS dataflow type. The accelerator was synthesized using 10nm technology, and the results show an average power efficiency of 8.89 TOPS per watt. The proposed accelerator has a better energy efficiency of 2.6 and 3.8 times compared with Samsung NPU and Google Tensor Processing Unit (TPU). However, Samsung NPU and Google TPU were implemented in different technologies, making the comparison unfair.

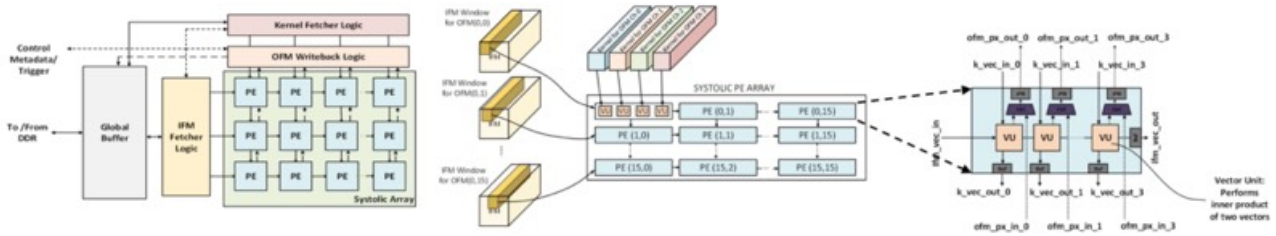


Figure 2.17: ShiDianNao accelerator architecture [Das et al., 2020].

Liu et al. [Liu et al., 2020a] propose a sparsity-aware 2D systolic array style architecture called Swan, using the WS dataflow. The architecture avoids storing zero values in internal buffers, improving computation for accelerators with limited interconnect and bandwidth resources. The architecture has a PE matrix, where each PE has a MAC and a systolic dataflow to reuse inputs for interconnecting and bandwidth saving. Figure 2.18 illustrates the Swan architecture. ABin and ABout store the feature maps and weights, while NPE module allows moving weights and feature maps through the PE array for reuse. The accelerator was synthesized with Synopsys Design Compiler, with 4096 16-bit fixed-point MACs, using TSMC 65nm technology at 600MHz. Results show a throughput of 4915 GOPS and power dissipation of 2.97W. Also, results show that Swan can achieve 1.5 to 2.1 times speedup and 6.0 to 9.1 times higher energy efficiency than state-of-the-art accelerators. However, the compared state-of-the-art accelerators were implemented in a different technology, making the comparison unfair.

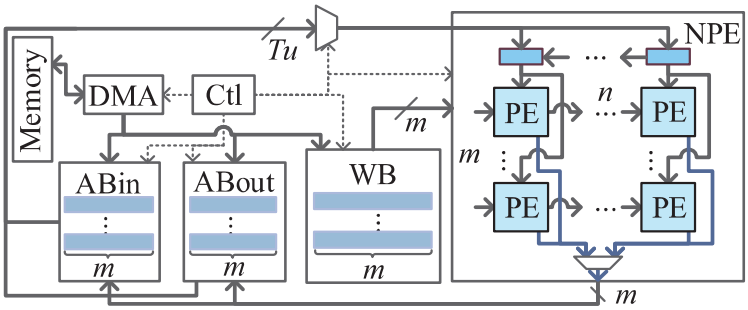


Figure 2.18: Swan general architecture [Liu et al., 2020a].

In another work, Liu et al. [Liu et al., 2020b] propose another 2D systolic array style architecture. Similar to [Liu et al., 2020a], Swallow also avoids the zero values to achieve high PE utilization. Figure 2.19 shows the Swallow architecture. Each PE has a MAC, and NPE has the gating mechanisms to avoid zero values. LRF stores the recent read data to reuse, and ZDet stores the convolution results if it is not zero. WB store the weights and is used to implement a dataflow based on WS. It is possible to reuse values in more than one convolution by shifting them to the neighbor PE. However, there is an underutilization and decreased performance if the IFMAP edge size can not match the fixed PE array size. The synthesis was performed with 65nm technology at 800MHz, and the results show a

peak performance of 614 GOPS, and power dissipation of 1.26W, resulting in an efficiency of 487.3 GOPS per watt.

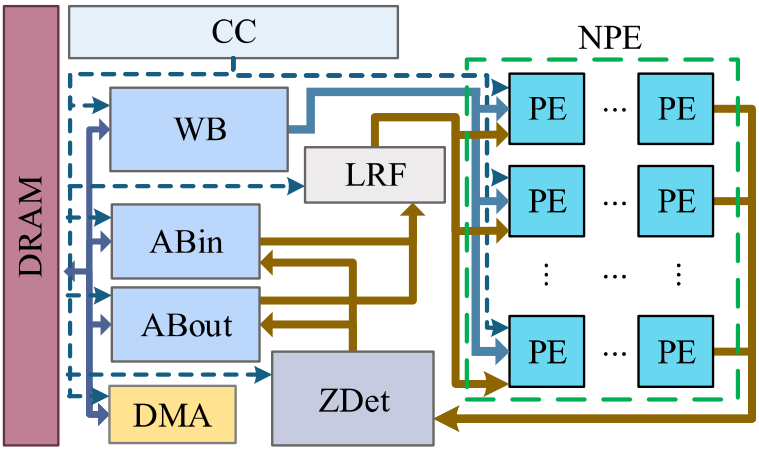


Figure 2.19: Swallow general architecture [Liu et al., 2020b].

Taoran Xiang et al. [Xiang et al., 2018] propose a dataflow called FG dataflow. The Authors present a two-dimensional accelerator called FDPU, shown in Figure 2.20. The FDPU processor consists of PEs, data buffers (Dbufs), instruction buffers (Cbufs), a micro-controller unit (MicC), and a Direct Memory Access (DMA) controller. Dbufs and Cbufs are implemented by using a scratchpad memory. MicC is used for controlling the execution of instructions on PEs. FDPU connects its components using a two-dimensional mesh network, allowing to transfer data values through PEs. The Fine-Grained dataflow divides the IFMAP and weight matrix into small matrices and stores them into Dbuf to execute the convolution. According to the Authors, the number of memory accesses increases in the FD dataflow compared to other dataflows, such as WS and IS. FDPU was implemented in 45nm technology, and the area and power of a single FDPU tile measured are approximately 44.71mm^2 and 3.27W. Results show that the performance of FDPU is improved over GPU (3.11 times faster), and energy consumption of FDPU is reduced (8.52 times) when executing AlexNet.

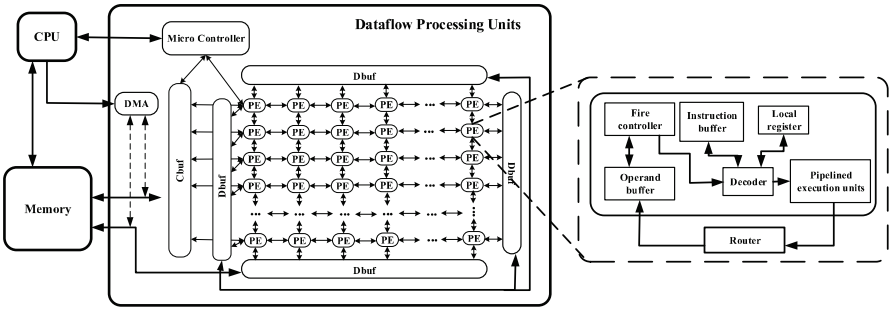


Figure 2.20: FDPU general architecture [Xiang et al., 2018].

Sungju Ryu et al. [Ryu et al., 2022] propose an area and energy-efficient precision scalable accelerator called BitBlade. The proposed accelerator allows performing sums with different input sizes (2/4 bits). Figure 2.21 shows the accelerator architecture. Each PE

receives a group of values with the same bit size. This accelerator use shift operations to adequate the data size through the PEs. Also, PE arrays are implemented using the WS dataflow. A chip was implemented in 28nm CMOS technology, and results show that the throughput and system-level energy efficiency were increased by up to 7.7 and 1.64 times higher than state-of-art accelerators like [Jiao et al., 2020].

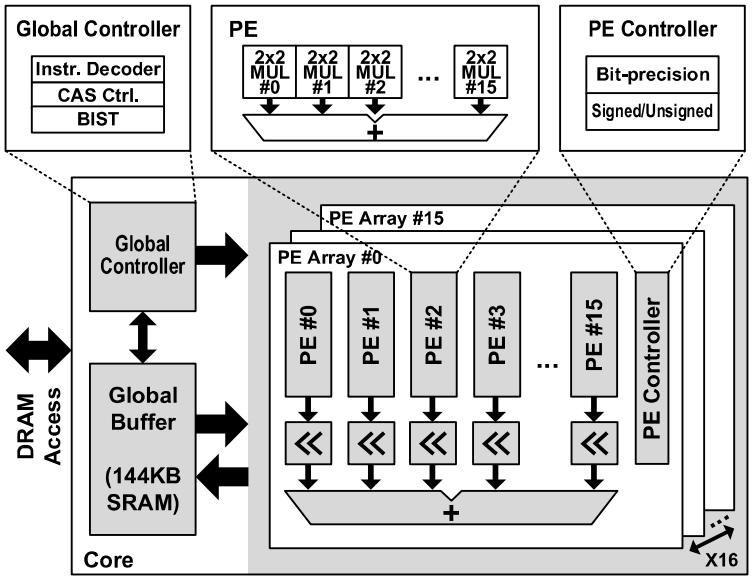


Figure 2.21: BitBlade general architecture [Ryu et al., 2022].

Li Du et al. [Du et al., 2017] propose a streaming-based accelerator, illustrated in Figure 2.22. COL BUFFER module is implemented to remap the buffer output to the convolution unit (CU) engine input, composed of 16 PEs. A pre-fetch controller periodically fetches the DMA controller parameters and updates the weights and bias values in the CU. ACCU is the accumulation buffer, and BUFFER BANK stores the inputs and outputs. Also, the proposed accelerator implements a max-pooling function in hardware, which improves energy efficiency avoiding unnecessary data movement to CPU or GPU to process pooling. In addition, the accelerator can compute the max-pooling function in parallel with convolution by using a separate pooling unit, thus achieving throughput improvement. This accelerator also provides a filter decomposition technique to support arbitrary convolution window size with additional zero-padding values. A prototype accelerator was implemented in TSMC 65nm technology with a core size of $5mm^2$. The hardware precision is 16-bit fixed-point, with WS dataflow, and the accelerator can support major CNNs and achieve 152 GOPS peak throughput and 434 GOPS per watts energy efficiency at 350mW.

Weison Lin and Tughrul Arslan [Lin and Arslan, 2021] propose a column streaming-based accelerator. The Authors propose a method where the weights are stationary in PEs (such as WS), and the IFMAP is divided into columns to be processed in the convolution. Figure 2.23 shows the general architecture and also illustrates the IFMAP column division. The data can be transmitted through the PE array in a wavefront interconnection (PEs are

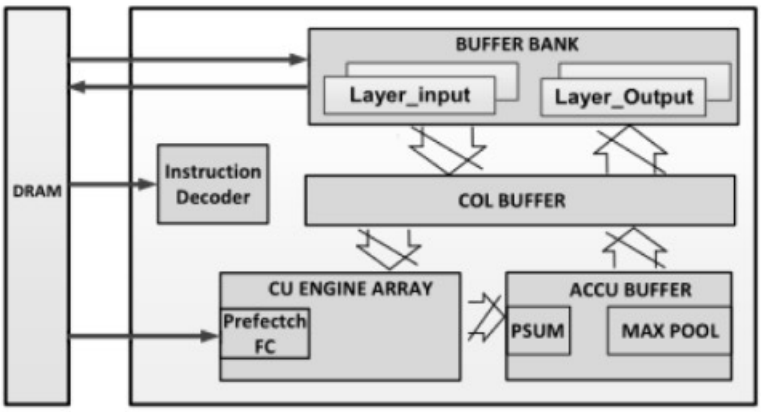


Figure 2.22: Streaming-based Accelerator general architecture [Du et al., 2017].

connected diagonally). The results are generated comparing with the [Du et al., 2017], and show that [Lin and Arslan, 2021] has a larger area and delay than [Du et al., 2017]. However, [Du et al., 2017] eliminates the zero-padding overhead, and the number of cycles is close. PPA results and setup are not addressed in this work.

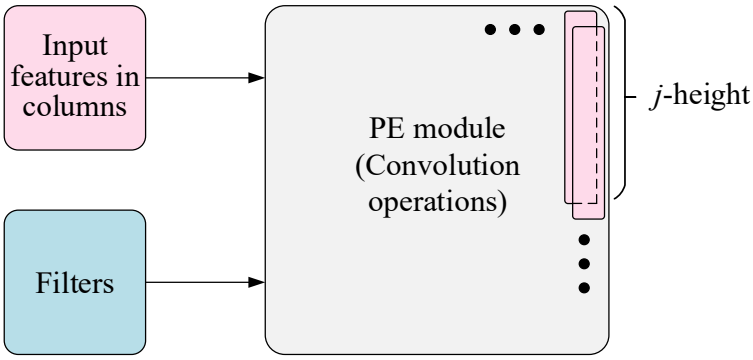


Figure 2.23: Column Streaming-based Accelerator general architecture [Lin and Arslan, 2021].

Boming Huang et al. [Huang et al., 2021] propose an In-Execution Configuration Accelerator (IECA) using the RS dataflow. The accelerator is implemented in two steps: (*i*) 1D array to implement a row; (*ii*) a tile with three rows, which configures a 2D systolic array. The accelerator uses more than one tile, resulting in a 3D-tiled accelerator, as shown in Figure 2.24. IECA uses a delay-chain structure to control when the inputs must be read. A delay-chain structure enables data reuse based on the stride. The accelerator allows diverse convolutional sizes. These delay-chain structures are composed of a register chain that delays the data control signals. A chip was fabricated with UMC 55nm LP technology at 250MHz, using 16-bit fixed-point as precision. The area is 1.658mm X 1.659mm, with 168 PEs and 109kB on-chip SRAM, with power equal to 114.6mW. The delay-chain structures account for 0.55% of the entire chip area, and the area efficiency of the IECA achieves 30.55 GOPS/mm^2 and 2.39% of the total power dissipation.

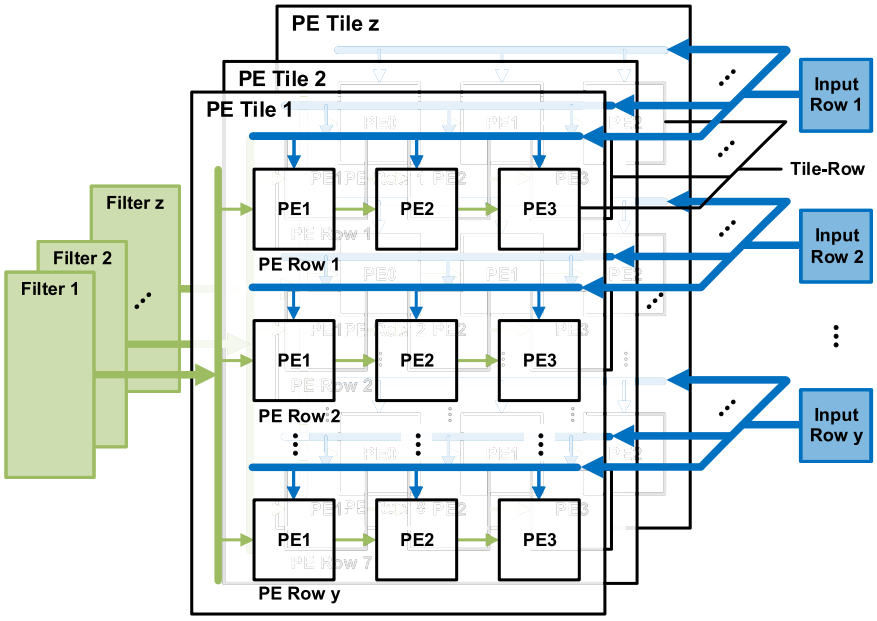


Figure 2.24: IEAC 3D tile [Huang et al., 2021].

Final Remarks Related do Dedicated Accelerators

This Section summarizes works related to dedicated accelerators. Table 2.1 shows characteristics of the described accelerators, using as reference the taxonomy presented in Figure 2.3. Each accelerator has a specific target like small area [Chen et al., 2014] or low power [Hsiao and Chang, 2020]. Few works present accelerators capable of reconfiguring themselves. Examples of reconfiguration include the word data size [Jiao et al., 2020], or the dataflow type.

The analysis of the state-of-the-art reveals the difficulty of performing a PPA comparison. The Authors target ASIC and FPGA technologies. For ASIC design, there are different technology nodes, while FPGA design uses different families. We observed in some works the Authors comparing their results with other technologies, making the comparison unfair. One of our goals is to propose a DSE flow enabling to extract PPA data for different technologies, easing the comparison with related works.

Table 2.1 also shows one of the contributions of this Thesis. The Thesis proposes a set of accelerators covering the three first columns (small area, low power, high performance), with 8 and 16-bit precision. Also, the accelerators are implemented using the WS, IS, and OS dataflow, once these dataflow types are the most commonly adopted by the literature. We reviewed 12 works based on WS, 7 based on IS, and 8 based on OS, while 5 use RS and 1 FD.

Table 2.1: Dedicated accelerators state-of-the-art summary.

Work	Small Area	Low Power	High Performance	Configurable	Precision	Tech/ FPGA	Array Style	Dataflow
[Chen et al., 2014]	yes	no	no	no	16-bit fixed-point	65nm	1D	OS
[Zhang et al., 2015]	yes	no	yes	no	32-bit float	VC707	1D	OS
[Du et al., 2015]	no	no	no	no	16-bit fixed-point	65nm	2D	OS
[Chen et al., 2016a]	no	yes	no	no	16-bit fixed-point	65nm	2D Systolic	RS
[Chen et al., 2016b]	no	yes	no	no	16-bit fixed-point	65nm	2D Systolic	RS
[Lu et al., 2017]	no	no	no	Dataflow Type	16-bit fixed-point	65nm	2D	WS / IS / OS
[Du et al., 2017]	no	yes	no	no	16-bit fixed-point	65nm	2D Systolic	WS
[Xiang et al., 2018]	no	no	yes	no	16-bit fixed-point	45nm	2D Systolic	FG
[Chen et al., 2019]	no	no	no	no	8-bit fixed-point	65nm	2D Systolic	RS
[Jiao et al., 2020]	no	no	yes	Word Size	8/16-bit int	12nm	1D	WS / IS
[Hsiao et al., 2020]	no	no	no	Dataflow Type	16-bit fixed-point	28nm / 40nm	1D	WS / IS / OS
[Hsiao and Chang, 2020]	no	yes	no	no	16-bit fixed-point	40nm	1D	WS / IS / OS
[Spagnolo et al., 2020]	no	no	no	no	8-bit fixed-point	Zynq-U 9EG	1D	OS
[Tavakoli et al., 2020]	no	no	yes	no	8-bits/ max. required value	Zynq XC7Z020	2D	WS /IS
[Shivapakash et al., 2020]	no	yes	no	no	8/20-bit fixed-point (q-ormat)	Kintex-7 KC705	2D	RS
[Bai et al., 2020]	no	no	no	no	8-bit fixed-point	Xilinx ZC706	2D	WS
[Udupa et al., 2020]	no	no	no	no	8-bit (not specified)	10nm	2D	IS
[Chen et al., 2020]	no	no	no	no	16-bit fixed-point	65nm	2D	WS /IS
[Das et al., 2020]	no	yes	no	no	8-bit fixed-point	10nm	2D Systolic	OS
[Liu et al., 2020a]	no	no	no	no	16-bit fixed-point	65nm	2D Systolic	WS
[Liu et al., 2020b]	no	no	no	no	16-bit fixed-point	65nm	2D Systolic	WS
[Lin and Arslan, 2021]	no	no	no	no	NA	NA	2D Systolic	WS
[Huang et al., 2021]	yes	no	no	no	16-bit int/fixed-point	55nm	2D Systolic	RS
[Ryu et al., 2022]	no	yes	no	Word Size	2/4-bit int	28nm	2D Systolic	WS
This thesis	**yes**	**yes**	**yes**	**no**	**8/16-bit fixed-point**	**28nm / 65nm**	**1D / 2D Systolic**	**WS / IS / OS**

2.2.2 Industrial Accelerators

This Section presents an overview of the current neural network accelerators in the industry. Tesla developed an SoC to accelerate machine learning applications [Tesla, 2019]. The system contains 12 ARM CPUs, GPU, on-chip SRAM, ReLU and pooling dedicated hardware, and a 96x96 MAC array. The MAC array produces a result for every clock cycle,

ensuring high throughput. The chip was fabricated using a 14nm technology node. The Tesla Accelerator has 6 billion transistors, with an area of 260mm^2. According to Tesla, the convolution accelerator array can reach a throughput of 72 TOPS at 2GHz. Experiments show that, with a throughput of 35 GOPS, a GPU has an improvement of 14 times regarding frames per second compared to a CPU. Results show that, also with a throughput of 35 GOPS, the Tesla accelerator has an increase in frames per second of 13 times compared to the GPU, totalizing an increase in frames per second of 182 times compared to a CPU. Besides all the improvements, the Tesla accelerator has an expensive hardware cost. However, this hardware is used in a real-time specific application, the autonomous Tesla car, and needs a high throughput to process all data coming from the inputs.

Apple developed an SoC that targeted to mobile devices, called A13 Bionic [Apple, 2022]. The SoC presents four CPU cores, a GPU, and a matrix multiplication accelerator. The SoC was fabricated using a 7nm technology with 8.5 billion transistors. According to Apple, the A13 performance is higher than the other companies' accelerators, such as Snapdragon 855 from Qualcomm and the kirin 980 from Hauwei, reaching almost 50% of improvement. According to Apple, the SoC was developed to reach low power and high performance, with a throughput of 1 TOPS, and a power reduction of 30% compared to the past version, the A12. The low power is achieved by disabling the neural engine once it is not used in all applications. Compared to Tesla accelerator, Tesla demonstrated a throughput 72 times higher than the Apple hardware. However, Apple focuses on mobile applications and does not require the same performance achieved by Tesla. Also, Apple focuses on low power dissipation, a necessary parameter for mobile applications, constraining the design in terms of area. Besides, mobile applications do not need real-time processing, consequently not requiring high throughput.

Google TPU [Google, 2022a] is a custom-designed machine learning ASIC to accelerate machine learning applications, such as network security and medical diagnoses. TPU was developed to perform machine learning applications in the cloud. According to Google, TPU is used in translating, photo manager, search assistant, and e-mail. Also, the Google TPU has a fault-tolerant feature for training. TPU results show an improvement of 27 times accelerating the ResNet-50 neural network training compared to GPUs.

Microsoft designed the Brain Wave NPU [Microsoft, 2022] for real-time machine learning applications, both in the cloud or edge. This approach uses an FPGA to accelerate the neural network inference of applications, such as computer vision and natural language processing, and is used combined with CPUs. Brain Wave NPU can reach low latency, high throughput, high efficiency, and flexibility due to FPGA, according to Microsoft.

Amazon developed the Inferentia chip [Amazon, 2018] to perform neural network inference. It is implemented to reach high throughput and low latency at a low cost. Inferentia supports frameworks to implement neural networks, such as TensorFlow and PyTorch. Also,

Inferentia has a throughput of thousands of TOPS, and it is possible to use more than one chip together to improve this throughput.

Intel proposes the Nervana Neural Network Processor-T (NNP-T) [Intel, 2022], an accelerator for both training and inference. Nervana has two chip versions, one dedicated to the training phase and the other one dedicated to the inference phase. Nervana uses High Bandwidth Memory (HBM) interface to get better performance, with eight channels, using a DDR channel interface combined with the LPDDR power-saving techniques. Nervana can train complex neural networks with low time-to-train, dissipating between 10W and 75W.

Qualcomm Snapdragon [Qualcomm, 2019] is an accelerator focused on mobile devices. It was developed to perform on-chip inference, reducing the latency when the application is processed on the cloud. Snapdragon can be used to perform machine learning tasks like image classification, object detection, face detection, and speech recognition. Snapdragon can achieve a throughput of 7 TOPS and can process more than 140 inferences per second on the Inception-v3 neural network.

Many companies build their accelerators, both for training and inference. Each company has its accelerator, designed according to its constraints. For example, Tesla works with autonomous cars and needs high throughput due to the amount of data that real-time applications must process. Thus, it is necessary to relax constraints related to power and area. On the other hand, Apple focuses on mobile applications that need less throughput capacity than Tesla, having power dissipation as a critical design constraint.

The same occurs in other companies, such as Google and Microsoft, developing accelerators targeting a specific application. Thus, it is possible to note that application-specific accelerators are a market trend. Table 2.2 summarizes the industrial accelerators, including other companies. This Table shows the relevance of the research and development of hardware accelerators.

2.3 Hardware Design Space Exploration Frameworks and Simulators

This section describes works that generate PPA analyses focused on simulators of CNNs and frameworks related to our proposal. Estimation frameworks can use a simulator to estimate PPA based on the hardware behavior or use analytical methods to evaluate PPA. The simulators are commonly implemented using high-level program languages, such as Python and C++, and simulate the CNN accelerator faster than RTL approaches.

Specific-domain frameworks targeting commercial platforms like Vitis AI from Xilinx [Xilinx, 2021] and TensorRT from NVIDIA [NVIDIA, 2022b] aid in model hardware for CNNs. These frameworks model the CNN using frameworks such as TensorFlow, and using HLS, convert the model to the hardware using custom IPs (Xilinx) or specific platforms (e.g., Jet-

Table 2.2: Industrial CNN accelerators.

Company	Product	Function
Amazon [Amazon, 2018]	Inferentia	Inference
Fujitsu [Fujitsu, 2018]	Deep Learning Unit	Inference
Alibaba [Alibaba, 2019]	Hanguang 800	Inference
Huawei [Huawei, 2019]	Ascend 910	Inference/Training
Tesla [Tesla, 2019]	FSD	Inference
Samsung [Samsung, 2019]	Exynos	Inference
Qualcomm [Qualcomm, 2019]	Snapdragon	Inference
Xilinx [Xilinx, 2018]	xDNN	Inference
Toshiba [Toshiba, 2019]	Visconti 5	Inference
Google [Google, 2022a]	TPU	Training
Microsoft [Microsoft, 2022]	Brain Wave NPU	Inference
Facebook [Facebook, 2022b]	Kings Canyon	Inference
Apple [Apple, 2022]	Bionic	Inference
IBM [IBM, 2022]	Watson	Inference/Training
Western Digital [Digital, 2022]	Machine Learning Accelerator	Inference/Training
Intel [Intel, 2022]	Nervana	Inference/Training
NVIDIA [NVIDIA, 2022a]	NVDLA	Inference/Training
Mediatek [Mediatek, 2022]	APU	Inference
Renesas [Renesas, 2022]	e-AI	Inference
Texas Instruments [Texas, 2022]	Sitara	Inference
NXP [NXP, 2022]	S32V234 MPU	Inference
Cerebras [Cerebras, 2022]	CS-1	Inference/Training

son). However, these frameworks use proprietary IPs, limiting the design space exploration. Thus, specific-domain frameworks are out of this Thesis's scope.

Similar occurs to the NVIDIA Deep Learning Accelerator (NVDLA) [NVIDIA, 2022a]. NVDLA is an open-source framework from NVIDIA to implement machine learning applications. The framework presents a full software ecosystem, which includes: (*i*) a complete training infrastructure; (*ii*) a compiler to convert existing models to a form that is usable by NVDLA software. Figure 2.25 shows the NVDLA framework flow. The NVDLA can read a neural network from a front-end environment, such as Caffe, and map to the NVIDIA accelerator. PPA results for a 16nm technology show for a 32 MACs array area of $0.55\mu m^2$, a throughout of 3.6 frames per second, and an average power of 17mW. For a 2048 MACs array, the results show an area of $3.3\mu m^2$, a throughout of 269 frames per second, and an average power dissipation of $291mW$.

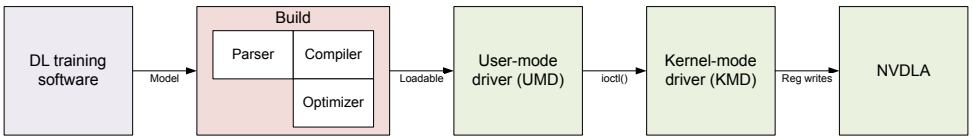


Figure 2.25: NVDLA flow diagram [NVIDIA, 2022a].

NVDLA is open-source, allowing generating case studies to evaluate PPA. However, the framework is restricted only to NVIDIA accelerator architecture, making it hard to compare with other accelerators. Also, to obtain PPA data, it is necessary to perform all the implementation flow, which means more time spent on the project. The same occurs for simulation once it is based on RTL simulation.

2.3.1 Hardware Design Space Exploration Frameworks

MLPAT [Tang and Xie, 2018] is a framework that allows the modeling of power, area, and timing for machine learning accelerators. Figure 2.26 shows the framework architecture.

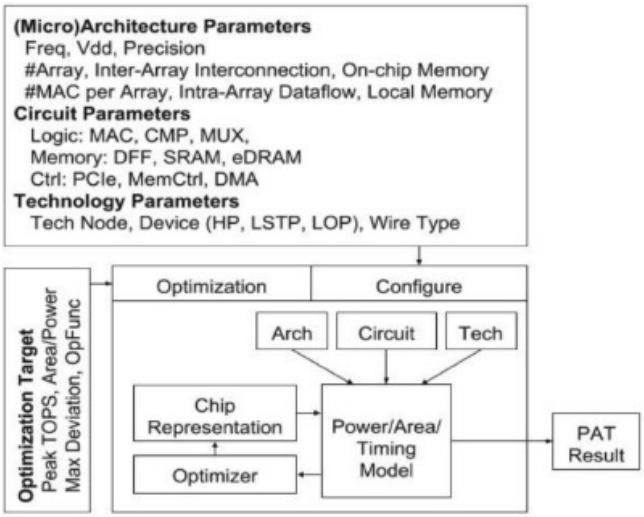


Figure 2.26: MLPAT Framework Architecture [Tang and Xie, 2018].

MLPAT supports modeling components such as systolic arrays, on-chip memory, and activation pipeline. Also, MLPAT supports different precision types, which allows validating the trade-off between accuracy and precision, and different dataflows, such as WS and OS. As input, the MLPAT allows specifying the accelerator architecture, the circuit, and the technology. The framework generates an optimized chip representation to report the results, such as area, power, and performance. The results show an error below 10% when compared with TPU-V1. The MLPAT does not perform simulation, but the Authors mention in the paper that it is possible to combine MLPAT with simulation tools. Thus, it is also possible to

have dynamic power results. Even with area, timing, and power analyses, the power results are inaccurate.

MAESTRO [Kwon et al., 2018a, Kwon et al., 2019] is a framework to describe and analyze neural network hardware, which allows obtaining the hardware cost to implement a target architecture. Figure 2.27 shows the framework architecture. It has a domain-specific language (DSL) to describe the dataflow that allows specifying the number of PEs, memory size, and NoC bandwidth parameters. The results generated by the framework are focused on performance analyses. In recent work, MAESTRO was used to estimate tradeoffs between execution time and energy efficiency for CNN models, such as VGG and AlexNet.

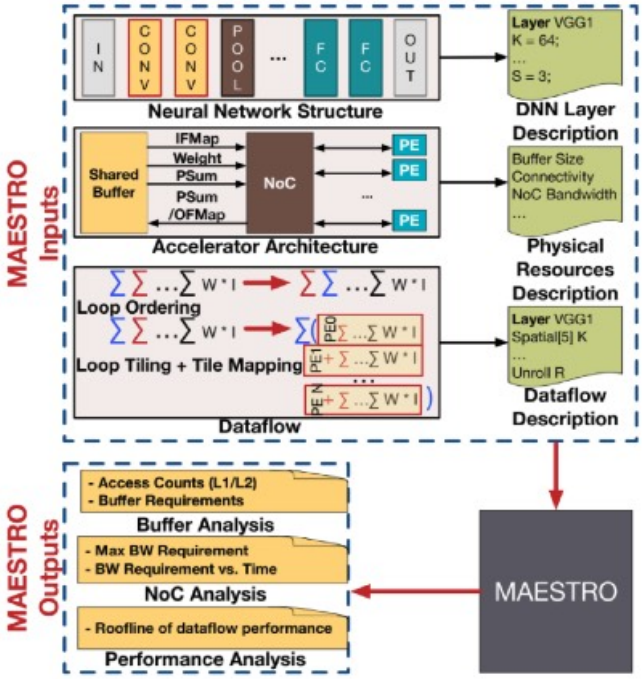


Figure 2.27: MAESTRO Framework Architecture [Kwon et al., 2018a].

Timeloop [Parashar et al., 2019] is a design space exploration framework for CNNs. It can emulate a set of accelerators, such as NVDLA [NVIDIA, 2022a]. Figure 2.28 shows the Timellop framework flow diagram. Timeloop focuses on the convolution layer analyses. Timeloop uses as input a workload description, such as input dimension and weight values, a hardware architecture description, such as arithmetic modules, and hardware constraint. Instead of using a cycle-accurate simulator, Timeloop uses data transfers deterministic behavior to perform analytic analyses. As energy models, Timeloop has memory, arithmetic units, and wire/network models based on TSMC 16nm FinFET.

Accelergy [Wu et al., 2019] allows estimating the energy of accelerators without a complete hardware description, using a library of basic components. Figure 2.29 shows the framework flow. Accelergy uses a high-level architectural description to capture the circuit behavior characteristics, such as memory reads. The obtained results are compared to post-layout results, showing an error of 5% in the energy estimation for the Eyeriss accelerator. Even considering the number of memory reads, Accelergy does not consider relevant

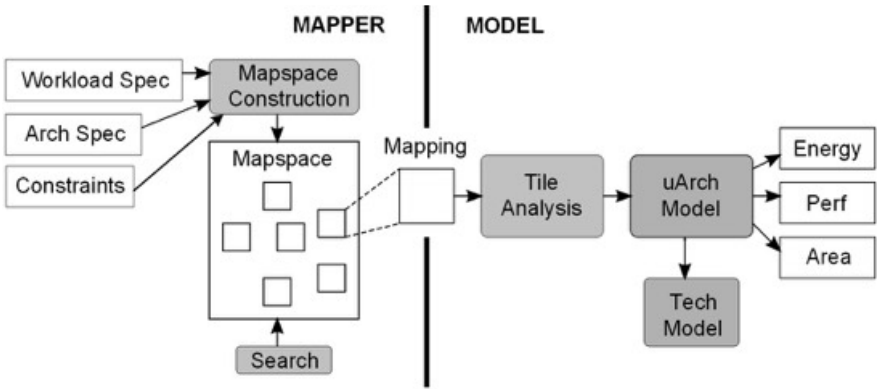


Figure 2.28: Timeloop Framework Diagram Flow [Parashar et al., 2019].

features of an accelerator. Accelergy considers whether the memory access pattern is random or whether it reads the same address repetitively, but it does not take into account dataflow types and data movement through the array. Besides, Accelergy does not provide a simulation environment.

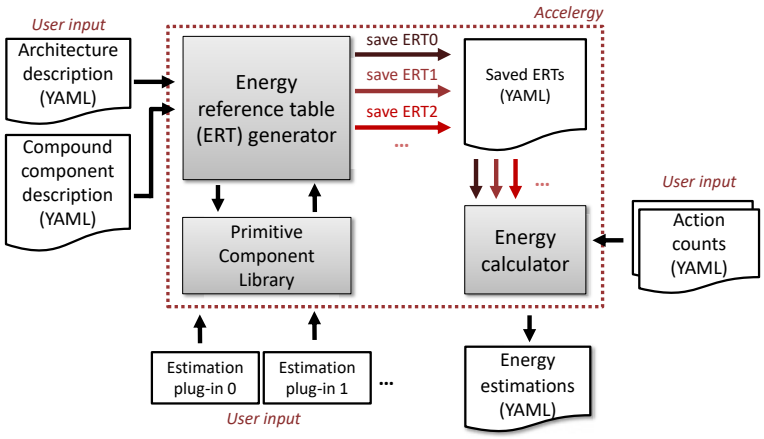


Figure 2.29: Accelergy Framework Diagram Flow [Wu et al., 2019].

Heidorn et al. [Heidorn et al., 2020] propose an analytical model that estimates throughput and energy to a given hardware constraint. A DSE is proposed to determine the accelerator architecture limits in terms of throughput, number of parallel operations, and memory. The Authors propose an accelerator to evaluate the model with a tile-local memory, a bus, and a coarse-grained reconfigurable array (CGRA). Each CGRA presents a two-dimensional array of PEs, and the accelerator can have more than one CGRA to parallelize the processing. Compared to implementations that execute a CNN layer-by-layer sequentially, results show that layer-parallel processing can reduce energy consumption by 3.6 times, hardware cost by 1.2 times, and increase throughput by 6.2 times for a MobileNet.

Zhao et al. [Zhao et al., 2020] propose an analytical performance predictor to estimate energy, throughput, and latency for ASIC and FPGA. Figure 2.30 shows a high-level view of the framework. The predictor uses DNN models, hardware architecture, dataflows types, and hardware cost regarding a technology node. The results are generated with

AlexNet and SkyNet CNN models, with Eyeriss, an FPGA implementation from [Hao et al., 2019], and synthesized results of a proposed accelerator. They show that the error achieves a minimum of 0.25% and a maximum of 17.66% for different CNN models, hardware architectures, and dataflow types.

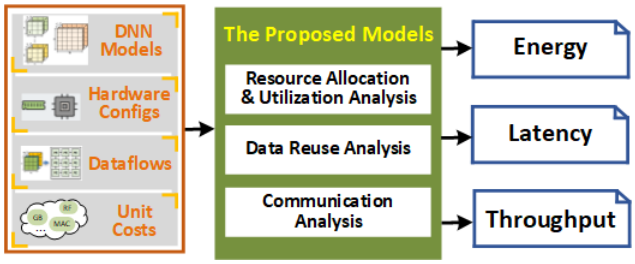


Figure 2.30: DNN predictor high-level architecture [Zhao et al., 2020].

DNNExplorer [Zhang et al., 2021] is a framework for DSE of ML accelerators, presented in Figure 2.31. DNNExplorer supports machine learning frameworks (Caffe and PyTorch), besides three accelerator architectures (named *paradigm* in the Figure).

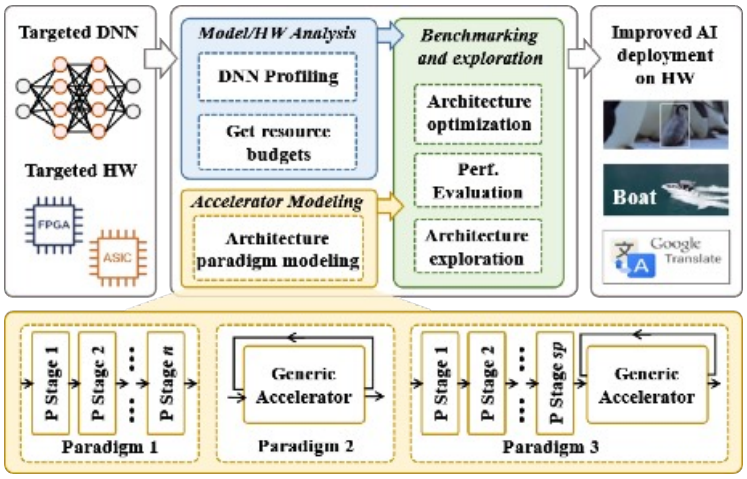


Figure 2.31: DNNExplorer Flow Diagram [Zhang et al., 2021].

The first architecture is based on a pipeline approach, where each stage process a layer of the CNN application. The second architecture is based on a 2D array and can be reconfigured for different layers. The Authors propose the last architecture, a hybrid of the other two architectures. The architecture also supports WS and IS dataflows. The DNNExplorer flow works as follow: (*i*) inform the definition files for DNNExplorer, which include information like layer type, quantization method, and technology (FPGA or ASIC); (*ii*) architecture selection; (*iii*) an optimization step based on the definition files and architecture. This framework adopts analytical models to estimate performance and hardware configuration. Results show that the accelerators proposed by the Authors can increase the throughput 4.2 times (GOPS) compared to the pipeline architecture 2.0 times compared to the second architecture.

Gemmini [Genc et al., 2021] is an open-source systolic array generator that allows evaluating deep-learning architectures. Gemmini generates a custom ASIC accelerator for matrix multiplication based on a systolic array architecture. Gemmini is compatible with the RISC-V Rocket ecosystem [Asanovic et al., 2016]. Figure 2.32 shows the Gemmini general architecture. The CPU component can be either an in-order CPU or an out-of-order CPU. The neural network accelerator comprises a systolic array, allowing both OS and WS dataflow types. A DMA component performs the memory operation and loads the data (feature maps and weight values) to the scratchpad memory. The activation function, such as ReLU, are implemented in hardware. There is also a transposer component, which is a small systolic array used to help in matrix multiplication. The accumulator stores partial results, with a bit width larger than the systolic array. Results are validated using Intel 22nm process technology and show speedups increase between 127 and 2,670 times compared to high-performance CPUs using CNN such as AlexNet, MobileNet, and ResNet50.

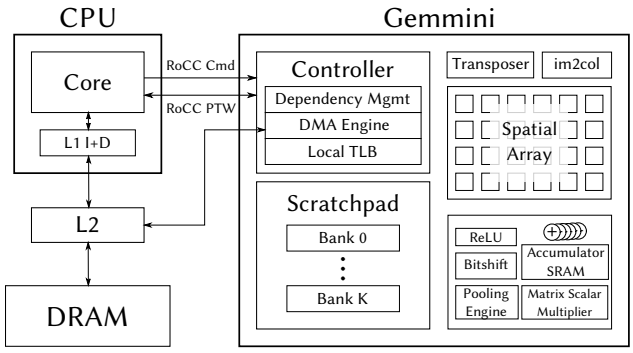


Figure 2.32: Gemmini general architecture [Genc et al., 2021].

Interstellar [Yang et al., 2020] is a DSE framework that uses Halide language (https://halide-lang.org) to generate hardware and compare different accelerators, such as different dataflows (WS, OS, RS) in 2D arrays and a MAC tree schemes. Halide is a DSL for image processing applications and allows mapping a loop-based application into CPUs or GPUs. As CNN applications are also loop-based, it is possible to extend Halide DSL to generate hardware. Thus, the Authors propose a systematic approach to describe the design space of DNN accelerators as schedules of loop transformations. The framework also optimizes the memory hierarchy, and results show a 3.5, 2.7, and 4.2 times energy improvement over Eyeriss accelerator using, VGG-16, GoogleNet, and MobileNet CNNs.

DeepOpt [Manasi and Sapatnekar, 2021] is a DSE framework to explore ASIC implementation of systolic hardware accelerators for CNNs. The main goal of this DSE is to reduce the number of memory accesses based on hardware characteristics like on-chip SRAMs and the number of parallel PEs. The DeepOpt uses a search tree to schedule the convolution process. Thus, it is possible to minimize the number of accesses from memory by modeling memory access patterns (weight and output stationary) and pruning branches

from the search tree. Results show improvements of 50 times in the energy-delay product for VGG-16 and 41 times for GoogleNet-v1.

Karbachevsky et al. [Karbachevsky et al., 2021] propose a method to estimate area and power values based on the bit operations performed (BOP) metric [Baskin et al., 2021]. BOP is the number of bit operations required to perform the calculation, defined by the input bit size, output bit size, number of inputs, and number of outputs. According to the authors, BOP metric allows estimating the area and power required by accelerator hardware with high accuracy in the early stages of the design process. Also, the method can show the trade-off between the number PEs and the bottlenecks caused by the parameters quantization, such as memory bandwidth or computational resources. Two WS dataflow 3x3 MAC arrays were implemented, with multipliers from the Synopsys standard library, synthesized using TSMC 28nm technology at 800MHz. The arrays bit sizes are 4 and 6-bit, and the input dimension has variable sizes of 4, 8, and 16. The output dimension has the same size as the input. Results show that the BOPs achieves a linear relation for the area with an R^2 of 0.9752. The paper does not present power results.

Ferianc et al. [Ferianc et al., 2021] propose a method to improve the performance of DSE analyses. The method is based on a Gaussian process regression model parameterized by the features of the accelerator and the target CNN, such as filter, channel, and data parallelism. Figure 2.33 illustrated the method. The method is capable of predicting the hardware latency and energy. The method was evaluated using two implementations: (*i*) a FPGA implementation using a Intel Arria GX 1150 board; (*ii*) an ASIC implementation using a 28nm technology. The method was compared to machine learning-based methods to perform DSE (linear regression, Gradient tree boosting, and neural network). Results show a reduction between 1.94 and 1.34 times in the prediction time.

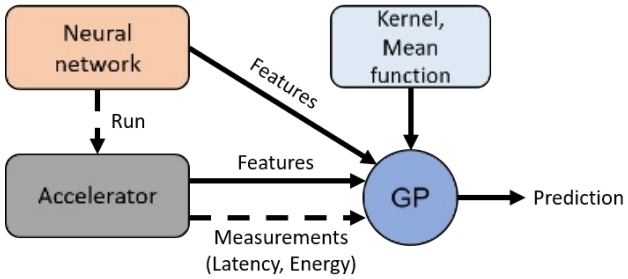


Figure 2.33: DSE Method Based on Gaussian Process Regression Model [Ferianc et al., 2021].

Aladdin [Shao et al., 2014] is a pre-RTL power-performance accelerator modeling framework. It estimates performance, power, and area. Aladdin infrastructure uses dynamic data dependence graphs (DDDG) to represent accelerators. The DDDG is generated from a C program and allows Aladdin to report the program dependencies and resource constraints. Results show that Aladdin has an error of 0.9% for performance evaluation, 4.9% for power evaluation, and 6.6% for area evaluation compared to RTL accelerators implementation.

2.3.2 Hardware Simulators

SCALE-Sim (Systolic CNN Accelerator Simulator) [Samajdar et al., 2018, Samajdar et al., 2020] is a systolic array cycle-accurate simulator. Figure 2.34 illustrates the simulator architecture. This simulator allows configuring micro-architectural features such as array size, array aspect ratio, scratchpad memory size, and dataflow mapping strategy. Also, it is possible to configure system integration parameters, such as memory bandwidth. SCALE-Sim simulates convolutions and matrix multiplications, and models the compute unit as a systolic array. Also, it allows simulation in a system context with CPU and DMA components. The Authors show detailed experiments to understand the design space and tradeoff in designing a systolic array-based CNN accelerator. A recent SCALE-Sim extension provides an analytic model to find the best accelerator configuration based on parameters like DRAM bandwidth.

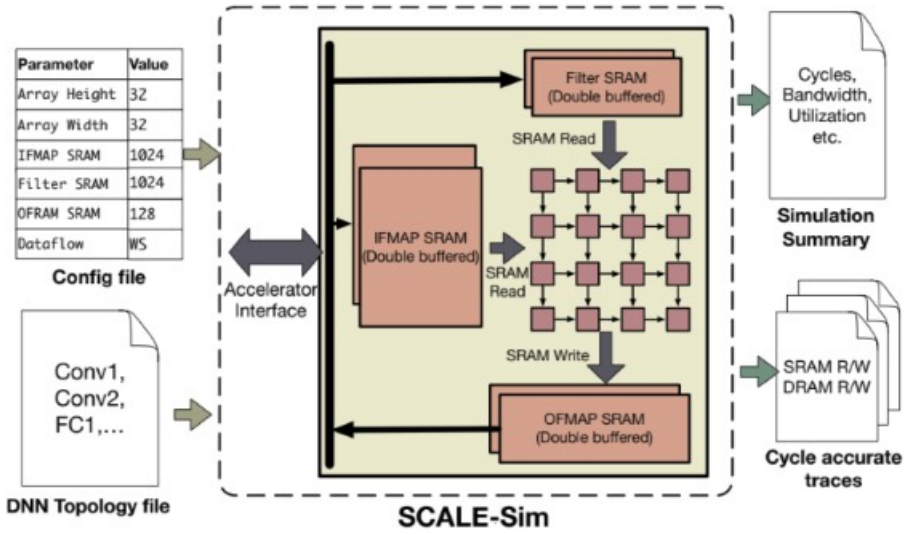


Figure 2.34: SCALE-Sim simulator architecture [Samajdar et al., 2018].

STONNE [Muñoz-Martínez et al., 2020] is a cycle-accurate architecture simulator for CNNs which allows end-to-end evaluation. Figure 2.35 shows the high-level architecture of STONNE framework. It is connected with Caffe framework [Caffe, 2022] to generate the CNNs, and models the MAERI accelerator [Kwon et al., 2018b]. The results are focused on performance and hardware utilization and show an average difference of 15% in total executed cycles than the original MAERI results. To estimate area and energy, STONNE uses the Accelergy energy estimation methodology [Wu et al., 2019], which considers basic modules to calculate the energy values, such as adders.

SimuNN [Cao et al., 2020] is a neural network simulator that allows pre-RTL verification and fast prototyping. Figure 2.36 shows the SimuNN architecture. It is compatible with TensorFlow, allowing using software application values to evaluate the hardware accelerator. The results generated by SimuNN are based on a fixed accelerator proposed

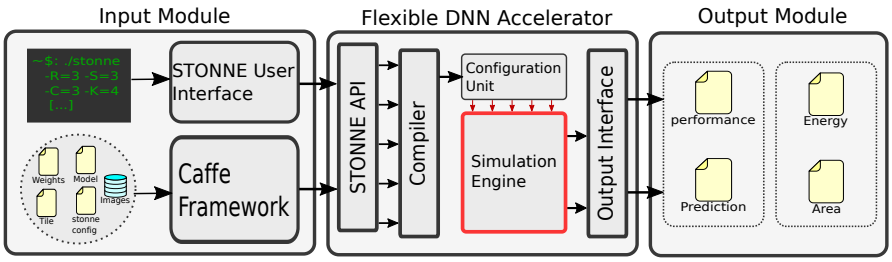


Figure 2.35: STONNE simulator architecture [Muñoz-Martínez et al., 2020].

by the Authors. The accelerator comprises a micro-controller, an instruction RAM, a DDR controller, a weight buffer, a feature map buffer, a feeder controller, a collector unit, and 14 three-stage pipelines PEs with nine multipliers each. The Authors show latency and energy results based on Altera FPGAs and ASICs, although the ASIC technology node is not mentioned.

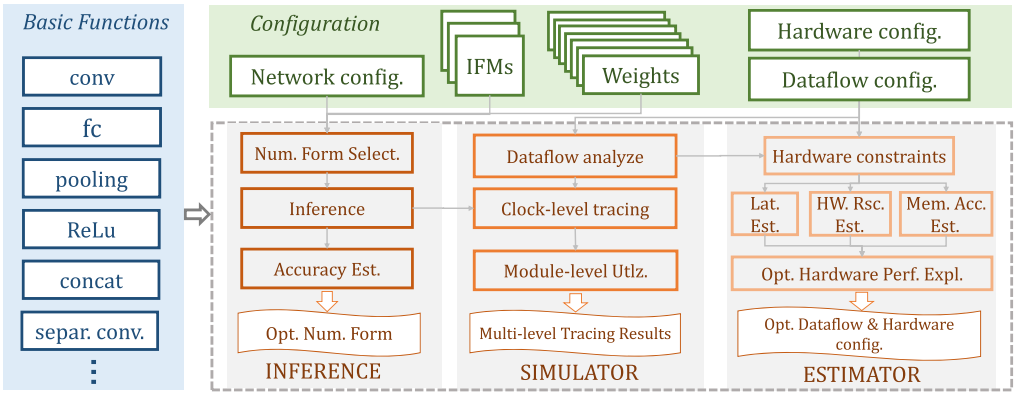


Figure 2.36: SimuNN Simulator Architecture. [Cao et al., 2020]

AccTLMSim [Kim et al., 2020] is a pre-RTL cycle-accurate CNN accelerator simulator based on SystemC transaction-level modeling (TLM). The simulator allows maximizing the throughput performance for a given on-chip SRAM size. An accelerator is proposed to validate the simulator, composed of a MAC array of 12 units, a double buffer scheme to enable memory read and MAC executions in parallel, and a DRAM controller. Each of the hardware blocks is implemented as a SystemC module using sockets, and the accelerator was also prototyped in a Xilinx Zynq FPGA using HLS. AccTLMSim is focused only on performance, not power or area.

2.3.3 Final Remarks Related to DSE Frameworks and Simulators

Previous works present gaps in evaluating CNN's accelerators. Table 2.3 summarizes the reviewed works. The first column represents if the work has integration with high-level modeling CNN frameworks, such as TensorFlow and Caffe. The second column shows

Table 2.3: DSE Frameworks and Simulators State-of-the-art Summary.

Work	Integration with CNN frameworks	High-level Simulation	Evaluation metrics based on basic components	PPA analyses based on entire convolution
Aladdin [Shao et al., 2014]	No	No	PPA	No
MLPAT [Tang and Xie, 2018]	No	No	PPA	No
Maestro [Kwon et al., 2019]	No	No	Performance	No
Timeloop [Parashar et al., 2019]	No	No	PPA	No
Accelergy [Wu et al., 2019]	No	No	Power	No
[Heidorn et al., 2020]	No	No	PPA	No
[Zhao et al., 2020]	No	No	Power	No
Interstellar [Yang et al., 2020]	No	No	PPA	No
SCALE-Sim [Samajdar et al., 2020]	No	Yes	Performance, Area	No
STONNE [Muñoz-Martínez et al., 2020]	Caffe	Yes	Performance	No
SimuNN [Cao et al., 2020]	TensorFlow	Yes	PPA	No
AccTLMSim [Kim et al., 2020]	No	Yes	Performance	No
DNNExplorer [Zhang et al., 2021]	Caffe, PyTorch	No	Performance	No
Gemmini [Genc et al., 2021]	No	No	Performance	No
DeepOpt [Manasi and Sapatnekar, 2021]	No	No	Performance, Power	No
[Karbachevsky et al., 2021]	No	No	Area	No
[Ferianc et al., 2021]	No	No	Performance, Power	No
This Thesis	**TensorFlow**	**Yes**	**No**	**Yes**

if the work provides a simulation environment. The third and fourth columns are related to the evaluated metrics. The third column presents metrics based on basic components, such as MACs and register files. The fourth column shows the evaluated metrics regarding the entire convolution.

MAESTRO does not allow the accelerator simulation, limiting the performance evaluation (e.g., throughput). SCALE-Sim does not provide power or energy results. MLPAT and Timeloop provide PPA based on basic operations, such as adders and multipliers. Methods relying on operations counting do not consider how these operators are interconnected (e.g., 1D or 2D systolic arrays or adder trees), resulting in imprecise hardware metrics.

Works [Heidorn et al., 2020] and [Zhao et al., 2020] show analytical results for power, performance, and area. Also, [Zhao et al., 2020] consider features like the dataflow type, which can contribute to the power dissipation. However, both [Heidorn et al., 2020] and [Zhao et al., 2020] do not support simulation neither integration with CNN frameworks. Similar occurs to Aladdin, once it does not perform simulations. However, it can be integrated into a simulation environment to consider the whole system performance, power, and area analyses [Shao et al., 2016].

Works like Gemmini, Interstellar, DeepOpt, [Karbachevsky et al., 2021], and [Ferianc et al., 2021], lacks on simulation capability and PPA analyses. Also, [Karbachevsky

et al., 2021] claim that the main effect of changing the circuit frequency is to reduce power dissipation. However, it is not true, once logical synthesis with different frequencies as target shows different area values.

STONNE and SimuNN are similar frameworks when compared to our proposal. Both integrate a flow that starts with frameworks to model CNNs, and both provide the accelerator simulation. However, SimuNN uses a fixed 2D array style, not comparing it with other styles like 1D. SimuNN has an energy estimation based on basic elements, not considering data movement through the accelerator. STONNE does not address power estimation, but the authors argue that it is possible to integrate STONNE with Accelergy (which only evaluates power). DNNExplorer also allows frameworks to model CNNs, but lacks simulation and PPA analyses. SCALE-Sim and AccTLMSim lack integration with frameworks to model CNNs.

2.4 Thesis Contribution for the State-of-the-Art

Considering the works analyzed before, some gaps were identified and summarized below:

1. Proposals that allow estimating DSE for different types of accelerators and dataflows;
2. Works that perform complete PPA analyses, not only one metric;
3. A method to estimate PPA accurately;
4. An analytical tool based on the entire convolution mechanism that allows a fast and accurate DSE;
5. A framework or environment that integrates these gaps.

State-of-art shows that few works are capable of estimating PPA metrics. The works that estimate PPA are based only on basic components of an accelerator, which can result in an inaccurate estimation. Few frameworks that allow PPA estimation execute simulation. Thus, this Thesis aims to fill the gaps by providing:

1. A DSE that integrates a CNN modeling framework to perform DSE using data from actual CNNs. The DSE starts with a framework to model CNNs (TensorFlow) and configures hardware accelerators able of executing these CNNs;
2. A simulation environment that uses the values extracted from TensorFlow (Chapter 3). The simulation environment allows to model hardware accelerators with high-level programming languages and performs fast simulations;

3. A physical synthesis flow that allows comparing different dataflow types (Chapters 4 and 5). The physical synthesis flow is the basis for the PPA. Similar to the simulation environment, physical synthesis flow is integrated with TensorFlow to extract accurate power values. Also, the synthesis flow allows performing a fair comparison with different kinds of accelerators, regarding the same technology and target frequency, for example;

4. An analytical tool based on the entire convolution mechanism, allowing a fast and accurate DSE (Chapter 6). This tool is based on a synthesis from a specific layer of a CNN generated in TensorFlow, and estimates other layers based on this synthesis.

3. HIGH-LEVEL MODELING FRAMEWORK FOR DSE

This Chapter presents the first attempt to fill the five gaps identified in the state-of-the-art [Juracy et al., 2021a], presented previously in Section 2.4, by using a high-level modeling framework for DSE. This framework is used in the early design stages and allows high-level validations, with three components:

1. The CNN framework, using the TensorFlow for training – Section 3.1, followed by a quantization method used to reduce the memory requirements – Section 3.2;
2. The physical synthesis of the hardware accelerator for PPA extraction – Section 3.3;
3. The cycle-accurate system simulator, using the URSA simulator – Section 3.4.

Section 3.5 presents results regarding PPA extraction, comparing the netlist simulation against the URSA simulator. Section 3.6 presents pros and cons on using a system simulator to perform DSE for CNN hardware accelerators.

Figure 3.1 presents the proposed framework. TensorFlow models the CNN, responsible for training and inference phases, generating weight and input feature map values. This work adopts an integer quantization to avoid floating-point operations and reduce the memory requirements. The last action executed by TensorFlow is exporting a header file with the weight and feature values used by the system simulator.

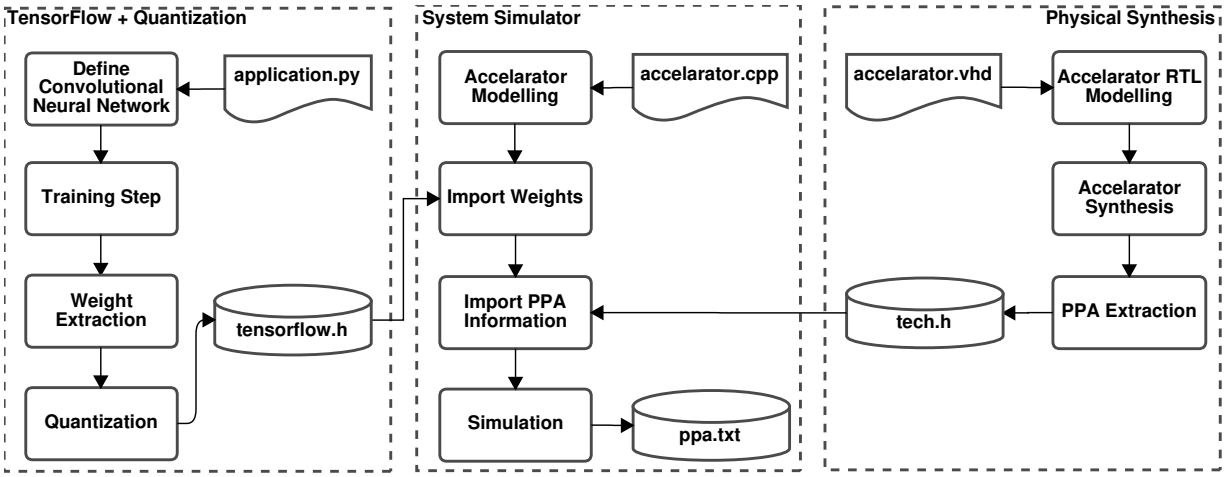


Figure 3.1: Convolution Accelerator Hardware Metric Extraction Framework. Source: [Juracy et al., 2021a]

The physical synthesis corresponds to the synthesis of the CNN accelerators. This step generates the CNN accelerator layout, and a netlist with extracted parasitic capacitances. The simulation of this netlist generates the switching activity, used to characterize the accelerator power dissipation.

The cycle-accurate system simulator [Domingues, 2020] models the hardware accelerator using high-level programming language, the CNN model generated by TensorFlow, and the PPA reports generated by the physical synthesis. The simulator captures information related to the CNN execution, presenting a summary of the accelerator performance, area, and energy results.

3.1 TensorFlow CNN Modeling Framework

This Section describes the use of TensorFlow as a front-end to analyze hardware accelerators. TensorFlow is used to:

1. Model the CNN and exploring its architecture;
2. Extract the weight values of the selected network;
3. Extract network output values to validate post-layout simulation.

Figure 3.2 shows an example of a TensorFlow code, which corresponds to the **application.py** in Figure 3.1. The environment allows exploring CNN architectures and their accuracy regarding the network depth, stride dimension, activation functions, and the number of filters. Thus, it is possible to tune the CNN architecture based on an target accuracy. The example in Figure 3.2 shows a CNN with four convolution layers with 16, 8, 3, and 1 filters, a fully connected layer, and strides with dimensions 2x2 and 1x1.

```
# Clearup everything before running
keras.backend.clear_session()

# Create model
model = keras.models.Sequential()

# Add layers
model.add(keras.layers.Conv2D(16, (3,3), strides=(2, 2), activation='relu',
↪  input_shape=(28, 28, 1)))
model.add(keras.layers.Conv2D(8,  (3,3), strides=(1, 1), activation='relu'))
model.add(keras.layers.Conv2D(3,  (3,3), strides=(2, 2), activation='relu'))
model.add(keras.layers.Conv2D(1,  (3,3), strides=(1, 1), activation='relu'))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation='softmax'))

# Build model and print summary
model.build(input_shape=featureShape)
model.summary()
```

Figure 3.2: TensorFlow Code Example. Source: [Juracy et al., 2021a]

TensorFlow allows extracting the value of weights after reaching the target accuracy in the training phase. After the training phase, a post-extraction quantization occurs. At the end, a header with the weight and feature map values is generated to be imported by the system simulator (**tensorflow.h** in Figure 3.1).

3.2 Shift-based Quantization

The quantization method is performed in python using the weights obtained in the training step with TensorFlow. As the focus of this Thesis is the hardware implementation, we are not concerned with improving or comparing the quantization method with others quantization methods. Nonetheless, this analysis is included in future work.

The quantization goal is to optimize the hardware implementation and reduce memory requirements. Thus, the method converts 32-bit floating-point weights values from TensorFlow to 8-bit integers by multiplying the weight values by a power of two. Therefore it is possible to avoid floating-point arithmetic in the accelerator, which reduces the area and power dissipation. The 8-bit integers are chosen to reduce the hardware area cost. Also, each OFMAP value is divided by the same power of two at the end of each convolution, allowing connecting directly with new hardware and avoiding software interaction.

The quantization has two conditions. The first is if the layer to be processed is the first (**Layer 0**). The first layer can use as input an RGB image, for example. These input values are not quantized, needing to be multiplied by the power of two value. Thus, as the layer to be processed is the first, the convolution together with the quantization process follows Equation 3.1.

$$\begin{aligned}\textbf{Ofmap}[f][x][y] = (\textbf{Bias}[f] * \textbf{ShiftValue}^2) + \\ \sum_{k=0}^{C-1}\sum_{i=0}^{W-1}\sum_{j=0}^{H-1}(\textbf{Ifmap}[k][S_x+i][S_y+j] * \textbf{ShiftValue}) * \textbf{Weights}[f][k][i][j] * \textbf{ShiftValue}))\end{aligned} \tag{3.1}$$

where: f, x and y are the current output channel, the horizontal and the vertical position respectively; C is the total number of input and filter channels, W and H corresponds to the filter size; S is the stride, and **O** is the output. **Ofmap** is the output, **Ifmap** the input, and **Weights** the filter tensors and **Bias** the bias vector; **ShiftValue** is the value used quantization, which is a power of two.

The OFMAP from first layer convolution is quantized, meaning that the IFMAP for the next layer is already quantized. From the second layer to the last convolutional layer, the quantization follows Equation 3.2.

$$\textbf{Ofmap}[f][x][y] = (\textbf{Bias}[f] * \textbf{ShiftValue}^2) + \sum_{k=0}^{C-1}\sum_{i=0}^{W-1}\sum_{j=0}^{H-1}(\textbf{Ifmap}[k][S_x+i][S_y+j] * \textbf{Weights}[f][k][i][j] * \textbf{ShiftValue}) \quad (3.2)$$

Figure 3.3 summarizes the quantization process. First, the values generated in TensorFlow (IFMAP, bias, and weights) are extracted to apply the quantization. Bias and weights are multiplied by the power of two value, while IFMAP is multiplied only if the convolution operation is in the first layer, as shown in Equation 3.1.

At the end of the convolution, each OFMAP value has a magnitude of $ShiftValue^2$. A division by the power of two shift value is applied to change the magnitude for **ShiftValue**, avoiding overflow and a new software intervention, as mentioned before. Thus, the other IFMAPs from CNN layers do not need to be multiplied by the shift value, as shown in Equation 3.2. The convolution process finishes executing the fully-connected layer in software.

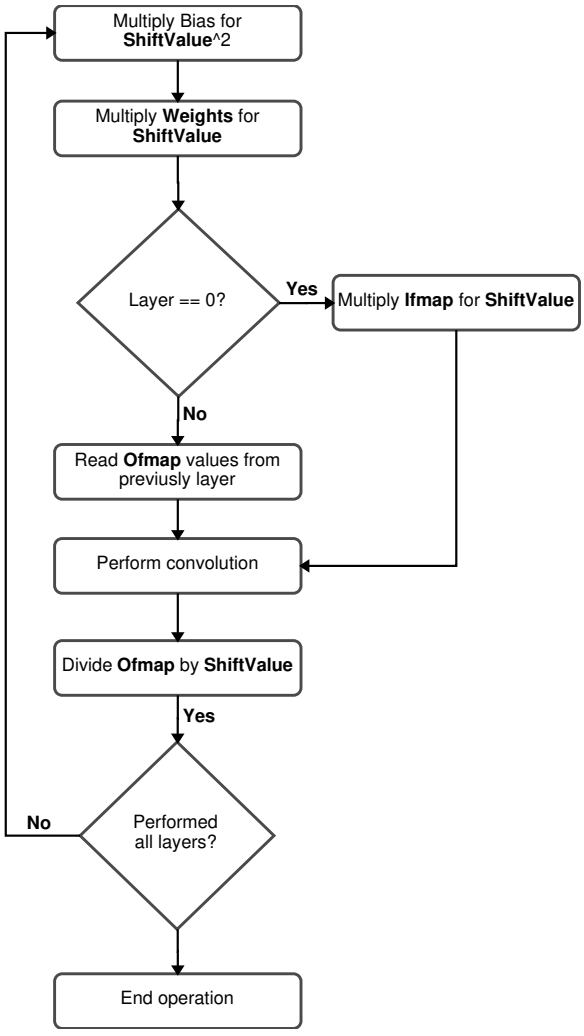


Figure 3.3: Flow diagram of proposed quantization.

3.3 PPA Extraction

The PPA extraction requires the CNN accelerator RTL description once the PPA results are extracted after the physical synthesis. After generating the RTL description, Cadence Genus and Innovus tools are used to execute the logical and physical synthesis. The accelerator area includes gates and wires, and not only cell counting. The simulation of the post-layout netlist generates the accelerator performance (operating frequency) and the switching activity (value change dump file – VCD). The VCD file provides the inputs for dynamic power estimation.

The post-layout simulation generates the switching activity for the power and energy estimation. We adopted two methods to generate VCD files:

1. Method 1: Six VCD files were created from a post-layout simulation. Each VCD represents a simulation scenario that generates one pixel of a convolution output feature map. Scenarios include a real convolution operation, two scenarios with random values, and three scenarios with constant input values (x“AAAA", x“5555", x“FFFF"). The simulation of these scenarios generates minimal, average, and maximum power dissipation values for a convolution operation.

2. Method 2: VCD file generated using the CIFAR10 dataset as input for the post-layout simulation. The CIFAR10 dataset contains RGB images, which ensure fewer zero values than other datasets, like the MNIST, increasing the switching activity of the accelerators.

Method 1 was a start point to extract power values from VCD files and uses synthetic values to generate power values. With the advance of the Thesis, Method 1 evolved to Method 2, which uses real CNN application values to generate VCD files, ensuring accurate power values, once are based on real values.

The PPA metrics are exported to the system simulator in a header format (**tech.h** in Figure 3.1). Section 6.1 details the physical synthesis flow.

3.4 URSA System Simulator

URSA [Domingues, 2020] is a C++ API for system-level modeling and simulation. It provides a set of language-related assets that can be used to create system-level, cycle-accurate hardware simulators, like SystemC. The URSA hardware components are modeled as finite state machines (FSM), and its underlying simulation engine is based on discrete-event simulation. A clock cycle in URSA corresponds to the activation of the transition function of the FSMs of the simulated system. This work uses the URSA to:

1. Model and simulate the CNN;
2. Model and simulate the accelerator;
3. Validate the CNN accuracy;
4. Generate PPA evaluation of the CNN.

Figure 3.4 shows how a CNN is modeled in URSA. One layer of a CNN is simulated in this example, composed of 16 filters with dimension 3x3, strides with dimension 2x2, and the IFMAP with dimension 28x28x1, same parameters of the first convolution layer shown in Figure 3.2. The **TBInit()** function is responsible for performing the memory read, feeding the accelerator, and executing it. When the accelerator is done (signalized by **_array->GetEOP() == 1**), the output value is stored in the **TBStore()** function, which is also responsible for controlling the end of the simulation.

Appendix A details the `TConv2dArray` implementation in URSA. Note that the designer needs to implement a low-level hardware behavior, requiring a second modeling effort. First, the CNN is modeled in TensorFlow for training, and next in URSA for PPA evaluation.

```
void Testbench::TBInit(){
        if (_array->GetEOP() == 0 && _wait_eop == 0 && _end_of_layer1 == 0) {
                TConv2dArray(Layer1_Weights,Input,16,3,3,2,2,28,28,1);
        }
}

void Testbench::TBStore(){
        if(_array->GetEOP() == 1) {
                StoreOfmap(Layer2_Input,Layer1_Bias,&_end_of_layer1,1,16);
        }
}
```

Figure 3.4: URSA Simulator Code Example. Source: [Juracy et al., 2021a]

The CNN application is simulated in URSA using the header files generated in Section 3.1 (**tensorflow.h**), the technology reports generated in Section 3.3 (**tech.h**), and the accelerator array (**accelerator.cpp** in Figure 3.1). Also, the simulator reports the CNN energy and performance estimation when the simulation finishes, according to the number of executed convolutions. Thus, the simulator performs analyses regarding the PPA values extracted from the physical synthesis, and the captured application information at the simulation, resulting in a fast estimation.

3.5 Results

This Section shows results for PPA extraction (Section 3.5.1), energy estimation (Section 3.5.2), and a comparison between netlist simulation and URSA regarding simulation time (Section 3.5.3).

3.5.1 PPA Results

The hardware accelerator used to generate the results is the NVDLA [NVIDIA, 2022a]. As mentioned on Chapter 2, NVDLA is an open-source framework from NVIDIA to implement machine learning applications, providing RTL codes. The NVDLA modules are used for building the accelerator RTL description by configuring multipliers, adders, and ReLU activation function units. Figure 3.5 illustrates an instance of the accelerator array. The accelerator array parameters are a function of the CNN configuration generated in TensorFlow, considering the filter dimension (in this case, a 3x3 convolution).

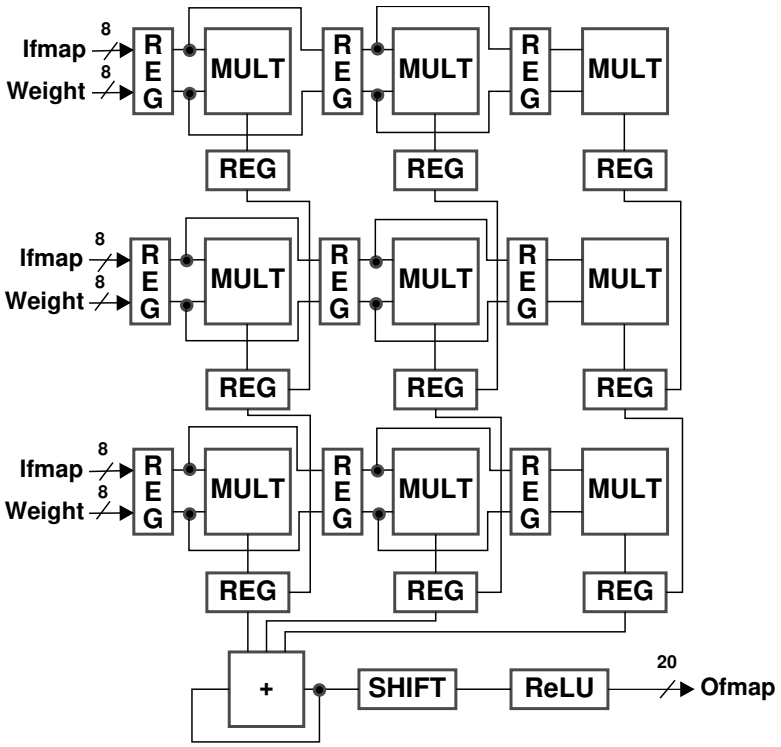


Figure 3.5: Hardware accelerator architecture based on the NVDLA modules. Source: [Juracy et al., 2021a].

The accelerator, Figure 3.5, is an array of 3x3 multipliers (**MULT**), an accumulator, and a ReLU module (**ReLU**). The **SHIFT** is a module used to perform the quantization, in such a way to normalize the convolution result to 8 bits (as described in Section 3.2). The

accelerator inputs are shifted horizontally through the array, while the outputs of the multipliers are shifted vertically until the accumulator. The accumulator output passes through the **SHIFT** and sets the **ReLU** input, which produces a zero value if the input value is negative, else by-pass the value to the output. The system simulator models the accelerator in a high-level abstraction description, using the same architecture.

Results use CNNs generated by TensorFlow to simulate the hardware. Three networks were generated using convolution operations, changing the network depth by 2, 3, and 4 layers, with 4, 12, and 38 filters respectively, to make possible observe the effects in area, accuracy, and energy with the increase of the convolution layers. All three CNNs were trained based on the MNIST dataset using 3x3 filters with strides between 1x1 and 2x2, ReLU as activation function, and a fully-connected layer with softmax activation function. The training step was performed in TensorFlow for 5 epochs. The fully-connected layer is not accelerated in hardware and is executed in software in the system simulator. The VCD file extraction uses Method 1 proposed on Section 3.3.

Table 3.1 presents results for 28nm and 65nm technology nodes. The accuracy was extracted using 100 inputs from MNIST dataset. The energy is estimated using Method 1 presented in Section 3.3. Results show that the quantization causes a small penalty in the hardware accelerator's accuracy compared to the TensorFlow results. Also, the Table shows that accuracy increases together with the CNN depth, which is a expected result. On the other side, the execution time and energy increase.

Table 3.1: PPA results for NVDLA-based accelerator running a MNIST application. Source: [Juracy et al., 2021a].

Tech.	**Freq.** GHz	**Area** μm^2	**# Conv. Layers**	**# Conv. Oper.**	**Accu. (%)**		**Exec. Time (ms)**	**Enegy (mJ)**		
					TensorFlow	**ORCA**		**Min.**	**Avg.**	**Max.**
28nm	1.6	35,003	2	62,800	0.95	0.90	0.5	11	14	17
			3	174,000	0.95	0.92	1.4	32	38	47
			4	375,600	0.96	0.93	3.1	69	84	102
65nm	1.0	97,890	2	62,800	0.95	0.90	0.8	24	31	39
			3	174,000	0.95	0.92	2.4	68	87	109
			4	375,600	0.96	0.93	5.2	147	189	236

Figure 3.6 plots energy values on the x-axis and the CNN accuracy on the y-axis. In this experiment, it is possible to note that it is necessary to spend approximately 40% more energy in the 28nm technology node to increase 0.02% the accuracy. This increase is more pronounced in the 65nm technology node, reaching approximately 60%. This result is correlated with the cell libraries and technology nodes. We suggest extending this analysis to other technology nodes, like 7nm and 10nm, and cell libraries. The result presented in Figure 3.6 shows an advantage of the newer technology nodes: a smaller energy overhead to increase the accuracy.

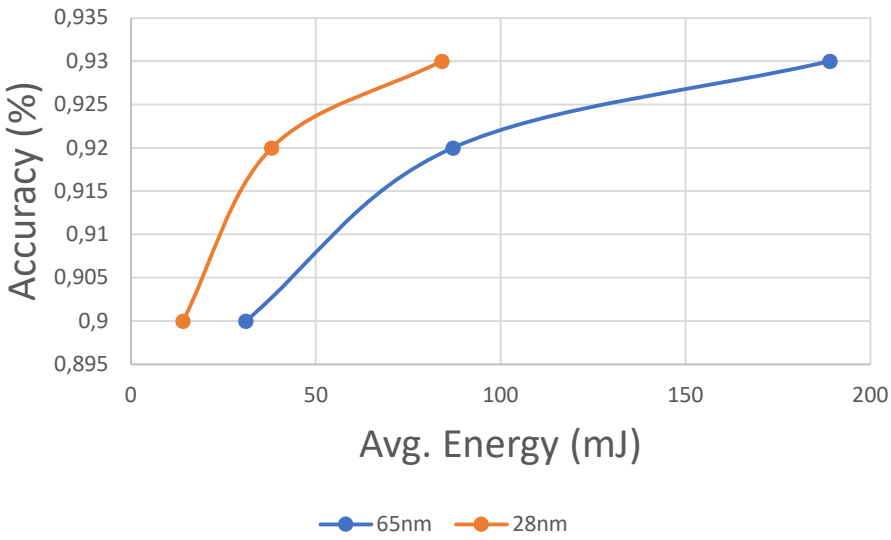


Figure 3.6: Accuracy and Average Energy Trade-off [Juracy et al., 2021a].

3.5.2 Energy Estimation Comparison Results

This Section brings a comparison between the energy estimation based on the netlist simulation, and energy estimation using URSA. The physical synthesis setup in this Section is the same from Section 3.5.3, also using a 3x3 accelerator and 28nm and 65nm technologies. This experiment adopts the second estimation method presented in Section 3.3 due to its accuracy, using one layer from a new CNN that is based in CIFAR10 dataset, and it is used both in simulation and VCD extraction. The netlist simulation input is a 32x32x3 feature map, 16 3x3 filters, stride 2, which generates a 15x15x16 output. The IFMAP, bias and weights are extracted as the same way that **tensorflow.h** in Figure 3.1, but converted to a VHDL package format to be applied in the hardware simulation. The equivalent netlist simulation setup is also used in URSA simulator. Table 3.2 presents the obtained energy for both netlist and URSA.

Table 3.2: Comparison of Estimated Energy of Netlist Simulation and URSA simulator.

Technology	**Energy (μJ)**		
	Netlist	**URSA**	**|Error (%)|**
65nm	603.49	603.73	0.04
28nm	264.45	264.74	0.10

The energy values related to the netlist simulation were obtained from the power estimated by industry-standard EDA (Cadence Voltus) multiplied by the simulation time, considering the simulation with 16 filters. The energy values related to the URSA simulation consider the pre-characterized power obtained from the previous physical synthesis and the number of clock cycles to execute a convolution. The energy estimation using URSA introduced an average error compared to the netlist simulation equal to 0.07% (average value).

This energy estimation error is smaller than, e.g., results presented by Accelergy work [Wu et al., 2019], which is 5%.

However, as Accelergy work, this result are related to one scenario, and other accelerator configuration can result in different estimatives. Also, the obtained energy value does not consider the access memory, which has in important impact in the total energy.

3.5.3 Simulation Time Comparison

This Section compares the time spent in a netlist simulator, and the time spent by URSA. Table 3.3 presents the simulation time for both netlist and URSA.

Table 3.3: Comparison of Netlist and URSA simulator.

Technology	**Simulation Time (sec)**		
	Netlist	**URSA**	**Speedup**
65nm	119.23	1.63	73.14
28nm	128.19	1.67	78.64

The simulation time using URSA is 75 (average value) times faster than the netlist simulation. Such results justify adopting the system simulator to execute a fast design space exploration using physical synthesis data. It is worth mentioning that this experiment simulated a small input feature map with a small number of filters. With the increase of the input feature map size and the number of filters, the speed up using URSA compared to the netlist simulation is expected to increase.

The adoption of the URSA system simulator enables faster simulations with an accurate energy estimation, showing that accurate analyses need to regard the entire convolution accelerator, and not only fundamental components, as adders and multipliers.

3.6 Final Remarks

Works like [Wu et al., 2019] assume that every design evaluation involves a high-level simulation at the architecture level, making the high-level modeling not considered an extra overhead. Thus, to evaluate the hardware accelerators for a given CNN model, we keep TensorFlow framework as a high-level frontend and the RTL description to generate the PPA metrics. The system simulator, URSA, is recommended to simulate a complete computational system.

Thus, lessons learned from a system-level simulator in the context of this Thesis include the following pros and cons:

Pros on using a system simulator:

1. Cycle-accurate simulation;
2. Possibility to describe hardware modules in an abstract way (Appendix A);
3. Generate gold models for circuit verification;
4. Validate the CNN accuracy;
5. The object-oriented approach allows reusing the hardware description, making it easier to build new hardware models;
6. Faster simulation compared to an RTL description;
7. URSA simulator enables entire system simulations once it contains microprocessors and DMA modules.

Cons on using a system simulator:

1. The URSA abstract modeling is complex and may not reflect the actual behavior of the hardware, even though it is cycle-accurate;
2. There is a redundancy of CNN network models: Python modeling for TensorFlow, C++ model for URSA, and VHDL/Verilog for the RTL models. Since the goal is to execute DSE from the physical synthesis of the RTL model, the URSA modeling could become unnecessary if we are not going to simulate the complete system (CNN network, processor, DMA, and memories).

Another limitation of the flow presented in this Chapter is the use of NVDLA for RTL modeling. Despite allowing an extensive parameterization of hardware parameters, NVDLA modules have an important silicon area and limit the exploration of different dataflow architectures.

This Chapter was a first step toward CNNs design space exploration. The TensorFlow and the quantization method continue to be used in the present work. However, in the following chapters (4 and 5), we present the design and evaluation of dedicated accelerators to replace the NVDLA modules. Dedicated accelerators allow configuring array styles (1D/2D) and dataflow architectures (WS, IS, and OS).

The flow for DSE presented in Figure 3.1 is simplified. Chapter 6 details different DSE flows, using the PPA reports obtained by the dedicated accelerators directly by TensorFlow.

4. MACHINE LEARNING HARDWARE ACCELERATOR DESIGN

This Chapter details the design of CNN hardware accelerators, at the RTL level, with the goal to create a rich set of architectures enabling PPA evaluation considering different design choices.

This Chapter contains two sections:

- Section 4.1: presents RTL implementations for array style comparison [Juracy et al., 2021b]. The array style comparison between 1D and systolic 2D array enables to assess trade-offs such as parallelism degree and performance;
- Section 4.2: presents the RTL design for WS, IS, and OS dataflows. It is worth mentioning that the external interface of the accelerators is the same, connecting them to external memories. This feature enables the evaluation of the memory accesses on the accelerator energy.

Chapter 5 evaluates these accelerators, using the same inputs, technology, and target frequency.

4.1 Array Style RTL Implementations

Array style is the PEs organization regarding interconnection in an accelerator [Moolchandani et al., 2021]. In the 1D architecture, PEs are connected sequentially, where each PE has a maximum of two neighbors, with data transferred in a pipeline fashion. A systolic 2D is similar but can have more than two neighbors, arranged as a matrix.

This section presents the design of two accelerator architectures. The first one is a systolic 2D accelerator with two relevant features: memory accesses reduction with high sustained throughput. The second one is a 1D array accelerator to reduce area and power dissipation at the cost of reduced performance. Both accelerators adopt WS dataflow, described in VHDL RTL, validated using the CIFAR10 dataset.

4.1.1 Systolic 2D Accelerator

Figure 4.1 illustrates the systolic 2D accelerator architecture, with its external interfaces and the input memory connection, which stores the bias value, weights, and IFMAP. This memory is assumed pre-loaded before the convolution process, delivering 1 byte per clock cycle. The arithmetic core contains a fixed 3x3 matrix with 3 multipliers, 6 MACs, 3 adders, and 12 registers. The accelerator presents a fetch FSM approach for the feature

reading (FB1/FB2), making it possible to read the memory values and execute the arithmetic process in parallel.

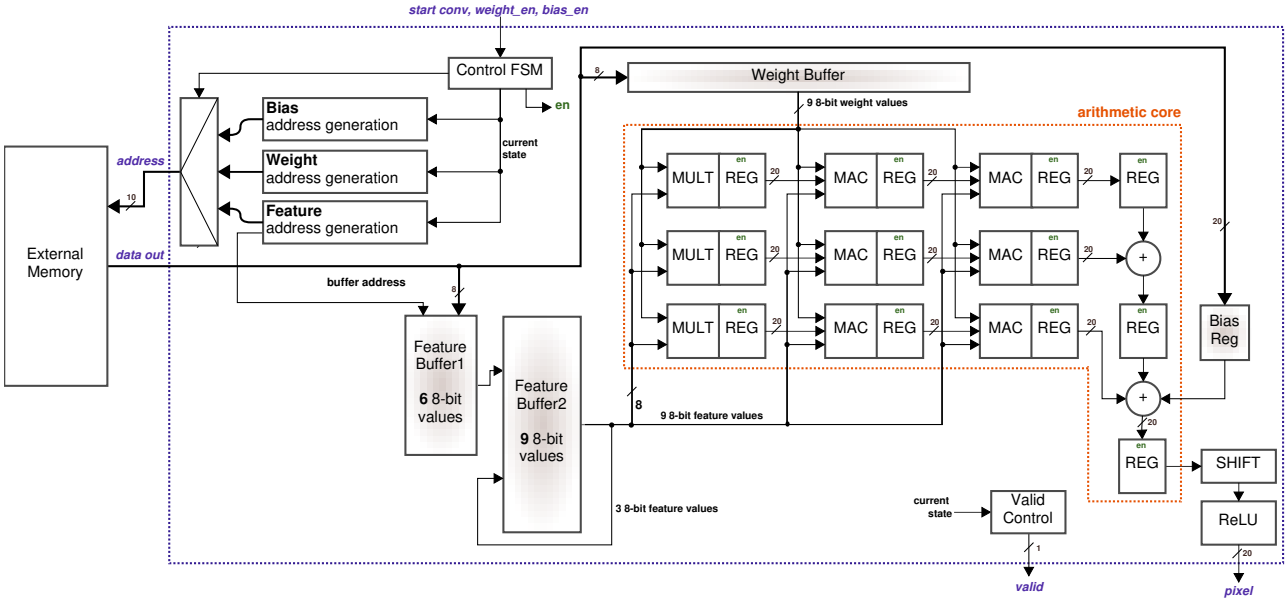


Figure 4.1: Systolic 2D Array Accelerator Architecture. Source: [Juracy et al., 2021b]

The initialization process occurs by loading the weight values (`weight_en`) and the bias value (`bias_en`) in the `weight_buffer` and `bias_reg` buffers. Next, the activation of the `start_conv` signal starts the convolution process. The convolution execution follows a loop controlled by the "Control FSM", until completing the IFMAP reading from the external memory.

This architecture assumes 3x3 weight filters and stride equal to 2. For each convolution, it would be necessary 9 memory accesses. Due to the stride value, it is possible to reuse one column of the 3x3 windows, executing 6 memory readings instead of 9, resulting in a 33% memory reading reduction. The proposed accelerator requires seven clock cycles for data reading and buffering, with the convolution computation executed in parallel to the memory reading:

- cycle 0: transfer the 6 values read from memory from FB1 to FB2, reuse 3 values from FB2, and update FB1 addresses. Given the combinational implementations of the arithmetic blocks (multipliers and adders) in the "arithmetic core", these start computing new values at the end of this cycle.
- cycles 1 to 6: read the IFMAP values from the input memory to FB1.
- cycle 5: at the end of the fifth cycle, the "Control FSM" activates signal `en` for all arithmetic core registers, generating a new output value. This value goes through two combinational blocks, SHIFT and ReLU.

- cycle 6: the “valid control” block activates the `valid` signal according to the convolution being executed. This block controls the bubbles at the end of the generation of the OFMAP line.

After convolution, the accelerator executes the activation function. We adopted ReLU, but other nonlinear functions can be supported, like LeakyReLU and PReLU [Keras, 2022]. The memory of the next convolution layer receives the `pixel` output.

Systolic 2D Accelerator Data Flow

The data movement through the MACs has a wave-front approach to compute up to 5 convolutions in parallel. Figure 4.2 shows the memory addresses related to the IFMAP data at the top, considering an IFMAP with 9 columns and a stride equal to 2. Thus, there are 4 convolutions per line, marked as *conv(a)* to *conv(d)*. The next convolutions correspond to *conv(e)* to *conv(h)*. The weight values reading occurs before the convolution starting, characterizing this approach as a weight stationary.

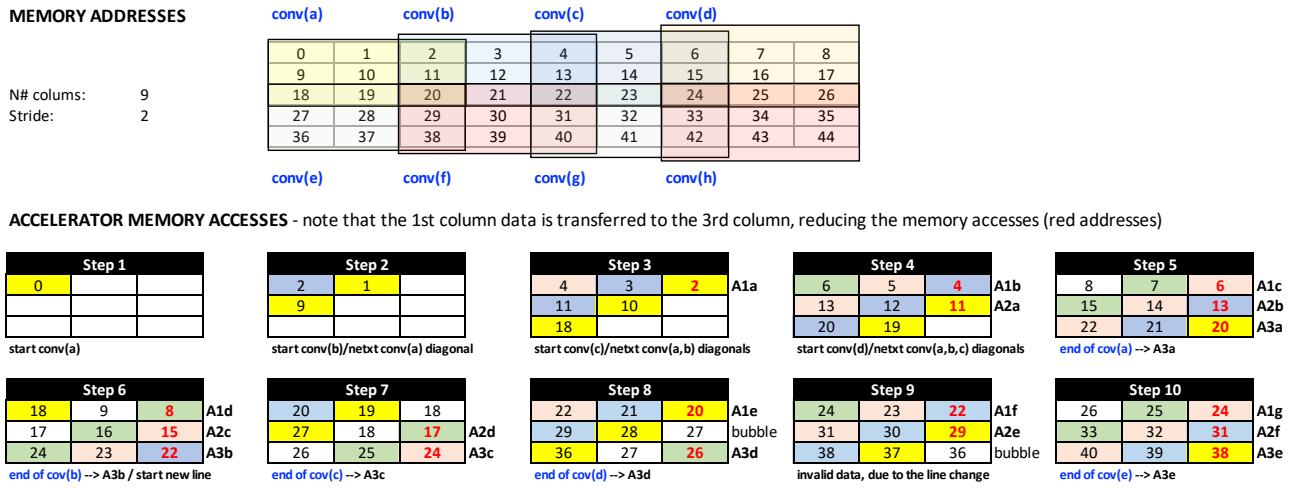


Figure 4.2: Convolution 2D - memory accesses and processing flow. Source: [Juracy et al., 2021b]

The bottom part of the figure has 10 steps. Each step corresponds to the memory reading, and in parallel, the arithmetic operations execution. We use steps 1 to 5 to illustrate a single convolution. At the end of the third step, the computation of the first line (addresses 0/1/2) is stored in a register (**A1a**). At the end of the fourth step, the result of the second line (addresses 9/10/11) is added to **A1a**, stored in a register (**A2a**). At the end of the fifth step, the result of the third line (addresses 18/19/20) is added to **A2a**, stored in a register (**A3a**). The value of this register corresponds to the output of the first convolution.

All steps make 6 memory accesses. The first four steps read values that are not used, corresponding to the filling of the matrix. Take, for example, step 5. In this step, 6 values are read from memory, corresponding to the addresses of the first 2 columns

(8/7/15/14/22/21). The third column is filled with data from the first column (once stride is equal to 2, allowing to perform the reuse scheme), thus reducing memory access (red numbers). Once the matrix is filled, 5 convolutions are processed simultaneously, one on each matrix diagonal.

Note that there is a bubble step between lines (at step 9). This is due to the load of the value in the last row column. The reading process ends in the last column that generates a valid convolution.

4.1.2 1D Accelerator

The second implementation, the 1D array, has a straightforward architecture to reduce area and consumption. Figure 4.3 illustrates the buffers and the arithmetic core of the 1D array. This implementation has three PEs, each with a MAC and a register. The initialization process is the same as the 2D architecture, with the weights and bias load. The generation of a valid output comprises a loop repeated three times, requiring 6 clock cycles at each interaction. The reading of three IFMAP values occurs in the first three clock cycles. In the subsequent three clock cycles, MACs compute new values. At the end of the 6 cycles, registers store values generated by each MAC (signal **en**). At the end of 3 iterations, the MAC registers are reset (signal **res**), and the resulting addition is stored in SUM REG by activating the **load** signal. The throughput is constant without generating bubbles at the end of each line. Note that the systolic 2D requires 7 clock cycles per convolution, while the 1D array 18 clock cycles.

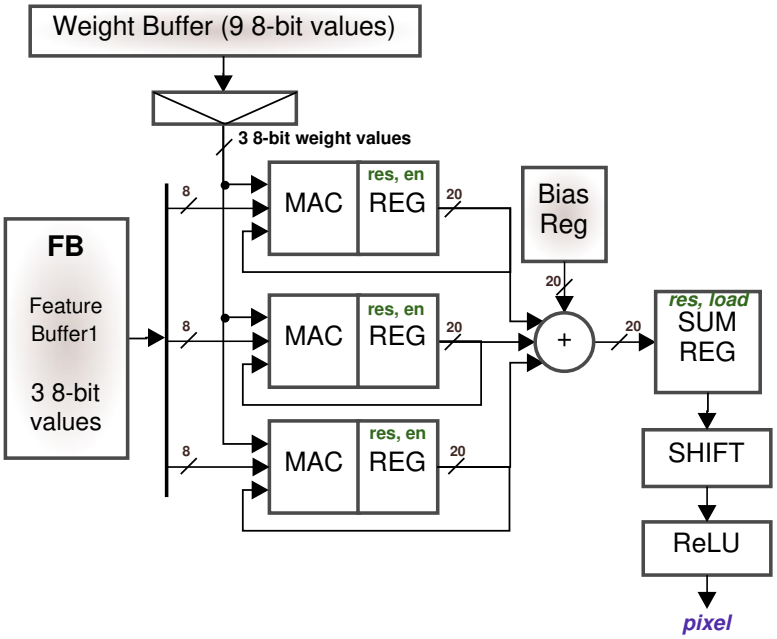


Figure 4.3: 1D Array Accelerator Architecture (buffers and arithmetic core). Source: [Juracy et al., 2021b]

4.2 Dataflow Implementations

The dataflow type refers to how the data to be processed is mapped in a given accelerator array. The mapping determines how to load and generate the data into the array. The dataflow is characterized by latency, throughput, and data reuse parameters.

The previous section described two accelerators focusing on the array style implementation. However, these accelerators abstracted the memory interfaces. This section describes the implementation of three 2D dataflows (WS, IS, and OS), considering a unified interface with input and output memories. External memories play an important role in the total accelerator energy, being mandatory to evaluate this cost.

Figure 4.4 details the 3 modules required to build the convolutional accelerators:

- INPUT memory, stores the IFMAP, filter weights, and bias values. It is a read only memory;
- convolutional core, executes the convolution;
- OFMAP memory stores the partial and complete convolution values.

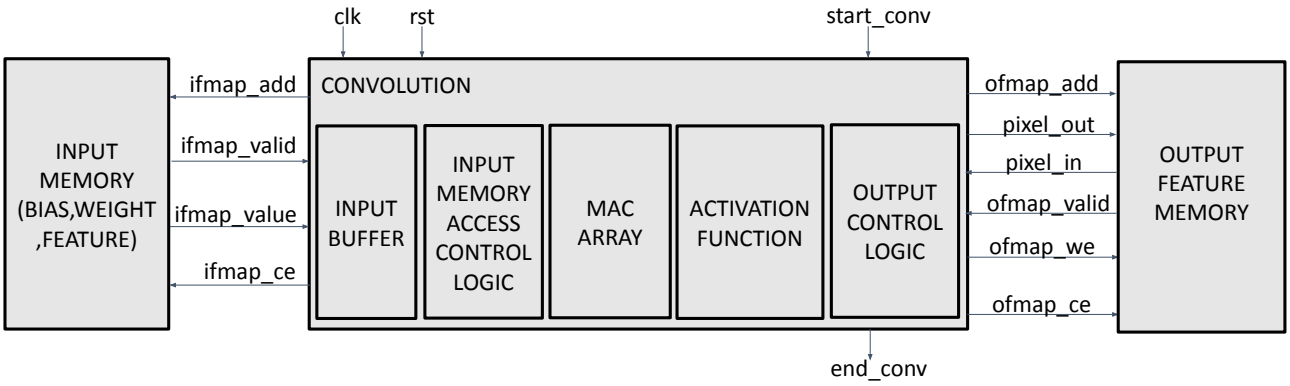


Figure 4.4: Generic architecture and the modules required to build the convolutional accelerators.

The convolutional core contains:

- input buffer: reduces the number of input memory readings. According to the accelerator type, this buffer may store, e.g., an input channel, a set of rows of the input channel, or a set of weights;
- input memory access control logic: control the input memory access. It is implemented using an FSM based on the dataflow type.
- MAC array: a matrix with multipliers, adders, and accumulators responsible for executing the convolution;

- activation function: a non-linear function applied to the OFMAP results. Examples are sigmoid, ReLU, leaky ReLU [Keras, 2022]. This works adopts the ReLU function ($max(0, x)$) due to its simpler hardware implementation;
- output control logic: control the OFMAP memory access. It can be implemented with buffers to reduce memory access.

The following signals control the memory access:

- `ifmap_add` and `ofmap_add`: IFMAP and OFMAP memory address;
- `ifmap_valid` and `ofmap_valid`: these signals are related to the memory latency. The signal indicates when a data from memory is ready to be consumed;
- `ifmap_ce` and `ofmap_ce`: memories chip enable, used to control the memory access;
- `ifmap_value`: data that come from IFMAP memory. `ifmap_valid` indicates when this data is ready to be consumed;
- `pixel_in`: data that come from OFMAP memory. `ofmap_valid` indicates when this data is ready to be consumed;
- `pixel_out`: data that is stored in the OFMAP memory. It is used to store partial sum values generated in the convolution operation;
- `start_conv`: input signal used to start the convolution operation;
- `end_conv`: output signal that indicates the end of the entire convolution operation.

The unified memory interface makes it possible to implement different dataflows with distinct protocols, allowing a fair comparison between the accelerators. This Thesis covers the WS, IS, and OS dataflow, once they are the most common approaches in state-of-the-art, as shown in Chapter 2.

All accelerators adopt a 3x3 MAC array, and the convolution stride equals 2. Despite being a design limitation, state-of-the-art CNNs adopt these values, such as VGG16 [Simonyan and Zisserman, 2014], ensuring that the proposed accelerators reflect real CNNs.

4.2.1 Weight Stationary (WS) Dataflow

Algorithm 1 presents the pseudo-code describing the WS hardware behavior. The core of the algorithm comprises lines 3 to 9. Lines 3-4 fetch a filter set $w(f, c)$ from memory, storing it in the input buffer. The loop between lines 5-9 reads windows from the IFMAP, executes the convolution, and produces a partial result. To obtain a convolution value in

the OFMAP memory (output memory), it is necessary $F - 1$ reads (for partial convolution values), and F writes (line 8). This procedure implies a large number of memory accesses, increasing the total energy consumption.

Algorithm 1: WS pseudo-code.

```
   Input:  C input channels, F output channels
   Output:  O
 1 foreach f in F do
 2     foreach c in C do
 3         Read weight filter set w(f,c) from input memory
 4         Store filter set in the input buffer // weight stationary
 5         foreach I(i)(j) in IFMAP(c) do
 6             Read a window I(i)(j) from IFMAP
 7             p ← convolution(I(i)(j), w(f, c))
 8             O[f][x][y] ← O[f][x][y] + p
 9         end
10     end
11 end
```

Figure 4.5 shows the hardware architecture. The dashed blue square indicates where the stationary values are stored. This accelerator version also uses the same double-buffer scheme of the 2D accelerator (Section 4.1) to allow parallelizing of the convolution process. Also, the wave-front approach presented in Section 4.1 is the same.

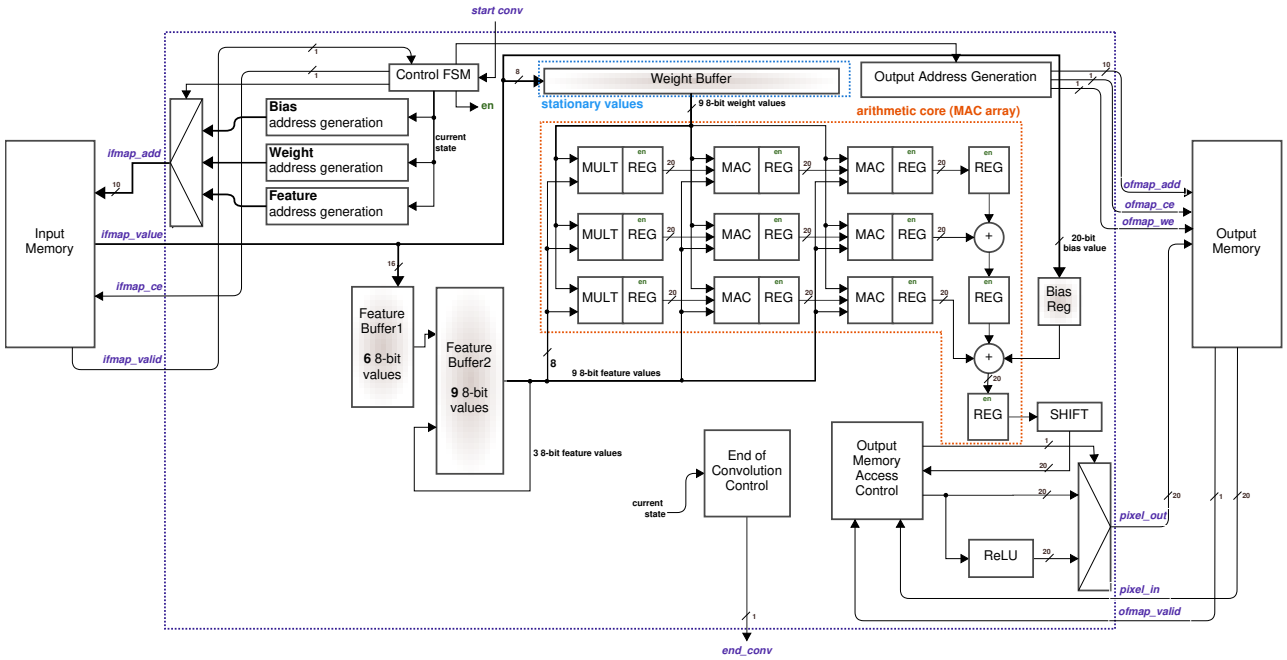


Figure 4.5: WS 2D accelerator and memory interfaces.

To control each dataflow, a specific FSM is designed. Figure 4.6 shows the WS Control FSM. The FSM works as follows:

1. **WAIT START**: the FSM waits the `start_conv` signal to rise;

2. **READ BIAS**: the accelerator reads a value from the IFMAP memory, and wait for the `ifmap_valid` signal;

3. **READ WEIGHT**: after reading the bias value, the accelerator starts to read the weight values, and wait for the valid signal to each value (9 in this case). This state corresponds to the stationary values (line 3 and 4 in Algorithm 1);

4. **START MAC**: allow the MACs to start convolution of a given channel;

5. **WAIT CONV**: wait for the execution of the convolutions in a given channel. In WS, `N_CONVS` is related to the partial values of an OFMAP, which are processed according to the input channel size (loop at lines 5–9 in Algorithm 1). After executing `N_CONVS`, a new bias value is read, which means a filter change in the convolution operation. When the accelerator reads all filters, the convolution ends and returns to the **WAIT START** state.

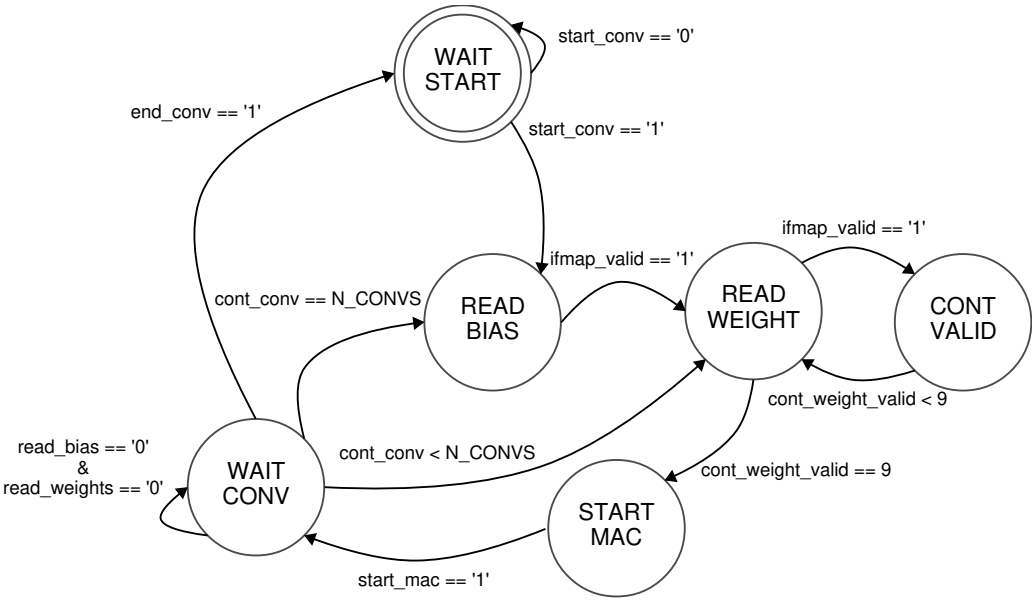


Figure 4.6: WS accelerator Control FSM.

The Control FSM remains in the **WAIT CONV** state during the computation of a given channel. The signal `start_mac` is the trigger to a second FSM, "fetch FSM", responsible for computing the convolutions and fetching the IFMAP values.

The Fetch FSM, detailed in Figure 4.7, executes the core of Algorithm 1, i.e., the loop between lines 5–9. After the rise of the `start_mac` signal, the accelerator starts to fetch IFMAP values from memory. States **FECTH IFMAP** and **CONT VALID** read IFMAP values, compute the partial convolution value, storing it in the OFMAP memory or a buffer. Next, the state **UPDATE_ADD** generates a new IFMAP address up to the end of the channel computation.

At the end of a channel computation it is necessary to read a new bias value or a new set of weights (signals `read_bias` and `read_weights`). In this case, the Fetch FSM returns to the **IDLE** state, releasing the Control FSM. Figure 4.7 also shows the reuse scheme based on the stride. The Fetch FSM reads 6 IFMAP values for each convolution operation instead of 9, which is the IFMAP window used in the convolutions.

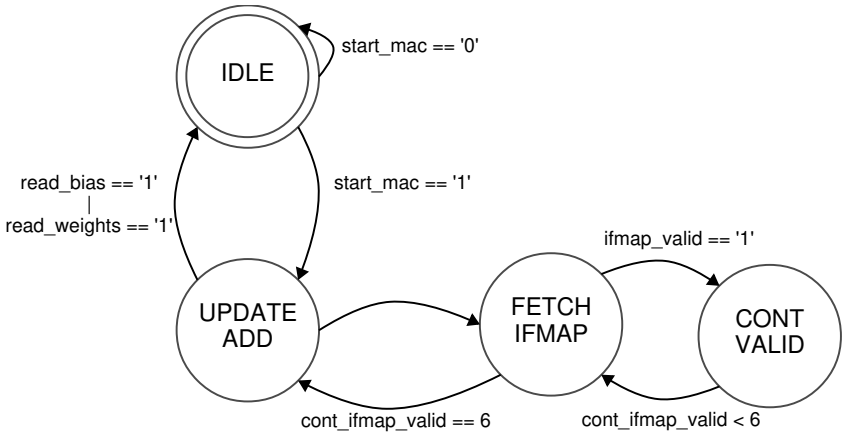


Figure 4.7: WS accelerator Fetch FSM.

The writing in the OFMAP memory (line 8 in Algorithm 1) has two approaches. The first one, detailed in Figure 4.5, uses the OFMAP memory to store partial values of a convolution generated by the MACs (line 7 in Algorithm 1). For example, a CNN with an RGB image input (3 input channels) needs to perform 2 reads (one for R channel values and one for G channel values). Also, three writes are necessary (one for R channel values, one for G channel values, and the last for the final convolution value). This operation is represented by the **Output Memory Access Control** module in Figure 4.5.

The second approach, detailed in Figure 4.8, uses an output buffer, reducing the OFMAP memory accesses. It is similar to the first approach, but the OFMAP memory is not used to store the partial values. Instead, an internal buffer with the OFMAP size stores partial values. For example, a convolution of a 32x32x3 RGB image with a 3x3 filter and stride 2 generates a 15x15 OFMAP. Thus, the output buffer also has 15x15 positions. Therefore, it is possible to eliminate the OFMAP memory reads and write only the final convolution value. However, this solution increases the accelerator area and energy.

4.2.2 Input Stationary (IS) Dataflow

Algorithm 2 presents the pseudo-code describing the IS hardware behavior. The core of the algorithm comprises lines 3 to 14. Lines 4-5 fetch an IFMAP window $I(i)(j)$ from memory, storing it in the input buffer. The loop between lines 7-12 reads filter sets from the input buffer, executes the convolution, and produces a partial result.

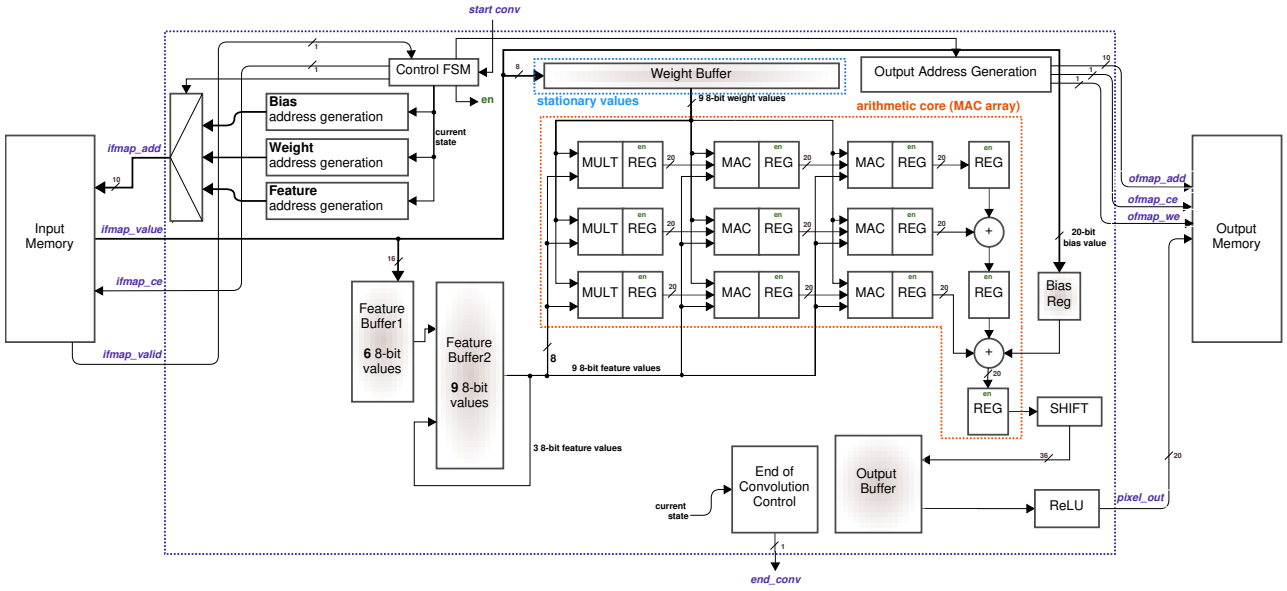


Figure 4.8: Buffered WS 2D accelerator and memory interfaces. For this version, the output buffer replacing the output memory control logic is what differentiates this architecture from the WS.

Algorithm 2: IS pseudo-code.

```
   Input: C input channels, F output channels
   Output: O
1  Read weights and bias, storing in the input buffer // IS optimization
2  foreach c in C do
3      foreach I(i)(j) in IFMAP(c) do
4          Read a window I(i)(j) from IFMAP
5          Store window I(i)(j) in the input buffer // input stationary
6          foreach f in F do
7              foreach cw in C do
8                  foreach w in w(f,cw) do
                       // stored in the input buffer
9                      p ← convolution(I(i)(j), w(f, c))
10                     O[f][x][y] ← O[f][x][y] + p
11                 end
12             end
13         end
14     end
15 end
```

Figure 4.9 shows the IS hardware architecture. The dashed blue square indicates where the stationary values are stored (*Feature Buffer*). Unlike WS dataflow hardware, the data move through the array in a pipeline fashion. The IS dataflow has a different approach for reducing the input memory access. Differently from a standard IS, the implemented IS stores weights and bias values in the input buffer (similar to a cache memory) to reduce the memory accesses while introducing a penalty in the area (line 1 of Algorithm 2). We adopted this approach once the data required for weights and bias is smaller than an IFMAP channel. For example, considering a 32x32x3 convolution layer with 16 3x3x3 filters, the input (3072 values) is larger than the number of weights and bias (448 values).

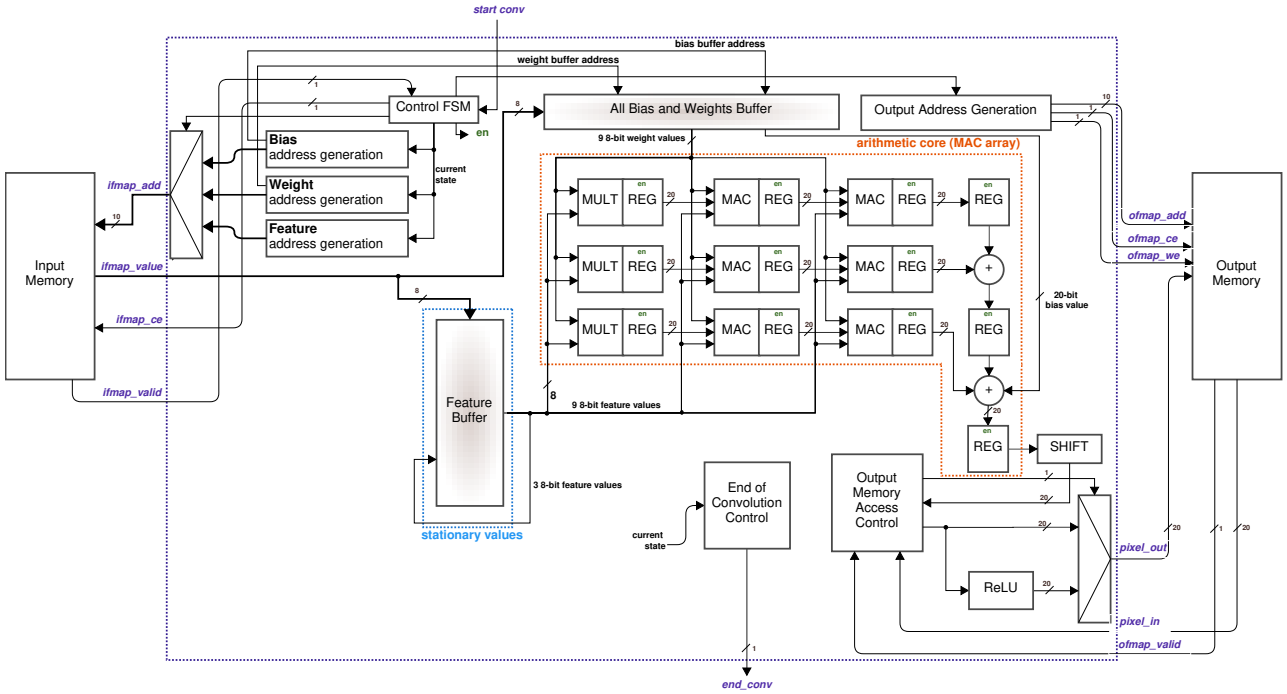


Figure 4.9: IS 2D Array accelerator and memory interfaces. IS version has no double buffer, and has a register bank to store all bias and weights values internally in the accelerator.

Figure 4.10 shows the Control FSM for the IS protocol. The protocol works as follow:

1. **WAIT START**: the FSM waits the `start_conv` to rise;
2. **READ BIAS**: the accelerator reads the bias values from the input memory and waits for the `mem_valid` signal. The bias values are stored in the internal buffer;
3. **READ WEIGHT**: the accelerator reads the weight values and waits for the `mem_valid` signal. The weight values are also stored in the internal buffer;
4. **READ IFMAP**: after reading all bias and weights values, the accelerator starts to read the IFMAP values, and wait for `mem_valid` signal (9 values in this case). This state corresponds to the read of the stationary values (lines 4 and 5 in Algorithm 2).
5. **START MAC**: allow the MACs to start convolution for a given filter channel;
6. **WAIT CONV**: wait for the convolutions to finish. In IS case, the partial values are related to the filter channel and the OFMAP channel line (loop at lines 8–11 in Algorithm 2). After executing `N_CONVS`, a new IFMAP window is read (loop at lines 3–14 in Algorithm 2). The convolution ends when the accelerator reads all IFMAPs and returns to the FSM initial state.

The Control FSM remains in the **WAIT CONV** state during the computation of a given filter channel. The signal `start_mac` is the trigger to a second FSM, “load FSM”,

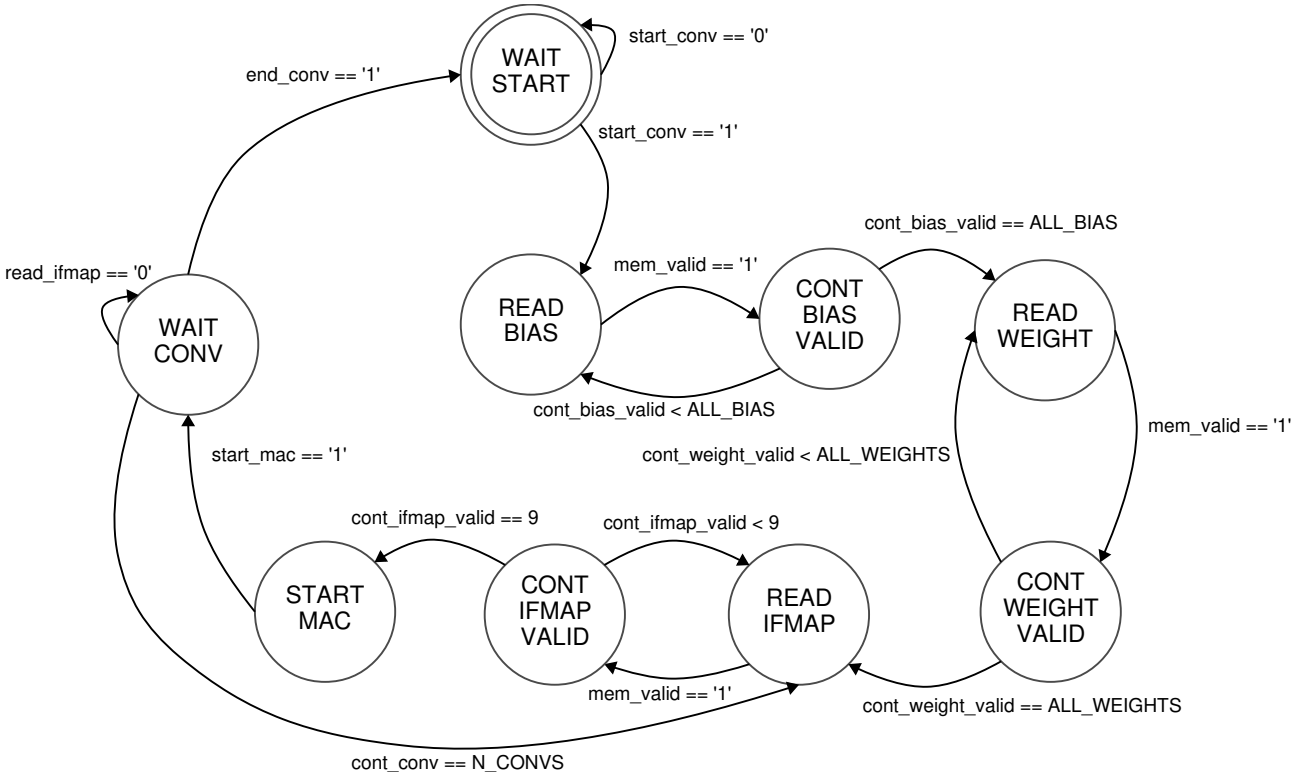


Figure 4.10: IS accelerator Control FSM.

responsible for computing convolution and loading the weight values from the internal buffer. Unlike WS accelerators, the accelerator performs convolution in parallel with the read of the weights from the input buffer, which works similarly to a cache memory.

The Load FSM, detailed in Figure 4.11, executes the core of the Algorithm 2, i.e., the loop between lines 6–13. After the rise of the `star_conv` signal, the accelerator starts to read the weight values from the buffer. Note that in IS is not necessary to wait for the `mem_valid` signal, which improves both performance and energy consumption. At the end of the filter channel computation, it is necessary to read a new IFMAP window (signal `read_ifmap`). In this case, the Load FSM returns to the **IDLE** state, releasing the Control FSM. Figure 4.11 also shows that IS reads 9 weight values per convolution.

The IS also has two versions, with and without output buffer. The IS OFMAP access (line 9 of Algorithm 2) is the same as WS. However, the size of the output buffer is different. Once IS generates a line of the OFMAP channels per time, the output buffer dimension is based on the width of the OFMAP and in the number of filters (output channels). For example, a convolution of a 32x32x3 RGB image with a 3x3 filters, stride 2, 16 output channels, generates 16 15x15 OFMAPs. In this IS case, the output buffer has 16x15 positions. Both output buffers have similar sizes for small OFMAPs, such as 15x15x16 OFMAPs (WS: 15×15 and IS: 15×16). For a larger OFMAP size, there is a clear advantage for the IS output buffer (e.g., 128x128x16: WS: 128×128, IS: 128×16). Figure 4.12 illustrates the buffered version of IS.

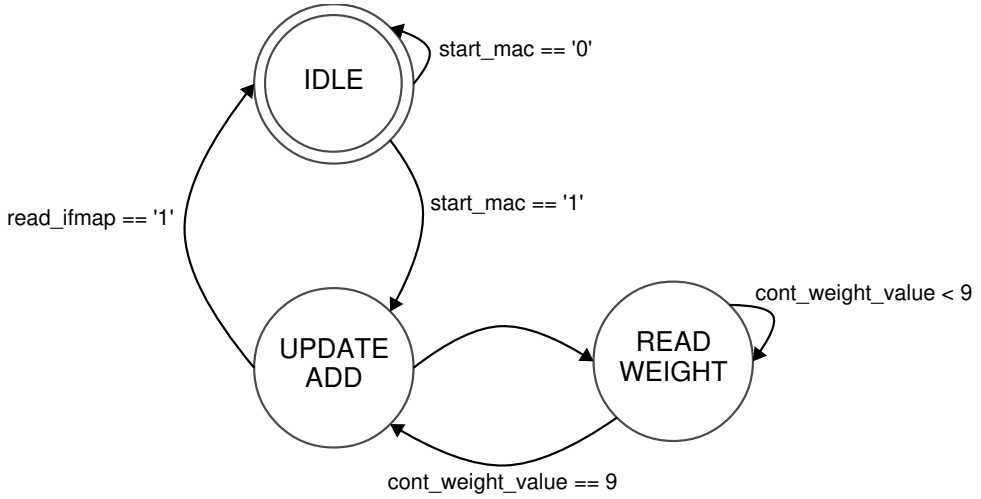


Figure 4.11: IS accelerator Load FSM.

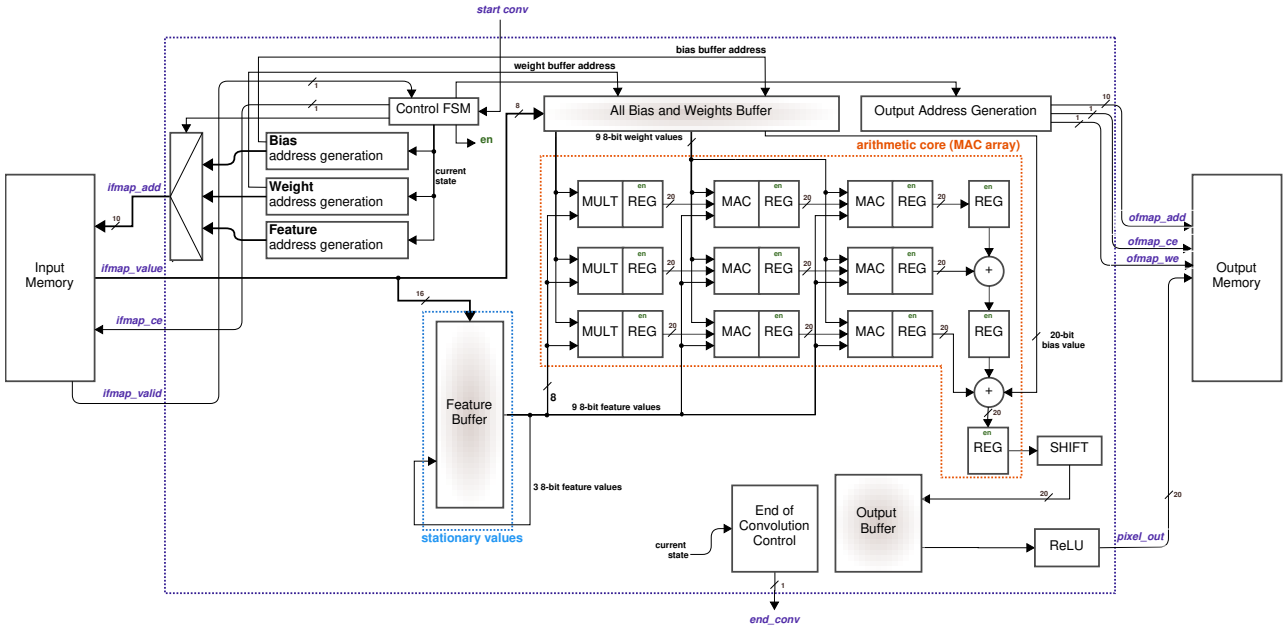


Figure 4.12: Buffered IS 2D Array accelerator and memory interfaces. For this version, the output buffer replacing the output memory control logic is what differentiates this architecture from the IS. Also, like IS, Buffered IS version has no double buffer, and has a register bank to store all bias and weights values internally in the accelerator.

4.2.3 Output Stationary (OS) Dataflow

Algorithm 3 shows the pseudo-code describing the OS hardware behavior. The core of the algorithm comprises lines 4 to 9. Lines 5 and 6 fetch a IFMAP window $I(i)(j)$ and a filter set $w(f, c)$ from memory. Lines 7 and 8 perform the convolution and accumulate the partial result in the output buffer *internal_p*. Line 11 stores a complete convolution value.

Figure 4.13 shows the hardware architecture. The dashed blue square indicates where the stationary values are stored. Like IS dataflow hardware, the data move through

Algorithm 3: OS pseudo-code.

```
Input: C input channels, F output channels
Output: O
foreach f in F do
    foreach c in C do
        internal_p ← 0
        foreach I(i)(j) in IFMAP(c) do
            Read a window I(i)(j) from IFMAP Input Memory
            Read a set of filters w(f)(c) from Input Memory
            p ← convolution(I(i)(j), w(f, c))
            internal_p ← internal_p + p
        end
    end
    O[f][x][y] ← internal_p // output stationary
end
```

the array in a pipeline fashion. OS is the dataflow that performs more memory access once there is no buffer to reuse weights or IFMAPs, and each convolution requires memory access. The stationary values, in this case, are the partial sums generated by the convolution.

The OS accelerator also executes computation and memory accesses in parallel, enabled by a double buffer scheme. Please refer to “feature buffer 1”-“feature buffer 2” and “weight buffer 1”-“weight buffer 2” in Figure 4.13.

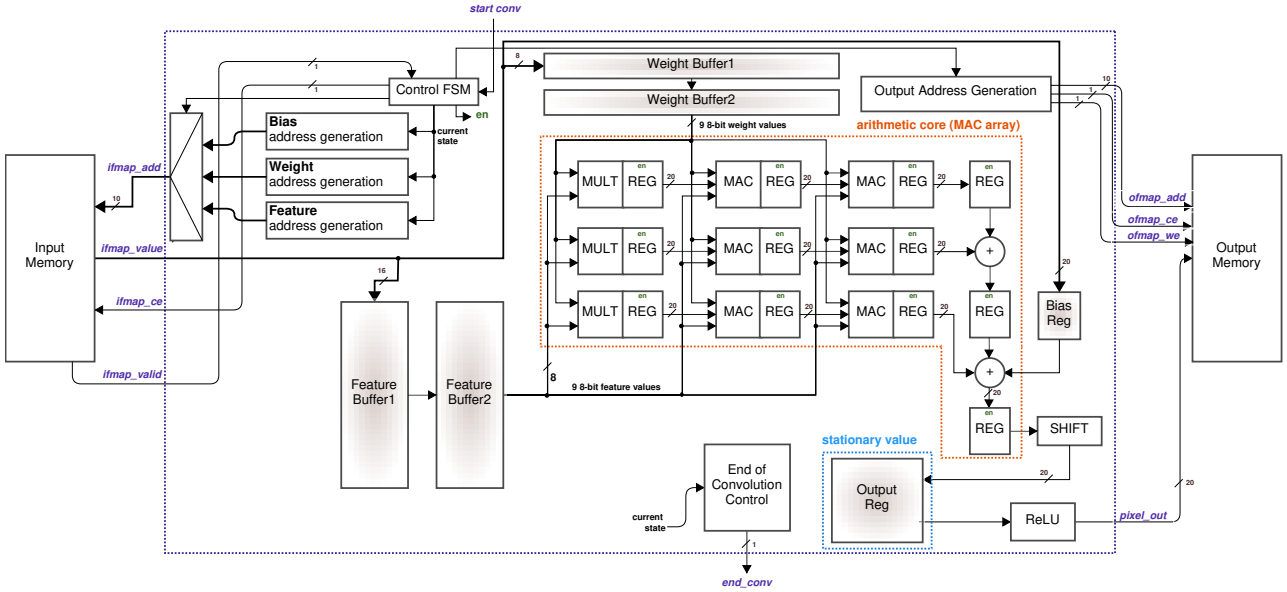


Figure 4.13: OS 2D Array Accelerator and memory interfaces.OS has a double-buffer scheme similar to WS, but instead, it has one for IFMAPs, and one for weights.

Figure 4.14 shows the OS Control FSM. The protocol works as follow:

1. **WAIT START**: the FSM waits the `start_conv` to rise;
2. **READ BIAS**: the accelerator reads a bias value from the IFMAP memory and wait the `mem_valid` signal;

3. **START MAC**: allow the MACs to start convolution for a given IFMAP and weight channel;

4. **WAIT CONV**: wait for the convolutions to finish. In OS case, the partial values are related to the IFMAP and filter channels (loop at lines 1–10). After executing `N_CONVS` (equal to the number of output channels), a new bias value is fetch from memory, and the filter changes. When the accelerator generate all OFMAP values, the convolution ends and returns to the FSM initial state.

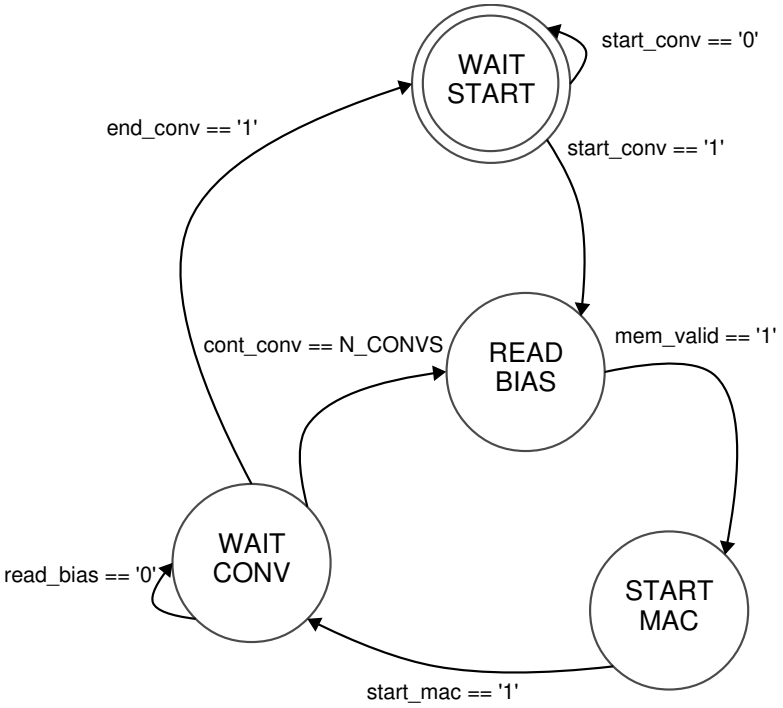


Figure 4.14: OS accelerator Control FSM.

The Control FSM remains in the **WAIT CONV** state during the computation of a given OFMAP value. The `start_mac` is a trigger to the Fetch FSM, detailed in Figure 4.15, responsible for computing the convolutions and fetch the IFMAP and weight values (using a double-buffer scheme similar to the WS).

The Fetch FSM executes the core of the Algorithm 3, i.e., the loop between lines 2–10. After the rise of the `start_conv` signal, the accelerator starts to read the IFMAP values from memory. After reading all necessary IFMAP values (9 in this case), the accelerator starts to read the weight values from memory (reading more 9 values), totalizing 18 memory reads. Note that, like WS dataflow, each memory fetch waits for the `mem_valid` signal to allow a new memory read, which reduces the throughput.

At the end of the OFMAP computation, it is necessary to read a new bias value (signal `read_bias`). In this case, the Load FSM returns to the **IDLE** state, releasing the Control FSM. OS dataflow has only one approach, once it always requires an output buffer to store the partial values of the convolution (stationary values).

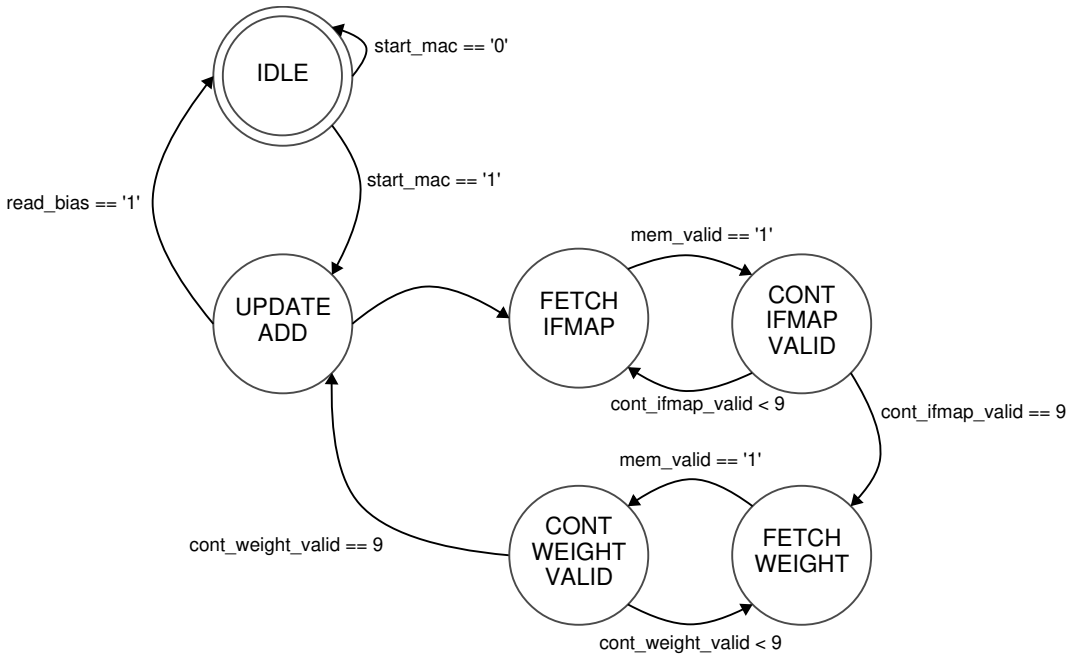


Figure 4.15: OS accelerator Fetch FSM.

4.2.4 Final Remarks

This Chapter described an original Thesis contribution, corresponding to the design of convolution hardware accelerators for CNNs, at the RTL level. This set of accelerators is a start point for an open-source CNN accelerators benchmark, enabling designers to compare different implementations. An open-source benchmark is also a gap observed in the literature.

The implementations explored both the array type (1D and 2D) and the different dataflow architectures, which differ by the data that is reused. Emphasis was given to the memory interface, including a mechanism that allows parameterizing their latency to evaluate the performance and the energy consumed by the accelerator, in a fair way (same technology, same target frequency, unified memory interface).

Next Chapter evaluates each accelerator, performing a quantitative performance evaluation among them.

5. MACHINE LEARNING HARDWARE ACCELERATOR RESULTS

This Chapter evaluates the designed convolution hardware accelerators presented in the previous Chapter. This Chapter is organized as follows:

- Section 5.1: evaluates and compares the array styles – 1D and 2D;
- Section 5.2: evaluates and compares the dataflow types – WS, IS, and OS.

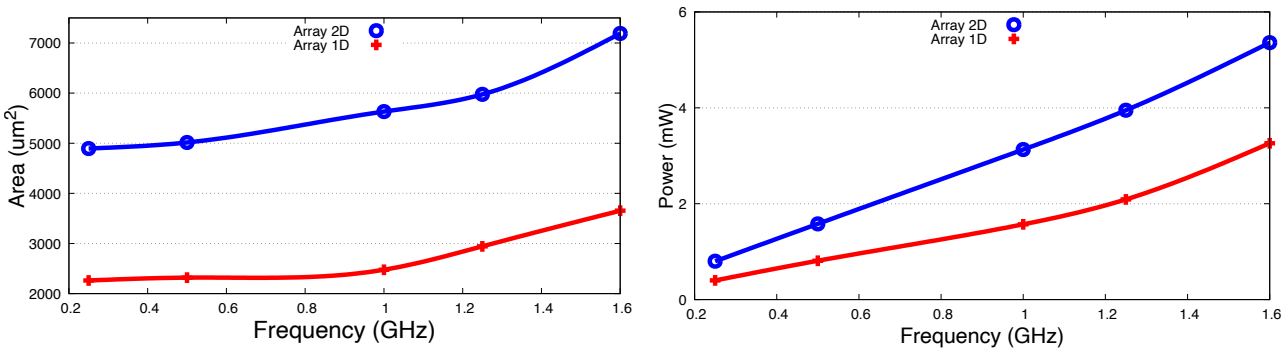
5.1 Array Style Results

This section evaluates the 1D and 2D array styles (Section 4.1). The DSE uses results obtained after physical synthesis, using the Cadence Genus tool for logic synthesis and the Cadence Innovus tool for physical synthesis. The power dissipation estimation uses as input a VCD file generated after the post-synthesis netlist simulation and the Cadence Voltus tool. The simulated netlist is a 32x32x3 feature map, with 16 3x3 filters, stride 2, generating a 15x15x16 output, which represents one layer of a CNN trained using the CIFAR10 dataset (same CNN used in Chapter 3).

Table 5.1 presents results varying the accelerator architecture (2D/1D) for a 28nm technology node. The select frequency is the one that results in a slack time equal to or near zero. Figure 5.1 presents the physical synthesis for both accelerators as a function of the frequency (0.25–1.6 GHz).

Table 5.1: PPA results for accelerators after physical synthesis (28nm@1.6GHz). The leakage power for 1D is $0.02mW$, while 2D has $0.04mW$.

Accelerator	Area – μm^2	Cell Count	Total Power – mW
Array 1D	3,654.70	2,964	3.26
Systolic 2D	7,190.91	5,922	5.36



(a) Area of 1D array and systolic 2D

(b) Power dissipation of 1D array and systolic 2D

Figure 5.1: Area-power results for 28nm as function of the frequency.

As expected, the power increases with the frequency. Note that the area rises for frequencies higher than 1GHz, due to the synthesis tool effort to meet the target frequency, mainly for the 1D array architecture. Results presented in Table 5.1 and Figure 5.1 are consistent with the accelerator architectures since the 2D architecture has nine MACs (in fact 6 MACs, 3 adders, 3 multipliers), and the 1D has three MACs in the arithmetic core.

DSE

The DSE uses Table 5.1 results, by multiplying the physical synthesis results by the total number of MACs to be used. We integrate the physical synthesis data to the URSA simulator to execute the DSE. The DSE considers five parameters:

- Accelerator architecture: 1D array and Systolic 2D.
- Parallelism: as presented in Chapter 2, accelerators can present an amount of MACs greater than 200 ([Chen et al., 2020] use 256 MACs). Our accelerators have 9/3 MACs (2D/1D), making it possible to parallelize these accelerators to process several channels simultaneously. The DSE explores from 1 to 16 accelerators in parallel, ranging from 9/3 to 144/48 MACs (2D/1D).
- Power, area: design parameters obtained from the physical synthesis for different frequencies.
- Performance: execution time to execute one 32x32x3 convolution, with 3x3 filters, stride 2x2, and 16 channels. Performance in represented in milliseconds to demonstrate the difference in performance and the benefits of using accelerators in parallel with high frequencies.

Charts presented in Figure 5.2 summarize the results for 40 evaluated scenarios (two accelerators architectures, four parallel configurations, and five operating frequencies). The charts present the PPA for each scenario. From the charts, it is possible to observe, for example:

- 1D array is, as expected, indicated for smaller area and power when compared to 2D systolic at the same frequency and number of filters, as shown in the scenario highlighted in red in both charts from Figure 5.2 (16 parallel accelerators@1.6GHz).
- systolic 2D is, as expected, indicated for higher performance when compared to 1D array (also shown in the scenario highlighted in red). Observe that the adoption of 16 accelerators for 2D systolic is only justified at frequencies higher than 1GHz. For smaller frequencies, eight accelerators deliver similar performance, with smaller area and power.

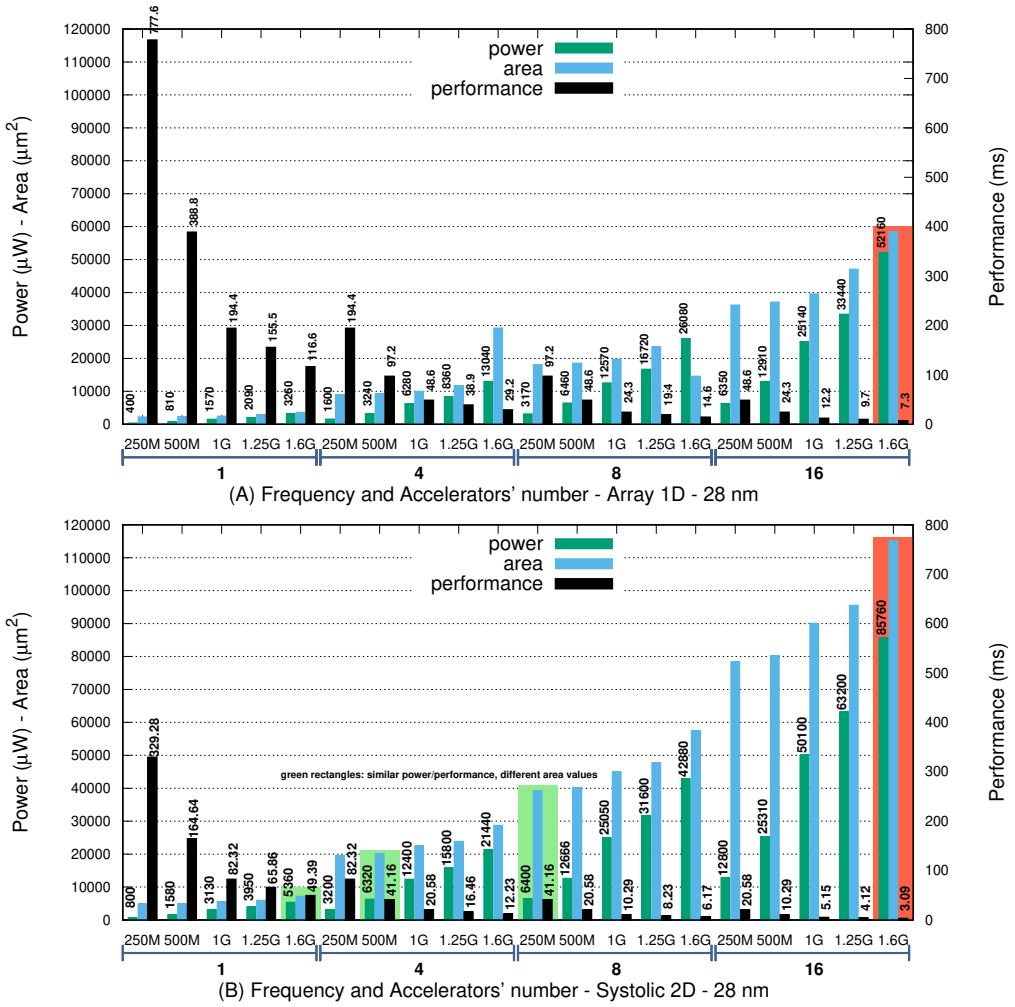


Figure 5.2: DSE results obtained with URSA for 28nm for 1D array and systolic 2D (note that power is presented in μW).

- consider the 2D architecture, Figure 5.2(b), for a 6.4mW power budget (green rectangles). The candidate configurations are 1 acc@1.6GHz, 4 acc@500MHz, and 8 acc@250MHz (*acc* stands for accelerator). The power and performance data are similar for these scenarios, but the area is much smaller using 1 accelerator. This chart allows the user to select the optimum accelerator configuration according to its constraints.

- others points can be observed through these charts. For example, still considering 2D architecture (Figure 5.2(b)). It is possible to note that it is preferable to use 1 acc@1GHz than 4 acc@250MHz, once it presents similar power and performance, but 4 times smaller area. Similar behavior occurs with 4acc@1GHz compared to 8 acc@500MHz.

- comparing 1D with 2D architectures for a 3.2mW power budget:

  – 1D, 4 acc@500MHz: 9,276μm^2, and 97.2ms;

  - 2D, 4 acc@250MHz: 19,58μm^2, and 83.32ms.

  In this case, the 1D array is the choice since, despite 15% lower performance (97.2 versus 82.32ms), it presents 50% smaller area (9,276 versus 19,58μm^2).

The average energy consumption for the 1D array is 313μJ up to 1.25GHz, increasing to 380μJ@1.6GHz. On the other hand, the systolic 2D presents an average energy consumption equal to 261μJ, regardless the frequency. Thus, the systolic 2D presents a better energy efficiency than the 1D array due to its performance. Such result reveals that one cannot consider only the number of arithmetic cores for decision making since a set of blocks are common to both architectures, as the register files.

Although we used the URSA simulator to perform DSE (Figure 5.2), we concluded that it is possible to perform DSE from physical design results by using an analytical approach. This method to perform DSE analytically, from a set of physical synthesis data, is the starting point for developing DSE flows, described in Chapter 6.

5.2 Dataflow Type Results

This section presents PPA results considering different dataflow types (Section 4.2). DSE is executed after physical synthesis. Cadence Genus and Innovus tools were used for logic and physical synthesis, with 28nm technology and a frequency of 500MHz. The logic synthesis uses clock-gating, added automatically by the tool, to reduce the accelerator energy consumption. The power dissipation is obtained with a VCD file generated with a post-synthesis netlist simulation and Cadence Voltus tool. The netlist simulation input is the first CNN layer with a 32x32x3 IFMAP (from CIFAR10 dataset - same from Chapter 3), 16 3x3 filters, stride 2, generating a 15x15x16 output. All inputs adopt 8-bit quantization and outputs 20 bits (quantization detailed at Chapter 3). The total energy is computed by multiplying the average power by the number of clock cycles required to execute a complete convolution.

The energy memory values were extracted from Cacti-IO tool [Jouppi et al., 2014]:

- SRAM. For a 28nm 4KB SRAM, Cacti-IO reports 260fJ/bit for reading and 180fJ/bit for writing. For comparison purposes, the literature reports energy values between 67fJ/bit [Fujiwara et al., 2013] and 20fJ/bit [Haine et al., 2017] for 28nm SRAM. The SRAM consumption is larger than the one reported in the literature for two reasons: (*i*) we considered a 500MHz frequency, while the literature considers 200MHz; (*ii*) the literature considers low-energy SRAM.

- DRAM. For a 16 kB DRAM, 12pJ/bit for reading and 11.7pJ/bit for writing. The literature shows values of 20pJ/bit [Son et al., 2013] and 15pJ/bit [Li et al., 2019] for DRAM. Thus, the DRAM values are close to the ones reported in the literature.

Figure 5.3 evaluates the energy consumption to obtain a 15x15x16 OFMAP, according to the memory type and its access latency (x-axis). The memory latency is controlled by the `ifmap_valid` (Section 4.2). Increasing the IFMAP and OFMAP does not change the behavior observed in these graphs. Accelerators that do not adopt output buffers have a smaller energy consumption than accelerators using output buffers when using SRAM. This result is because SRAM and buffers use the same static memory implementation, resulting in a similar consumption. *Thus, buffering brings no advantage when using SRAM memories*. The observed result is inverse when using DRAMs as external memories, with buffers acting as cache memories. The buffered IS accelerator presents a significant energy reduction (IS buf) compared to the other implementations. The OS accelerator is expensive in terms of energy because it constantly fetches data from the input memory, regardless of the memory type.

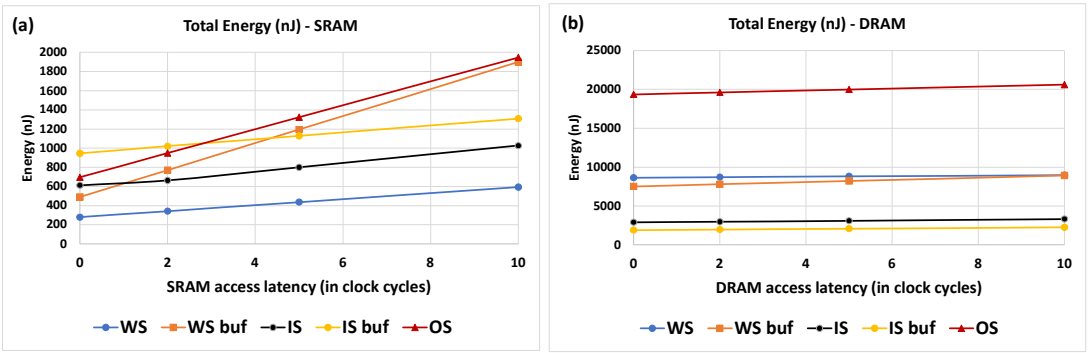


Figure 5.3: Convolutional accelerators energy varying the memory type (SRAM or DRAM), and the access latency.

Figure 5.4 evaluates the accelerators performance. The IS accelerators present a performance that is slightly affected by the memory latency because the IFMAP is read once, that is, input stationary. Also, the bufferization of weights and bias reduces the memory access, decreasing the memory impact on performance (line 1 of Algorithm 2 on Section 4.2.2). The WS performance is affected by the memory latency because the number of IFMAP readings is higher than the IS architecture. The OS architecture has a small buffer in the output, requiring frequent IFMAP and weight readings, resulting in a heavy performance penalty due to the memory latency.

Table 5.2 show the obtained results for the accelerators, considering an SRAM with access latency of 2 clock cycles. It is possible to note a reduction in area whem compared to the NVDLA approach on Chapter 3 . As the NVDLA is an approach without buffer, it is possible the compare with non-buffered WS, and OS approches, which has an average area

Table 5.2: Hardware Metrics for SRAM Memory.

	WS	WS buf	IS	IS buf	OS
Energy Memory (nj)	178.42	153.02	44.19	18.79	410.54
Energy Core (nJ)	163.63	617.16	619.28	1,001.79	537.37
Area (μm^2)	6,319	26,779	46,543	68,077	6,596
Performance (cycles)	225,078	225,078	135,450	135,450	595,093
Memory Writes	10,800	3,600	10,800	3,600	3,600
Memory Reads	78,256	71,056	14,398	7,198	194,735
Internal Buffer Size (bits)	80	4,580	3,584	8,384	28

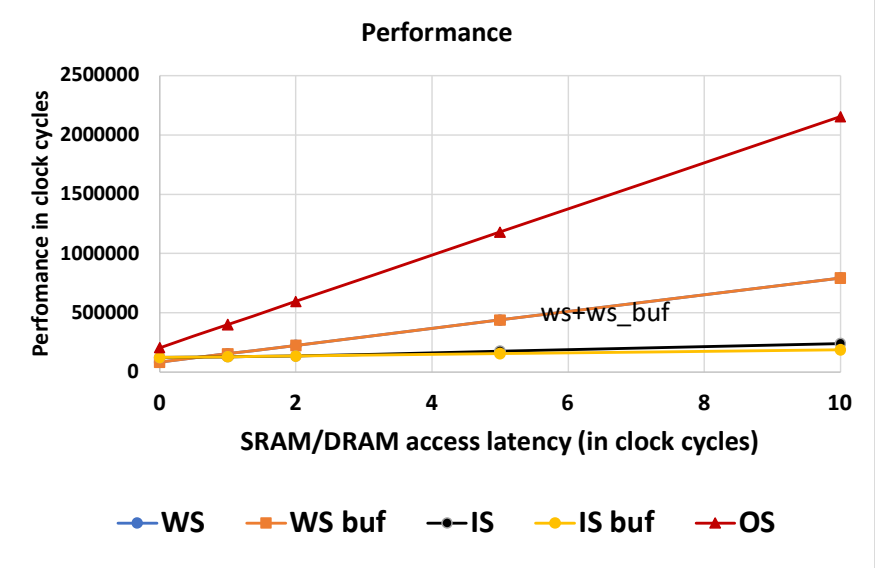


Figure 5.4: Convolutional accelerators performance (execution time).

of 6,457.91μm^2. The NDVLA approach present an area of 35,003μm^2, which means an reduction of 5.42 times.

Figure 5.5 details the results in a radar format. Required buffer size (internal buffers), area, and performance (cycles) are normalized by the worst result among accelerators. The energy (memory and core) is normalized by the worst total energy, as well as the memory accesses (reads and writes). Figure 5.5 shows that:

- the convolutional core (energy core) consumes the large parcel of the total energy consumption for WS and IS dataflows. However, OS present similar values for core and memory energy.
- buffering at the output effectively reduces memory writings (from 10,800 to 3,600). The normalization masks these results due to the higher number of readings (71,056). Buffering brings a slight reduction in the memory energy consumption at the cost of larger area and core consumption (WS versus WS_buf and IS versus IS_buf).

- IS is 1.66 times faster than WS (225,078 and 135,450 clock cycles for WS and IS, respectively). However, this performance increase is obtained by buffering all input values, which generates an increase in the accelerator area.
- OS is penalized by the memory readings, resulting in the worst performance and highest memory and core energy consumption. Despite these disadvantages its presents a small area footprint, similar to the WS.

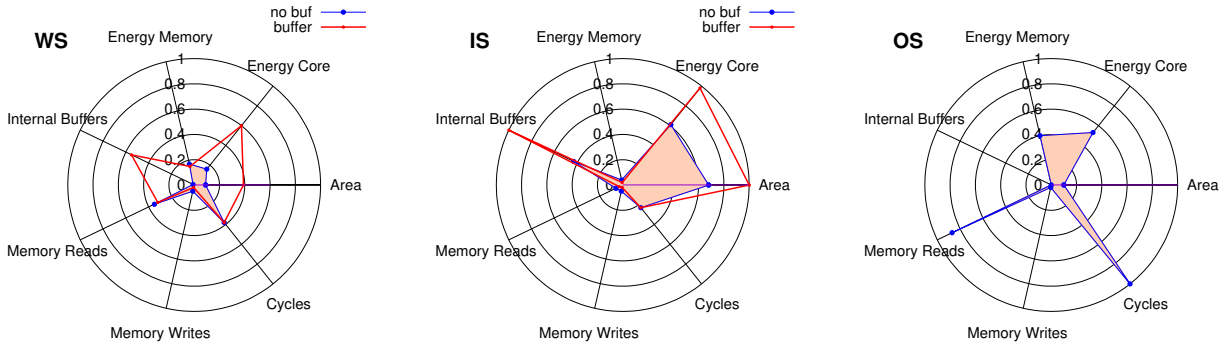


Figure 5.5: Performance for the convolutional accelerators, considering a 32x32x3 IFMAP, 15x15x16 OFMAP, stride=2, and a **2** clock cycle **SRAM** latency. The filled area highlight the non-buffered approach. The values are normalized by the worst value of each radar axis.

To summarize, when using SRAM as external memory, WS is the accelerator with the smallest area and energy consumption, while IS presents the best performance.

Table 5.3 show the obtained results for the accelerators, considering an DRAM with access latency of 5 clock cycles. Area, memory writes, and internal buffer size are the same as the accelerator using SRAM (Table 5.2). The memory reads has an small difference due the difference in memory latency. These differences are detailed in Chapter 6.

Figure 5.6 presents results when using DRAM as external memory, with a latency of 5 clock cycles. OS is omitted once it present the worst characteristics regarding performance (see Figure 5.4) and memory access (see Figure 5.5), and its characteristics get worse with the use of DRAM. Figure 5.6 shows that:

- The DRAM (energy memory) consumes the large parcel of the total energy consumption for the most accelerator. Buffered IS has similar values for memory and core energy. Its occurs once buffered IS has a big amount of register, once it use buffers for all inputs and the partial values;
- the IS architecture is 2.5 times faster (403,102 and 175,269 clock cycles for WS and IS, respectively), with an energy consumption up to 3.65 times smaller (8,543.22 and 2,337.87nJ for WS and IS, respectively) than WS. Note that the use of buffers in the

inputs helps to decrease the memory latency impact on performance, but increase the area;

- WS still the accelerator with the smallest area. The WS is 7.37 times smaller (6,319.27 and 46,543.33 for WS and IS, respectively) than IS. The buffered WS is 2.54 (26,779.32 and 68,077.57 for WS and IS, respectively) than buffered IS;
- Buffering the IS accelerator reduces its total energy in 1.49 times, improves the performance in 1.13 times, with an increase of 1.46 times in area.

Table 5.3: Hardware Metrics for DRAM Memory.

	WS	WS buf	IS	IS buf	OS
Energy Memory (nj)	8,543.22	7,172.17	2,337.87	966.81	19,076.91
Energy Core (nJ)	257.96	1,040.93	755.41	1,110.25	912.03
Area (μm^2)	6,319	26,779	46,543	68,077	6,596
Performance (cycles)	438,105	438,102	175,269	155,019	1,179,253
Memory Writes	10,800	3,600	10,800	3,600	3,600
Memory Reads	78,208	71,008	13,723	6,523	194,720
Internal Buffer Size (bits)	80	4,580	3,584	8,384	28

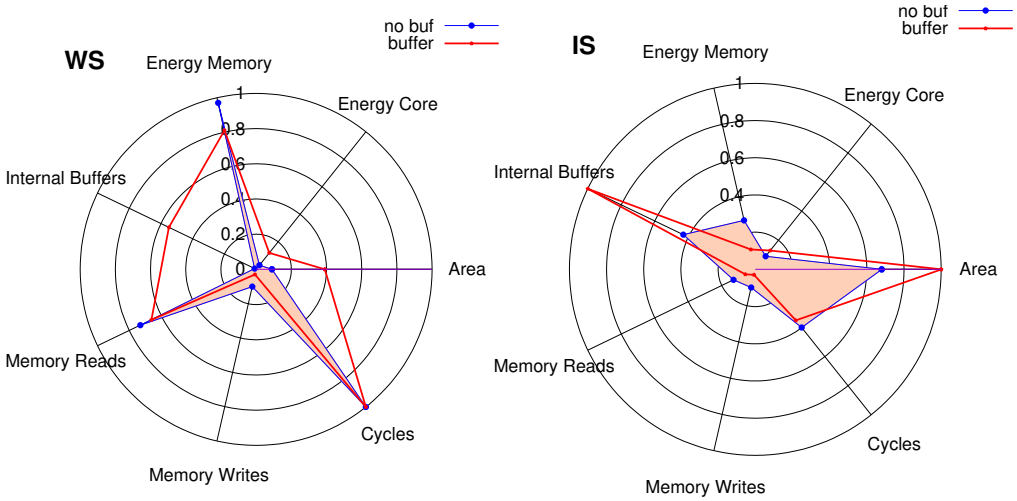


Figure 5.6: Performance for the convolutional accelerators, considering a 32x32x3 IFMAP, 15x15x16 OFMAP, stride=2, and a **5** clock cycle **DRAM** latency. The filled area highlight the non-buffered approach. The values are normalized by the worst value of each radar axis.

To summarize, when using DRAM as external memory, buffered IS is the accelerator to use for low power applications, once the use of internal buffers helps to reduce the memory access, reducing the energy consumption. It is possible to reduce energy and improve performance of the IS dataflow by using an output buffer at a significant area overhead cost. Small area applications are suitable for WS dataflow.

5.2.1 Final Remarks

The proposed evaluation method allows to compare fairly the proposed accelerator architectures (regarding same technology and target frequency, for example). Thus, it is possible to analyze which accelerator is suitable for a specific application. The obtained results also validate the proposed physical synthesis flow and the analytical DSE method, both described in Chapter 6.

6. DESIGN SPACE EXPLORATION FLOWS

Chapter 3 presented a method for DSE using a system simulator, URSA, adopting the accelerator proposed by NVIDIA, called NVDLA. Despite the advantages of using URSA to model a complete computational system, it presented the following drawbacks related to model CNNs:

1. Accelerator modeling close to the actual hardware (cycle-accurate), which can generate a redundant implementation;
2. An RTL implementation still required for extract PPA information. URSA does not allow synthesis. Thus, RTL implementation is still required to get hardware implementeation results such as area;
3. Some hardware aspects may be omitted by the high-level modeling. For example, synchronization states can be omitted.

These drawbacks led us to Chapters 4 and 5, where we present different hardware architectures at the RTL level, evaluating their performance after logical and physical synthesis steps. Thus, this Chapter presents the flows used to carry out DSE for convolutional neural networks (CNNs). This Chapter is organized as follows:

- Section 6.1: DSE using a physical synthesis flow, called *synthesis flow*;
- Section 6.2: DSE using only MAC-related physical synthesis data. We compare results obtained with this flow with those obtained with the synthesis flow, and because it is a method used in related works;
- Section 6.3: DSE using an analytic approach derived from the synthesis flow. This analytic flow is an original contribution of this Thesis, as it allows fast and accurate DSE for CNNs.
- Section 6.4: presents results related to the three DSE methods, as well as a comparison with DSE presented in the literature.

As in Chapter 3, we assume TensorFlow framework as front-end. TensorFlow is adopted because it allows abstract modeling of CNNs, and has the necessary infrastructure for their training. Once CNNs are modeled and trained, the DSE flows are used to choose the accelerator that meets the designer's constraints.

6.1 DSE Physical synthesis Flow

The DSE physical synthesis flow perform both logical and physical synthesis steps, and uses IFMAP and weights data from real CNNs to obtain PPA results. The method includes the following steps:

(i) describe the CNN at the TensorFlow framework, exporting the weights and IFMAP values in VHDL packages format to be used in the RTL and gate-level simulations;

(ii) execute the logical and physical synthesis steps of the accelerator being evaluated;

(iii) obtain PPA values after post-layout simulation considering the switching activity derived from the CNN values.

Figure 6.1 details the physical synthesis flow for PPA extraction. The first step is to model the CNN application in the TensorFlow framework to generate the VHDL packages (**tensorflow.vhd** and **gold.vhd** files), used in the RTL and gate-level simulations. The **tensorflow.vhd** package contains the weights and IFMAP values, while the **gold.vhd** package contains the expected outputs. This step also generates the file **parameters.txt**, responsible for configuring the RTL description. This step is generic, supporting different CNNs, such as MNIST or CIFAR10.

The second step is the RTL simulation to verify the accelerator description behavior. This step uses the **tensorflow.vhd** and **gold.vhd** to perform the RTL simulation and verify if the accelerator behavior is correct. It is necessary to check if the simulation output matches the expected values during the development of a new accelerator. Once validated the accelerator description, it is possible to bypass this second step.

The third and fourth steps correspond to the physical synthesis. The third step is the logical synthesis, which has as inputs the technology files (LIB and LEF files) and constraints (as clock frequency or power), and as outputs the gate-level description (netlist) and a constraint file the be used in the next step. The fourth step is the physical synthesis, corresponding to the placement and routing procedures. This step generates a new netlist, with annotated wire capacitances (**netlist.v**) and a set of reports.

The fifth step is the annotated gate-level simulation, also using the **tensorflow.vhd** and **gold.vhd** files. This step may fail due to the applied constraints, as clock frequency and input/output delays. In this case, the designer must modify the constraints used in the third and fourth steps to obtain a netlist that simulates correctly. The output of this step is the **dump.vcd** file, with the switching activity induced by the CNN IFMAP and weight values. The last step uses the VCD file to estimate the accelerator power dissipation.

The execution of this flow produces an accurate PPA estimation for a given accelerator architecture with actual CNN data. However, it is necessary to execute this flow for each new set of weights and inputs. The reason is that different data sets present dif-

ferent switching activities, changing the power dissipation. Also, the hardware may show differences due to the number of channels in a given layer or the IFMAP and OFMAP sizes, changing the number of bits in counters or the buffers depth. It is worth noting that the accelerators descriptions are configurable according to the CNN parameters, such as IFMAP and filter sizes, not requiring any designer intervention when the CNN features change. Thus, *we have an accurate PPA but requiring a significant processing time, which can take several hours, once the logical and physical synthesis steps can take together about 10 (for a non-buffered architecture) and 25 (for a buffered architecture) minutes for a 3x3 accelerator*. This value can increase for larger accelerator arrays.

6.2 MAC-based DSE Flow

The DSE physical synthesis flow described in the previously Section allows to obtain accurate results. However, it can take several hours to be executed. This Section describes a widely used DSE flow based on an estimate of the required number of MACs. Several works in the literature use the MAC-based method to estimate area and power [Tang and Xie, 2018, Parashar et al., 2019, Heidorn et al., 2020, Zhao et al., 2020, Cao et al., 2020], reducing the time spent in physical synthesis.

This DSE flow uses the PPA values (in fact, only power and area) related to MACs and registers extracted from the physical synthesis flow. The power and area are estimated from the number of MACs required by the accelerators.

The PPA estimation accuracy of this flow is expected to be worse than the physical synthesis flow, as it does not consider the effect related to the control circuitry (FSMs), buffers, and accesses to the memory. We use this flow as a baseline, because it is a method adopted in the literature to estimate the PPA of CNNs.

We propose two approaches for this flow. The first uses only the MAC data, while the second considers the MAC input and outputs registers. The second approach is expected to reduce the PPA estimation error compared to the MAC-only method. The Results section presents data for both approaches.

6.3 Analytic DSE Flow

As mentioned before, on one hand, a physical synthesis flow is accurate but can take several hours to be executed. On the other hand, MAC-based DSE can generate innacurate results because it ignores relevant parts of the circuitry. The *proposed analytic DSE flow* estimates the PPA of CNN layers in a analytic way, by using results obtained from the physical synthesis of one layer of a CNN application. Thus, it is possible to reduce the PPA estimation time, corresponding to a trade-off between the physical synthesis and

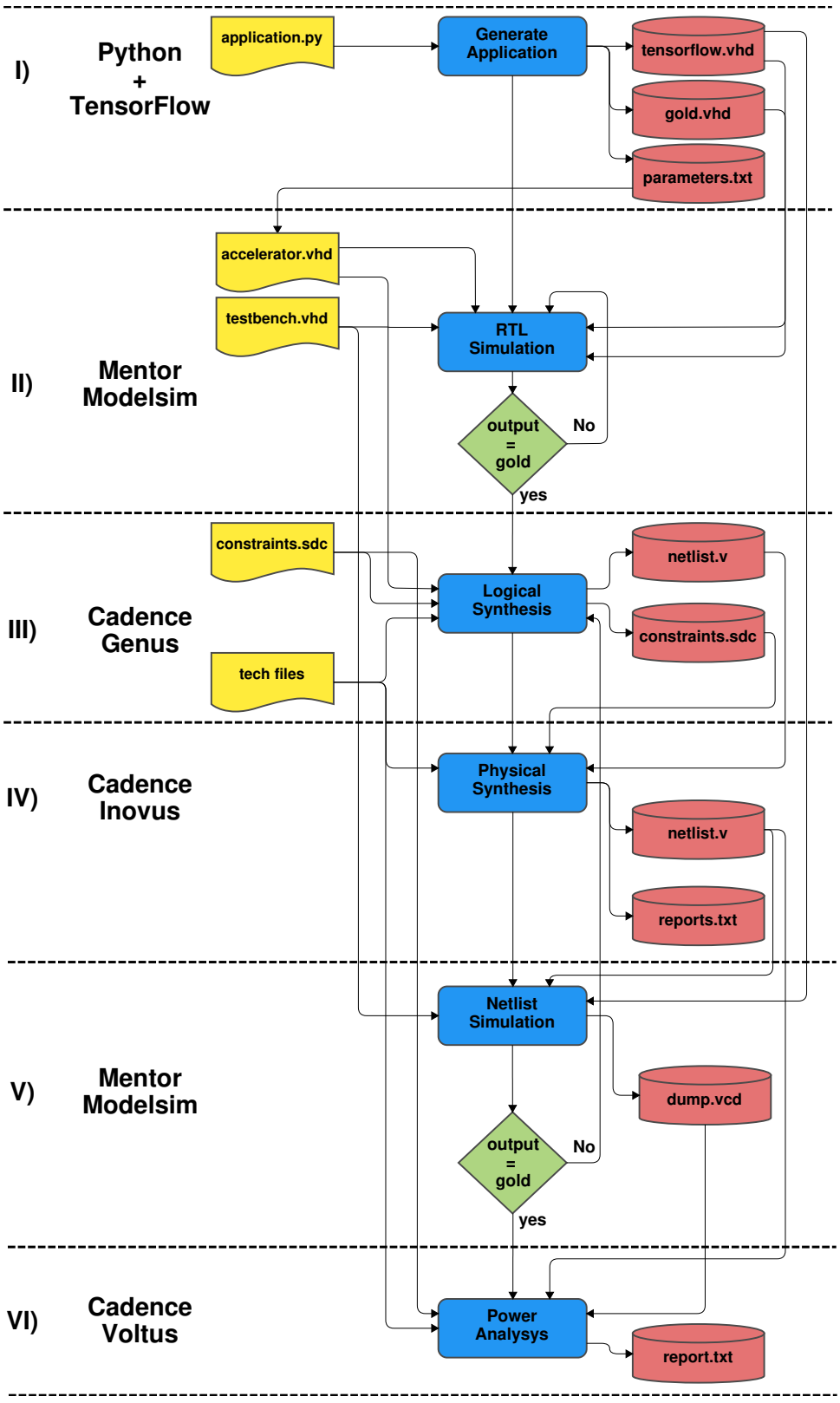


Figure 6.1: DSE physical synthesis flow for PPA extraction.

MAC-based flows. This flow is faster than the physical synthesis flow because the synthesis is executed once for each accelerator, and more accurate than the MAC-based flow, once consider all the convolution, not only MACs. The analytic DSE flow is an original contribution of this Thesis.

Figure 6.2 shows the analytic DSE flow. The **TensorFlow + Quantization** step and the **Physical synthesis** described in the Figure are the same of Figure 3.1. The **Analytic Flow** is a step that parses the PPA logs generated by the physical synthesis. The analystic flow uses the file **accelerator.txt** as input, containing the hardware parameters, like input size. After read the inputs, the Analytic Flow step get the PPA values from physical synthesis, parser the reports to obtain the values for DSE, and report the results. The analytic flow also uses the Cacti-IO tool [Jouppi et al., 2014] to estimate the energy related to the memory accesses.

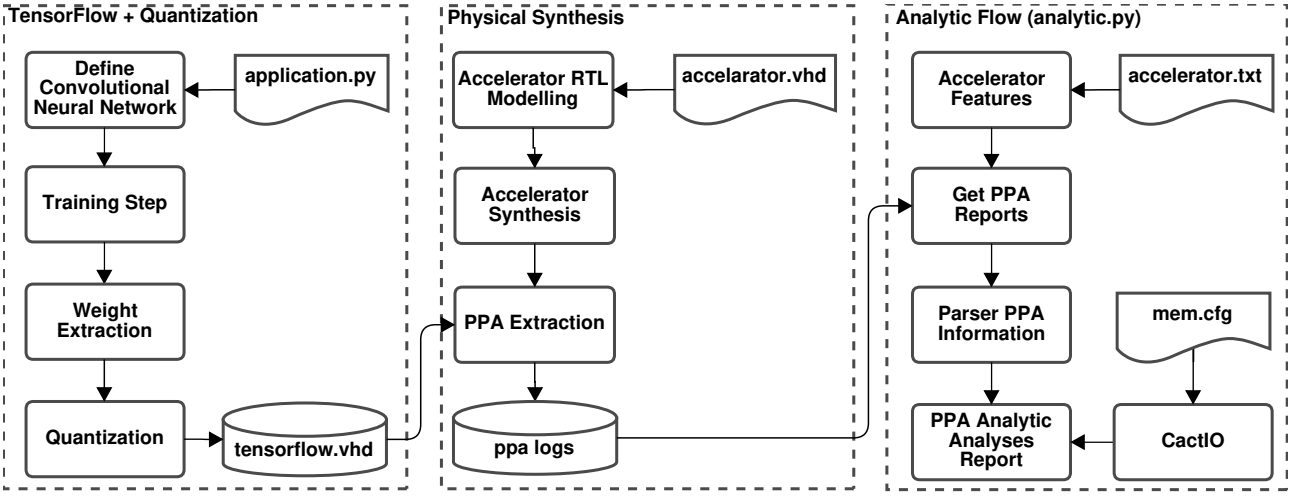


Figure 6.2: DSE analytic flow for PPA extraction.

The **accelerator.txt** file contains the hardware parameters described below. Each parameter corresponds to a variable in the analytic model, represented in italic:

(a) Clock period (ns);

(b) Word size (bits);

(c) 2D dataflow type: WS, buffered WS, IS, buffered IS, or OS;

(d) Memory type: SRAM or DRAM. The memory type defines its latency - *MemLat*;

(e) IFMAP size: single integer value (we assume square IFMAPs) - *IFMAP_D*;

(f) Number of input channels: integer value - *InChannels*;

(g) Number of output channels: integer value - *OutChannels*;

(h) Filter size: single integer value (we assume square filters) - *Filter_D*;

(i) Stride: integer value.

For example, a designer may estimate the first layer of a systolic 2D WS accelerator, entering the following parameters: a) clock period=2ns; b) 16-bit word size; c) WS; d) SRAM memory type; e) IFMAP size=32 (cifar10); f) 3 input channels (RGB); g) 16 filters; h) fiter size=3 (3x3 filters); i) stride=2.

The analytic flow produces as outputs:

- Power: power values for the accelerator, output buffer, and the sum of both (total power), *mW*;
- Performance: number of clock cycles required to execute the layer convolution;
- Area: area values for the accelerator, output buffer, and the sum of both (total area), μm^2;
- Input buffer size: number of bits to store all CNN layer input values;
- Accelerator energy: total power $\times$ number of cycles $\times$ clock period, *fJ*;
- Memory accesses:
  - Number of input memory reads (IFMAPs, weights and bias);
  - Number of input memory writes (always zero, once the input memory acts as a ROM);
  - Number of output memory reads (partial sums values);
  - Number of output memory writes (partial sums values, and OFMAPs);
- Memory read energy: total memory reads $\times$ energy per reading (estimated by Cacti-IO), *nJ*;
- Memory write energy: total memory writes $\times$ energy per writing (estimated by Cacti-IO), *nJ*;
- Total energy: accelerator energy + memory read energy + memory write energy, *nJ*.

The behavior of each dataflow type provides the equations to estimate the number of clock cycles (i.e., performance), the number of memory reads, and the number of memory writes. This behavior comes from the RTL simulation, which gives the number of cycles and memory accesses for each convolution, according to the FSMs controlling the hardware, mapping the performance to the analytic model parameters. Accelerators that need output buffer have the area and power values estimates using an interpolation method, with data obtained after simulating at least three layers, using the physical synthesis flow (Section 6.1).

The OFMAP size is a function of the IFMAP size (*IFMAP_D*), filter size (*Filter_D*), and the stride value. The analytic model computes OFMAP according to Equation 6.1.

$$OFMAP\_D = \left\lfloor \frac{IFMAP\_D - FILTER\_D}{Stride} + 1 \right\rfloor \tag{6.1}$$

For example: a 32 $\times$ 32 IFMAP, with 3 $\times$ 3 filters and stride=2 generates a 15 $\times$ 15 OFMAP. The Next Sections detail the methods and equations used to build the analytic flow.

6.3.1 Performance Estimation

Each accelerator has an equation to generate its performance in clock cycles. WS and buffered WS have similar performance, represented by Equation 6.2. Weights and bias are stationary, i.e., pre-loaded in a buffer.

$$CyclesWS = 6 \times OFMAP\_D^2 \times InChannels \times OutChannels \times (1 + MemLat) \tag{6.2}$$

Where:

- 6 constant[1]: number of clock cycles to read 9 (3×3) IFMAP values. Due to the stride value (equal to 2), each reading reuse one column, reducing memory accesses;
- $OFMAP\_D^2 \times InChannels$: number of convolutions to produce one output channel. Remember that the IFMAP reading and the convolution occurs in parallel (pipeline implementation);
- The process is repeated for all output channels (*OutChannels*);
- The constant value added to the memory latency (1 clock cycle) corresponds to the address phase.

Equation 6.2 is responsible for most part of the required cycles to compute the convolution of a given layer (>80%). The analytic flow also computes the time spent to read the weights, and the number of 'bubbles' in the pipeline when it is necessary to return to the first X coordinate, after *OFMAP_D* convolutions.

IS and buffered IS have similar performance, represented by Equation 6.3. In the IS approach values read from the IFMAP are stationary, i.e., they are used to compute a partial output value at each output channel.

$$\begin{aligned} Term1 &= OutChannels \times (1 + MemLat) + (Filter\_D^2 \times OutChannels \times InChannels) \times (1 + MemLat) \\ Term2 &= Filter\_D^2 \times OFMAP\_D^2 \times InChannels \times (1 + MemLat) \\ Term3 &= (9 \times OFMAP\_D^2 \times InChannels \times OutChannels) \\ CyclesIS &= Term1 + Term2 + Term3 \end{aligned} \tag{6.3}$$

Where:

- *Term*1: cycles to load bias and weights, and store in internal buffers. The number of bias values is equal to the number of *OutChannels*;
- *Term*2: cycles to read *Filter_D* $\times$ *Filter_D* IFMAP values read from the memory;

[1] The current analytic model only considers 3×3 filters and stride=2.

- *Term*3: cycles to execute all convolutions of the layer.

Equation (6.3) represents most part of the cycles to compute the convolution of a given layer (*CyclesIS* >83% for IS and *CyclesIS*>90% for buffered IS). As in the previous equation, the time spent with bubbles is also accounted by the proposed analytic flow.

The WS dataflow reads the IFMAP for each partial result (Equation 6.2). For the IS dataflow, a partial reading of the IFMAP is performed (Term 2 of Equation 6.3), reusing these values for all partial convolutions (Term 3 of Equation 6.3). This remark is consistent with the results presented in Chapter 5, where the IS performance is better than the WS. On the other hand, IS requires buffers to store input values, penalizing its area.

The OS dataflow does not have buffers for IFMAP and weight values. Thus, the OS dataflow reads 18 values from the input memory to execute each convolution (9 weights and 9 IFMAP values). Due to the pipeline implementation, the convolution occurs in parallel to the memory reading. Equation 6.4 computes most of the required cycles to compute the convolution in a given layer (>98%). The analytic flow considers the number of clock cycles to write in the OFMAP memory and the bubbles in the pipeline. The number of memory readings is the main difference concerning the WS dataflow (Equation 6.2), which is larger in OS.

$$CyclesOS = 18 \times OFMAP\_D^2 \times InChannels \times OutChannels \times (1 + MemLat) \tag{6.4}$$

6.3.2 Memory Accesses Estimation

Each dataflow has a specific equation related to the number of memory readings. Equation 6.5 presents the number of memory readings for WS and buffered WS.

$$\begin{aligned} Term1 &= 6 \times (OFMAP\_D + 5) \times InChannels \times OutChannels \\ Term2 &= (Filter\_D^2 + 1) \times InChannels \times OutChannels \\ Term3 &= 6 \times OFMAP\_D^2 \times InChannels \times OutChannels \\ MemReadWS &= Term1 + Term2 + Term3 \end{aligned} \tag{6.5}$$

Where:

- *Term*1: refers to “invalid” readings. At the end of each row, the WS accelerator accesses memory locations not used in the convolution. It would be possible to avoid these readings at the cost of more control logic in the hardware. Our design choice was to keep the hardware simple.
- *Term*2: number of reads to load weight and bias values;

- *Term*3: number of reads to load IFMAP values (core of Equation 6.2).

Equation 6.6 presents the number of memory readings for IS and buffered IS.

$$\begin{aligned} Term1 &= OutChannels + ((Filter\_D^2) \times InChannels \times OutChannels) \\ Term2 &= Filter\_D^2 \times OFMAP\_D^2 \times InChannels \\ MemReadIS &= Term1 + Term2 \end{aligned} \tag{6.6}$$

Where:

- *Term*1: number of reads to load bias and weight values;
- *Term*2: number of reads to load *Filter_D* $\times$ *Filter_D* IFMAP values from the memory.

Equation 6.7 presents the number of memory readings for the OS dataflow.

$$MemReadOS = 18 \times OFMAP\_D^2 \times InChannels \times OutChannels \tag{6.7}$$

It is possible to observe the smaller number of memory accesses for the IS dataflow (Equation 6.6) *Term*2 compared to the WS and OS dataflows (Equations 6.5 *Term*3 and 6.7).

Equation 6.8 computes the number of memory writings for a buffered accelerator (WS and IS). The output buffer reduces the memory writes once partial results are stored on it.

$$OfmapWrites\ (buffered\ acc.) = OFMAP\_D^2 \times OutChannels \tag{6.8}$$

Equation 6.9 computes the number of memory writings for a non-buffered accelerator (WS, IS, and OS). Non-buffered accelerators read and write partial sums in the output memory.

$$OfmapWrites\ (non\ buffered\ acc.) = OFMAP\_D^2 \times OutChannels \times InChannels \tag{6.9}$$

6.3.3 Output Buffer Area and Power Estimation

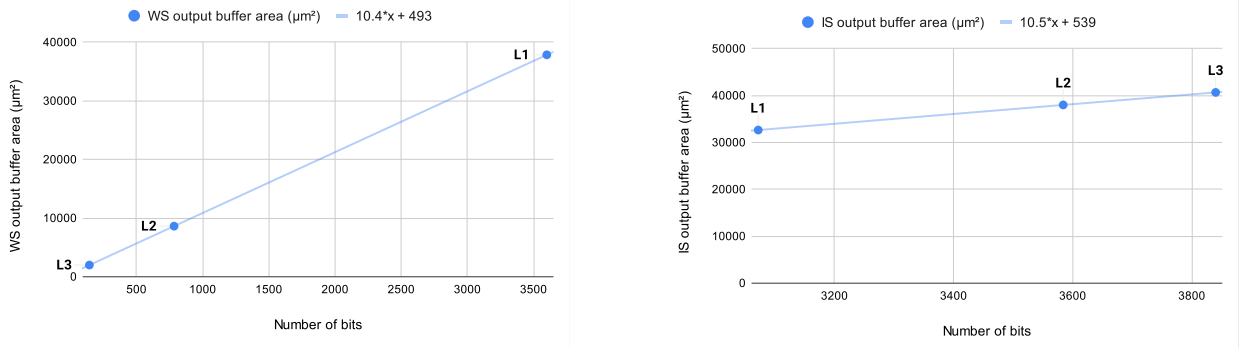
The output buffer area and power are obtained from interpolation. The data source for the interpolation are the results obtained from the physical synthesis flow for a three-layer Cifar10 CNN (which is presented in Section 6.4). The variable *NumBits* is the number of bits of each output buffer. Equation 6.10 computes the number of bits for the WS output buffer. The WS output buffer has the size of one OFMAP channel ($OFMAP\_D^2$) multiplied by the word size (16 bits).

$$NumBits = OFMAP\_D^2 \times 16 \tag{6.10}$$

Equation 6.11 computes the number of bits for the IS output buffer. The IS dataflow computes one line of results (*OFMAP_D*), for all output channels (*OutChannel*).

$$NumBits = OFMAP\_D \times OutChannel \times 16 \tag{6.11}$$

Figures 6.3a and 6.3b present in the x-axis the number of bits for each dataflow, and in the y-axis the area. The Cifar10 CNN has three convolutional layers. The OFMAP sizes for each layer are (Figure 6.5, page 103): L1: 15x15, 16 output channels; L2: 7x7, 32 output channels; L3: 3x3, 64 output channels. According to Equation 6.10 the size of the WS output buffers decreases from layer 1 to layer 3 since the OFMAP size reduces. On the other side, according to Equation 6.11 the size of the IS output buffers increases from layer 1 to 3 due to the increase in the number of output channels.



(a) buffered WS

(b) buffered IS

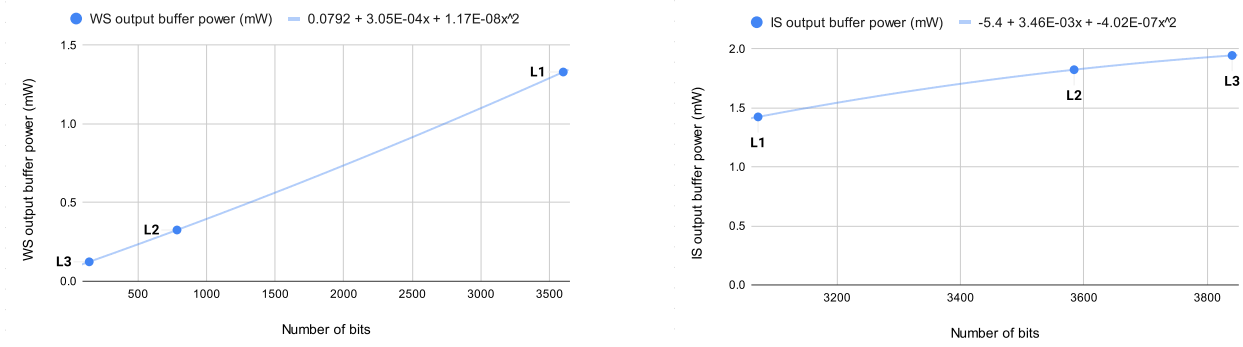
Figure 6.3: Output buffer area results obtained from the physical synthesis flow, for the three layers of Cifar10 CNN.

The interpolation of the area results are used to compute the output buffer area. Equations 6.12 and 6.13 compute the output buffer area for WS and IS, respectively.

$$WSOutputBuffArea = (10.4 \times NumBits) + 493 \tag{6.12}$$

$$ISOutputBuffArea = (10.5 \times NumBits) + 539 \tag{6.13}$$

The same interpolation method is applied to obtain the power dissipation due to the output buffers. However, each memory type has a interpolation equation, due to the latency access. Figure 6.4a and Figure 6.4b present the power results for the three layers of Cifar10 CNN using a SRAM memory. The buffered WS reduces the power from layer L1 to L3 due to the reduction in the OFMAP size. The buffered IS increases the power from layer L1 to L3 due to the increase in the number of output channels.



(a) buffered WS

(b) buffered IS

Figure 6.4: Output buffer power results obtained from the physical synthesis flow, for the three layers of Cifar10 CNN, using a SRAM memory type.

The interpolation of the power results is used to compute the output buffer power dissipation. Equations 6.14 and 6.15 compute the output buffer power dissipation for WS and IS using a SRAM, respectively. Equations 6.16 and 6.17 compute the output buffer power dissipation for WS and IS using a DRAM, respectively.

$$SRAM\_WSOutputBuffPower = 0.0792 + (0.000305 \times NumBits) + (0.0000000117 \times NumBits^2) \quad (6.14)$$

$$SRAM\_ISOutputBuffPower = -5.4 + (0.00346 \times NumBits) + (-0.000000402 \times NumBits^2) \quad (6.15)$$

$$DRAM\_WSOutputBuffPower = 0.0794 + (0.000245 \times NumBits) + (0.0000000109 \times NumBits^2) \quad (6.16)$$

$$DRAM\_ISOutputBuffPower = -7.98 + (0.00484 \times NumBits) + (-0.000000595 \times NumBits^2) \quad (6.17)$$

6.4 Results

This Section presents results obtained for the three DSE flows previously presented. As a case study, we adopt the CNN illustrated in Figure 6.5, implemented at TensorFlow. This CNN contains three convolutional layers and a fully-connected layer. TensorFlow executes the fully-connected layer, not accelerated in hardware. The number of filters per layer is 16, 32, and 64. The CNN implemented in the TensorFlow uses the Cifar10 dataset with a 32x32x3 (RGB) IFMAP.

After training, the obtained accuracy was 67%. The quantization method (Section 3.2) was verified using two shift values: 4 and 8. For a shift value equal to 4, it is possible to use 8-bit words at the inputs. However, this shift value reduces the accuracy to

44%. Using a shift value equal to 8 implies 16-bit words at the inputs. The obtained accuracy with 16-bit words at the inputs was 66.98%, a value 0.02% smaller than the one obtained in TensorFlow with float point values. For this reason, the hardware accelerators used in this Section adopt 16-bit words at the inputs.

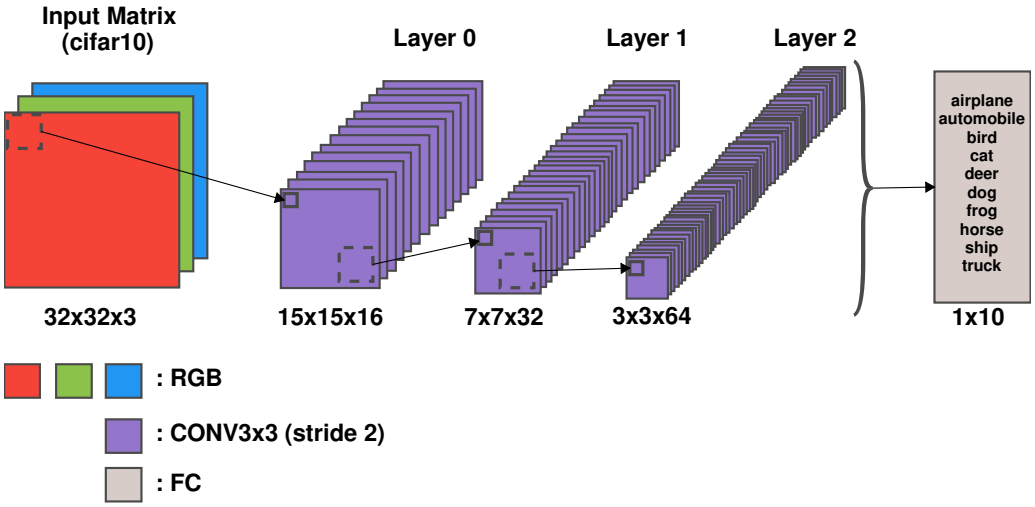


Figure 6.5: Cifar10 CNN.

Cadence Genus and Innovus tools were used for logic and physical synthesis, with 28nm technology and a frequency of 500MHz. The logic synthesis uses clock-gating to reduce the accelerator energy consumption. The power dissipation uses the VCD file generated after a post physical synthesis simulation and the Cadence Voltus tool.

The netlist simulation inputs are the Cifar10 CNN layers (Figure 6.5) extracted from TensorFlow (as showed in Figure 6.1). The first layer (**Layer 0**) uses the 32x32x3 IFMAP (RGB image from CIFAR10 dataset), 16 3x3 filters, stride 2, generating a 15x15x16 output. The **Layer 1** uses a 15x15x16 IFMAP (OFMAP from **Layer 0**), 32 3x3 filters, stride 2, generating a 7x7x32 output. The last convolution layer (**Layer 2**) uses a 7x7x32 IFMAP (OFMAP from **Layer 1**), 64 3x3 filters, stride 2, generating a 3x3x64 output. Thus, power values come from a real dataset and not synthetic values. The total energy is computed by multiplying the average power by the number of clock cycles required to execute a complete convolution.

The external memories modeling, SRAM and DRAM, adopts the Cacti-IO tool [Jouppi et al., 2014]. For a 28nm 64KB SRAM, Cacti-IO reports 0.01356nJ for reading operation and 0.01351nJ for writing operation. For a 64kB DRAM, 0.1633nJ for reading operation and 0.1662nJ for writing operation.

6.4.1 MAC-based DSE Flow Results

This Section evaluates the MAC-based DSE flow, using the physical synthesis flow as the reference. MAC-based estimation, routinely used in the literature [Tang and Xie,

2018, Parashar et al., 2019, Heidorn et al., 2020, Zhao et al., 2020, Cao et al., 2020], is the first evaluated. This setup is built to demonstrate that executing power and area estimation using only MACs or MACs plus registers does not produce accurate results.

Five tables present the results, one for each dataflow – Table 6.1 to Table 6.5. The "9X9 MAC" and "9X9 reg MAC" columns contain results for the MAC-based flow. The next column presents results for a given accelerator using the physical synthesis flow. The last two columns show the error induced by the MAC-based flow. The error is the percentage of a given result value against the physical synthesis flow.

Note that in the 5 tables the MAC-based flow results are the same, regardless of the dataflow. The values are the same because the MAC-based flow only considers the arithmetic core (i.e., the number of MACs) or this value plus input and output registers. The MAC-based flow does not consider the control logic (FSMs), internal registers, internal buffers, and logic to interconnect components.

Table 6.1: MAC-based and physical synthesis flows results for the WS accelerator.

WS	9x9 MAC	9x9 reg MAC	WS	error 9x9 MAC (%)	error 9x9 reg MAC (%)
area layer 0 (μm^2)	7,214.67	9,468.41	14,563.97	50.46	34.99
area layer 1 (μm^2)	7,214.67	9,468.41	15,037.57	52.02	37.03
area layer 2 (μm^2)	7,214.67	9,468.41	15,002.49	51.91	36.89
power layer 0 (mW)	0.57	1.16	0.95	39.99	22.13
power layer 1 (mW)	0.57	1.16	0.87	34.35	33.59
power layer 2 (mW)	0.57	1.16	0.88	35.32	31.64

Table 6.2: MAC-based and physical synthesis flows results for the buffered WS accelerator.

Buffered WS	9x9 MAC	9x9 reg MAC	WS buf	error 9x9 MAC (%)	error 9x9 reg MAC (%)
area layer 0 (μm^2)	7,214.67	9,468.41	13,399.05	46.16	29.34
area layer 1 (μm^2)	7,214.67	9,468.41	13,777.02	47.63	31.27
area layer 2 (μm^2)	7,214.67	9,468.41	13,762.66	47.58	31.20
power layer 0 (mW)	0.57	1.16	0.98	42.25	17.53
power layer 1 (mW)	0.57	1.16	0.92	38.06	26.05
power layer 2 (mW)	0.57	1.16	0.90	37.20	27.81

Table 6.3: MAC-based and physical synthesis flows results for the IS accelerator.

IS	9x9 MAC	9x9 reg MAC	IS	error 9x9 MAC (%)	error 9x9 reg MAC (%)
area layer 0 (μm^2)	7,214.67	9,468.41	15,425.33	53.23	38.62
area layer 1 (μm^2)	7,214.67	9,468.41	15,702.45	54.05	39.70
area layer 2 (μm^2)	7,214.67	9,468.41	15,696.24	54.04	39.68
power layer 0 (mW)	0.57	1.16	2.04	72.06	43.14
power layer 1 (mW)	0.57	1.16	1.92	70.31	39.58
power layer 2 (mW)	0.57	1.16	1.72	67.03	32.91

Area and power results are underestimated with the MAC-based flow. Area estimation considering only the arithmetic core is roughly 50% of the area obtained with the physical

Table 6.4: MAC-based and physical synthesis flows results for the buffered IS accelerator.

Buffered IS	9x9 MAC	9x9 reg MAC	IS buf	error 9x9 MAC (%)	error 9x9 reg MAC (%)
area layer 0 (μm^2)	7,214.67	9,468.41	13,886.36	48.04	31.82
area layer 1 (μm^2)	7,214.67	9,468.41	14,156.94	49.04	33.12
area layer 2 (μm^2)	7,214.67	9,468.41	14,162.16	49.06	33.14
power layer 0 (mW)	0.57	1.16	2.09	72.82	44.68
power layer 1 (mW)	0.57	1.16	1.90	70.09	39.14
power layer 2 (mW)	0.57	1.16	1.80	68.33	35.56

Table 6.5: MAC-based and physical synthesis flows results for the OS accelerator.

OS	9x9 MAC	9x9 reg MAC	OS	error 9x9 MAC (%)	error 9x9 reg MAC (%)
area layer 0 (μm^2)	7,214.67	9,468.41	14,938.34	51.70	36.62
area layer 1 (μm^2)	7,214.67	9,468.41	15,336.23	52.96	38.26
area layer 2 (μm^2)	7,214.67	9,468.41	15,753.20	54.20	39.90
power layer 0 (mW)	0.57	1.16	1.02	44.28	13.39
power layer 1 (mW)	0.57	1.16	1.03	44.71	12.51
power layer 2 (mW)	0.57	1.16	0.87	34.98	32.33

synthesis flow. Area estimation with the input and output registers with the arithmetic core results in errors between 30% and 40%. The power estimation error can vary from 12.51% (OS reg MAC) to 72.86% (IS MAC). The main reason explaining the differences observed in the power estimation is the switching activity. The MAC-based flow uses an average of the switching activity of MACs and registers, fixing this value for the estimation. The physical synthesis uses the switching activity of the whole circuit with actual data.

As mentioned before, Several works in the literature use the MAC-based method to estimate area and power. We demonstrated from the above results that estimating area and power using the number of arithmetic operators produces results far from the actual hardware result. Thus, we claim that methods such as the one based on physical synthesis or the analytic one, with results presented in the next section, provide reliable results, unlike the method based on counting arithmetic operations.

6.4.2 Analytic DSE Flow Results

This Section evaluates the proposed analytic DSE flow, which estimates, as shown below, the PPA values for CNN applications more fast and more accurate. The reference for the analytic DSE flow is the physical synthesis flow for the Cifar10 CNN layer 0. Thus, the analytic DSE flow should produce a small error when estimating layer 0 because the PPA of this layer is the basis to generate the analytic model equations.

Tables 6.6 to 6.12 summarize the results. Appendix B presents the analytic DSE flow results for all performance figures. Tables show results for area, performance, memory

accesses, power, and energy estimations. Each table presents one performance figure for the five dataflows, considering the physical (**synt. flow** columns) and analytic (**analytic**) flows. Column **|error %|** presents the absolute error - yellow, orange and red values represent absolute error below 5%, between 5% and 10%, and above 10%. Rows in the tables represent the memory type (SRAM or DRAM) and the layers (L0, L1, and L2).

Area Estimation

Table 6.6 presents results for area estimation. Bold values in Table 6.6 at |error %| column corresponds to the comparison with the reference layer (L0). As expected, dataflows without output buffer presented the same result for the synthesis and analytic flows. Dataflows with output buffers presented a small error (below 1%) due to the interpolation approach.

Table 6.6: Cifar10 CNN area analytic results.

AREA (μm^2). Orange: error above 5% and below 10%															
dataflow	**ws**			**ws buf**			**is**			**is buf**			**os**		
results	**synt. flow**	**analytic**	**| error % |**	**synt. flow**	**analytic**	**| error % |**	**synt. flow**	**analytic**	**| error % |**	**synt. flow**	**analytic**	**| error % |**	**synt. flow**	**analytic**	**| error % |**
L0	14,564	14,564	**0.00**	51,260	51,332	0.14	15,425	15,425	**0.00**	54,561	54,745	0.34	14,938	14,938	**0.00**
L1	15,038	14,564	3.15	22,405	22,046	1.60	15,702	15,425	1.76	52,165	52,057	0.21	15,336	14,938	2.59
L2	15,002	14,564	2.92	15,753	15,390	2.31	15,696	15,425	1.73	46,810	46,681	0.28	15,753	14,938	5.17

The area estimation for layers L1 and L2 stays below 4% for layers L1 and L2, excepting OS L2, with an error of 5.17%. The reason for explaining the error is the increased number of output channels compared to L0, which affects the control logic and counters. The OS dataflow does not have buffers, requiring a more complex circuitry to manage memory accesses.

It is worth highlighting that the IS implementations require an input buffer to store weights and bias values, which increase the accelerator area. The area for these buffers is not included in the area results since this buffer acts as a cache memory, requiring an external memory. We adopted this approach because it is need memory compilers to generate these buffers. The use of memory compilers is considered a future work. The IS for this CNN needs 7,168, 74,240, and 295,936 bits for layers L0, L1, and l2, respectively. Thus, in terms of total area, the IS dataflow is larger than the other ones, requiring further development to reduce this area, as split the weights in small samples to allow reduce the output buffer size, and not read and store all weight values in internal buffers.

The overall area estimation error stays below 6%, with an average of 1.85% (with the minimal value equals to 0.14%, and the maximum value equals to 5.17%) and a standard deviation of 1.51%.

Performance Estimation

Table 6.7 presents results for performance estimation, which uses Equations from Section 6.3.1. The performance results consider different memory latencies, 2 clock cycles for SRAM and 5 clock cycles for DRAM. The error observed in Table 6.7 occurs due to synchronization states, not included in the Equations. These synchronization states occur mainly at the end of a row, where buffers must be flushed to start a new one. Improvements in the equations to include these synchronization states in the estimations are needed to obtain smaller errors.

Table 6.7: Cifar10 CNN performance analytic results. SRAM access latency 2 clock cycles, DRAM access latency 5 clock cycles.

PERFORMANCE (clock cycles). Orange: error above 5% and below 10%															
dataflow	ws			ws buf			is			is buf			os		
results	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |
SRAM - L0	236,888	236,592	0.12	236,887	236,592	0.12	142,875	140,529	1.64	136,125	126,129	7.34	605,942	591,264	2.42
SRAM - L1	642,152	647,680	0.86	642,151	647,680	0.86	291,364	313,072	7.45	278,134	266,032	4.35	1,407,206	1,359,616	3.38
SRAM - L2	766,152	804,864	5.05	766,151	804,864	5.05	244,980	267,744	9.29	235,494	232,032	1.47	1,036,742	999,936	3.55
DRAM - L0	449,915	451,584	0.37	449,911	451,584	0.37	182,694	170,898	6.46	155,694	145,698	6.42	1,190,102	1,175,328	1.24
DRAM - L1	1,218,251	1,245,184	2.21	1,218,247	1,245,184	2.21	363,202	373,248	2.77	313,222	301,120	3.86	2,763,878	2,716,096	1.73
DRAM - L2	1,448,331	1,572,864	8.60	1,448,327	1,572,864	8.60	335,586	349,440	4.13	298,758	295,296	1.16	2,035,910	1,998,720	1.83

The overall performance estimation error stays below 9%, with an average of 3.50% (with the minimal value equals to 0.12%, and the maximum value equals to 9.29%) and a standard deviation of 2.78%.

Memory Accesses Estimation

Table 6.8 presents results for IFMAP memory access, which is read-only. The error is more significant in the IS dataflows in layers L0 and L1. The reason is similar to the error observed in the performance estimation, where there are synchronization states, mainly in the exchange of rows. At the end of a row, there are invalid reads to avoid increasing the complexity of the FSM. Thus, the memory CE (chip enable) is active for some clock cycles, inducing these invalid reads. If the memory latency is small (2 cycles for SRAM), more invalid reads may occur, while this effect is masked for higher latencies (5 cycles for DRAM). For this reason, this error is higher in IS L0 and IS L1 due to the larger IFMAP size.

The overall IFMAP reading estimation error stays below 10%, with an average of 1.22% (with the minimal value equals to 0.01%, and the maximum value equals to 9.38%) and a standard deviation of 2.73%.

Tables 6.9 and 6.10 presents results for OFMAP memory access. The estimation model for OFMAP memory presents a 0% error, which means the analytic model correctly captured the OFMAP memory accesses. Note that the buffered (WS and IS buf) and OS dataflow do not need to read partial values from the OFMAP. Only WS and IS dataflows store partial values in the OFMAP, which need to be read to compute the final values.

Table 6.8: Cifar10 CNN IFMAP read accesses results.

Reads accesses: IFMAP. Orange: error above 5% and below 10%															
dataflow	ws			ws buf			is			is buf			os		
results	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |
SRAM - L0	71,056	71,056	0.00	71,056	71,056	0.00	7,198	6,523	9.38	7,198	6,523	9.38	194,735	194,704	0.02
SRAM - L1	192,544	192,544	0.00	192,544	192,544	0.00	12,480	11,696	6.28	12,480	11,696	6.28	452,255	452,192	0.01
SRAM - L2	229,440	229,440	0.00	229,440	229,440	0.00	21,376	21,088	1.35	21,376	21,088	1.35	333,119	332,992	0.04
DRAM - L0	71,008	71,056	0.07	71,008	71,056	0.07	6,523	6,523	0.00	6,523	6,523	0.00	194,720	194,704	0.01
DRAM - L1	192,032	192,544	0.27	192,032	192,544	0.27	11,696	11,696	0.00	11,696	11,696	0.00	452,224	452,192	0.01
DRAM - L2	227,392	229,440	0.90	227,392	229,440	0.90	21,088	21,088	0.00	21,088	21,088	0.00	333,056	332,992	0.02

Table 6.9: Cifar10 CNN OFMAP read accesses results.

Reads accesses: OFMAP. Orange: error above 5% and below 10%															
dataflow	ws			ws buf			is			is buf			os		
results	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |
L0	7,200	7,200	0.00	0.00	0.00	0.00	7,200	7,200	0.00	0.00	0.00	0.00	0.00	0.00	0.00
L1	23,520	23,520	0.00	0.00	0.00	0.00	23,520	23,520	0.00	0.00	0.00	0.00	0.00	0.00	0.00
L2	17,856	17,856	0.00	0.00	0.00	0.00	17,856	17,856	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 6.10: Cifar10 CNN OFMAP write accesses results.

Writes accesses: OFMAP. Orange: error above 5% and below 10%															
dataflow	ws			ws buf			is			is buf			os		
results	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |
L0	10,800	10,800	0.00	3,600	3,600	0.00	10,800	10,800	0.00	3,600	3,600	0.00	3,600	3,600	0.00
L1	25,088	25,088	0.00	1,568	1,568	0.00	25,088	25,088	0.00	1,568	1,568	0.00	1,568	1,568	0.00
L2	18,432	18,432	0.00	576	576	0.00	18,432	18,432	0.00	576	576	0.00	576	576	0.00

Accelerator Power Estimation

Table 6.11 shows the results for power estimation. Note that this table only considers the accelerator power. Similar to the area estimation, the L0 is the reference. WS, IS and OS present an absolute error equal to 0% (bold values on Table 6.11), while the buffered dataflows presented an error due to the interpolation approach.

Table 6.11: Cifar10 CNN power analytic results.

POWER (mW). Orange: error above 5% and below 10%, red: error above 10%															
dataflow	ws			ws buf			is			is buf			os		
results	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |	synt. flow	analytic	| error % |
SRAM - L0	0.95	0.95	**0.00**	2.32	2.32	0.05	2.04	2.04	**0.00**	4.04	4.06	0.34	1.04	1.04	**0.00**
SRAM - L1	0.87	0.95	9.39	1.25	1.31	5.34	1.92	2.04	6.25	3.73	3.93	5.47	1.02	1.04	1.47
SRAM - L2	0.88	0.95	7.78	1.03	1.11	7.70	1.73	2.04	17.99	3.22	3.53	9.56	1.03	1.04	0.68
DRAM - L0	0.74	0.74	**0.00**	1.91	2.13	0.07	1.79	1.79	**0.00**	3.84	3.96	0.08	0.88	0.88	**0.00**
DRAM - L1	0.71	0.74	4.32	1.06	1.13	6.26	1.70	1.79	5.41	3.54	3.84	5.48	0.87	0.88	0.87
DRAM - L2	0.71	0.74	4.23	0.87	0.93	6.79	1.45	1.79	23.55	2.87	3.44	14.46	0.88	0.88	0.47

The power estimation has a higher error than the area and performance estimation. Two reasons explain this mismatch:

- The power reference is layer L0, with its switching activity. The switching activity of other layers is different, affecting the power estimation. The switching activity is a function of the input data, not being possible to capture it in the analytic model.
- The buffered dataflows has an error induced by the interpolation method.

Despite the larger errors observed mainly when communicating with DRAM memory, the overall power estimation presents an average error of 7.00% (with the minimal value equals to 0.05%, and the maximum value equals to 23.55%), and a standard deviation of 6.21%.

Total Energy Estimation

Table 6.12 presents results for the energy estimation, considering the accelerators and the memory accesses. Memory accesses are responsible for most of the consumed energy. According to [Chen et al., 2016b], the memory energy can spent 200 times more energy than the accelerator array. In our experiments we observed in OS dataflow the memory consuming 20 times more energy than the accelerator (Table 5.3). Thus, Table 6.8 is the reference for the expected error. Two situations occur:

- The IS dataflow presents a small energy error estimation because the number of OFMAP accesses is higher than the IFMAP readings (where the estimation presents errors), resulting in a small energy estimation error, below 3%.
- The IS buf dataflow makes more IFMAP readings (with an estimation error equal to 9.38%) than OFMAP writes. The result is a higher energy estimation error.

Table 6.12: Cifar10 CNN energy analytic results.

TOTAL ENERGY (nJ) - accellerator and memories. Orange: error above 5% and below 10%															
dataflow	**ws**			**ws buf**			**is**			**is buf**			**os**		
results	**synt. flow**	**analytic**	**| error % |**	**synt. flow**	**analytic**	**| error % |**	**synt. flow**	**analytic**	**| error % |**	**synt. flow**	**analytic**	**| error % |**	**synt. flow**	**analytic**	**| error % |**
SRAM - L0	1,207	1,207	0.00	1,013	1,013	0.00	342	332	2.69	147	138	6.26	2,690	2,690	0.02
SRAM - L1	3,270	3,270	0.00	2,633	2,633	0.00	828	817	1.28	191	181	5.55	6,155	6,155	0.01
SRAM - L2	3,603	3,603	0.00	3,120	3,120	0.00	782	778	0.49	298	295	1.29	4,526	4,524	0.04
DRAM - L0	14,569	14,577	0.05	12,196	12,204	0.07	4,037	4,037	0.00	1,665	1,664	0.00	32,401	32,398	0.01
DRAM - L1	39,376	39,460	0.21	31,624	31,708	0.26	9,923	9,924	0.00	2,172	2,172	0.00	74,120	74,115	0.01
DRAM - L2	43,119	43,454	0.78	37,234	37,569	0.90	9,426	9,426	0.00	3,541	3,541	0.00	54,492	54,481	0.02

The overall energy estimation error stays below 7%, with an average of 0.66% (with the minimal value equals to 0.12%, and the maximum value equals to 28.65%) and a standard deviation of 1.54%.

Analytic Model Compared to the State-of-the-art

Table 6.13 compares the analytic model results with results available in the literature. Power and energy consider only the accelerator, not the memories. The Table presents for each work the min/max/average error, when it is available. In this table:

- red values: average error is higher than the ones obtained with our analytic model;
- green values: average error is lower than the ones obtained with our analytic model.

Table 6.13: Analytic and state-of-the-art result errors comparison.

	Aladdin [Shao et al., 2014]			MLPAT [Tang and Xie, 2018]			Accelergy [Wu et al., 2019]			STONNE [Muñoz-Martínez et al., 2020]			This Thesis		
error	**min**	**max**	**avg**	**min**	**max**	**avg**	**min**	**max**	**avg**	**min**	**max**	**avg**	**min**	**max**	**avg**
area	4.30	10.60	6.60	—	—	5.00	—	—	—	—	—	—	0.14	5.17	1.85
performance	0.20	2.60	0.90	—	—	—	—	—	—	11.00	19.00	15.00	0.12	9.29	3.50
acc power	2.30	8.30	4.90	—	—	10.00	—	—	—	—	—	—	0.05	23.55	7.00
acc energy	—	—	—	—	—	—	—	—	5.00	—	—	—	0.12	28.65	8.11

Few works in the literature present a comprehensive estimation as the method proposed in this Thesis. MLPAT and Accerlergy present a limited evaluation of area and energy. Stone evaluates only performance with higher errors compared to our method. Aladdin presents the most complete evaluation compared to the other methods. However, the proposed method has an area evaluation more accurate than Aladdin, while the evaluation of the other metrics has errors of the same order.

As a conclusion, the proposed flow presents a more comprehensive evaluation of more metrics with lower errors, and provides a large set of estimates for each dataflow, as shown in Appendix B.

Analytic Model Final Remarks

The proposed analytic flow enabled an accurate PPA estimation. Table 6.14 summarizes the results. Improvements may be done in the IFMAP memory readings estimations and in the power interpolation approach.

Table 6.14: Analytic approach summary results.

	Avg. Error (%)	**Std. Dev. (%)**
Area	1.85	1.51
Performance	3.50	2.78
IFMAP Read	1.22	2.73
OFMAP Read/Write	0.00	0.00
Acc. Power	7.00	6.21
Total Energy	0.66	4.34

Besides the PPA accuracy, the method is *fast*. Using the analytic approach, a DSE analysis for a given CNN, like the one presented in the tables, took 0.025 seconds. Physical synthesis of one accelerator take 45 minutes (Intel i9-7940X@3.10GHz, 28 cores, 64 GB memory). Considering all layers, and channels, the same DSE obtained with the analytic approach would take several hours to be performed with the physical synthesis flow.

7. CONCLUSION AND FUTURE WORK

This Thesis proposed the following statement: *It is possible to execute fast and accurate design space exploration (DSE) for machine learning accelerators, considering different CNN architectures models using standard frameworks. The DSE flow must be comprehensive in terms of power, performance, and area (PPA) estimation. Providing PPA enables the designer to select the most relevant parameters (according to the literature) to design a hardware accelerator*.

The first part of the statement is: *It is possible to execute fast and accurate design space exploration (DSE) for machine learning accelerators, considering different CNN architectures models using standard frameworks*. We proposed two approaches for DSE. The first adopts a system simulator (URSA), which is cycle-accurate and uses a high-level language to describe the hardware abstractly. The second is a fast and accurate DSE, using an analytic approach, which does not need the abstract model to estimate PPA.

This part of the statement fulfilled specific goals 1 and 6:

- *CNN framework integration* (Chapter 3): we adopted TensorFlow as a front-end to implement a CNN application. A quantization method was validated in TensorFlow, avoiding the use of float-point hardware and reducing the area and power of the accelerators.

- *DSE method* (Chapters 3 and 6): the first DSE method used TensorFlow and the URSA system simulator. Its advantage is the abstract model, but, at the same time, this model can be complex and does not reflect accurately an optimized hardware. The second DSE method used only TensorFlow, coupled with physical synthesis results. Besides the DSE results regarding PPA metrics, the second method is faster than a classic physical synthesis flow. Also, errors presented by the second DSE method have the same magnitude order as the literature. Still, they are more reliable once the obtained results are based on an entire convolution, not only on basic components such as the number of MACs. Also, the proposed DSE flows are more comprehensive compared to the literature, once they deliver a complete PPA analyses with more architectural parameters, compared to the literature, as showed in Table 6.14.

The second part of the statement is: *The DSE flow must be comprehensive in terms of power, performance, and area (PPA) estimation.*. The proposed methods presented a complete PPA analysis based on values of actual CNN applications.

This part of the statement fulfilled specific goals 2, 4 and 5:

- *CNN hardware accelerator design* (Chapter 4): we started using the open-source NVDLA hardware modules to build accelerators. However, NVDLA showed impor-

tant silicon area costs and limited flexibility to explore different dataflows. Thus, a set of accelerators were proposed, composing an accelerator library.

- *CNN hardware accelerator physical synthesis* (Chapter 6): we adopted a classic synthesis flow using industrial tools (Cadence and Mentor), executing logic and physical synthesis. This flow supports different technology nodes, such as 65nm and 28nm, and generates accurate hardware estimations.
- *PPA extraction method* (Chapter 6): the accelerators' simulation uses data (weights and IFMAP) from TensorFlow, generating a switching activity corresponding to actual CNNs. This method ensures accurate power (**P**) estimation. The post-synthesis simulation generates the performance (**P**) estimation, considering the effect of parasitic capacitances. The physical synthesis gives the accelerator area (**A**).

Accelerators source code and synthesis scripts are available at the following GitHub repository: https://github.com/leorezende93/acc_dse_env.

The third part of the statement is: *Providing PPA enables the designer to select the most relevant parameters (according to the literature) to design a hardware accelerator*. By modeling the CNN application in TensorFlow, the designer may select the accelerator that meets the design constraints.

This part of the statement fulfilled specific goal 3:

- *Comparison method* (Chapter 5): we showed it is possible to use the proposed methods to compare different accelerator types. This comparison can be performed using the physical synthesis flow or analytically. Thus, this Thesis presents a fast DSE method, which is more comprehensive and more accurate when compared to the state-of-the-art.

Also, this Thesis comprises all the steps of a hardware/software machine learning application development, which are:

- Software development: the use of TensorFlow to implement a CNN application;
- Hardware architecture design: define hardware architecture and behavior, as style and dataflow;
- Hardware synthesis: define the hardware constraints, as clock frequency, and perform the physical synthesis;
- PPA analyses: analyses the obtained results of power, performance, and area.

Thus, the comprised steps addressed in this Thesis allow software developers to quickly estimate the hardware resources used by the CNN application without needing a prototyping platform like FPGA or the manufactured chip.

To conclude, the above analysis demonstrated that it is possible to propose methods for fast, fair, and accurate design space exploration related to hardware accelerators for CNNs. This Thesis advanced the state-of-the-art by offering techniques to generate a comprehensive PPA evaluation, integrating front-end frameworks (such as TensorFlow) to a hardware back-end design flow. The TensorFlow front-end generates the hardware back-end data for simulation, while the back-end provides the analytical model to the front-end.

7.1 Future Work

It is possible to group future works into different research topics.

System Level DSE. Extend the use of system simulators to perform DSE regarding an entire system composed by CPUs, DMA, and CNN accelerators.

As mentioned on Chapter 3, system simulators, such as URSA, help in estimating PPA values of an entire system. This Thesis does not cover DSE regarding an entire system. Thus, one of the future works is to extend the proposed frameworks to cover systems composed of CPUs, DMA, and CNN accelerators, for example, as illustrated in Figure 7.1. Also, it is possible to combine the system simulator with the analytic flow to improve the accuracy of the PPA results, like clock cycles estimation.

Thus, we proposed the following future works:

- Integrate the DSE analytic flow with a system simulator to deliver accurate PPA results with simulation results as convolution engine behavior. To make this integration, it is possible to use simulators like Gem5 [Gem5, 2022], which allow the reuse of hardware blocks, such as microprocessors, memories, and bus architectures.
- Software and hardware accelerators integration. Integrate the accelerator with microprocessors, and develop APIs to access the accelerators.

Accelerator Design. Extend the set of accelerators and functions implemented in hardware.

- Implementation of other dataflow types: the literature shows other dataflows like Row Stationary (RS), No Local Reuse (NRL), and Fine-Grained (FG) [Moolchandani et al., 2021, Xiang et al., 2018]. The implementation of these dataflows allows to extend the comparison proposed on Chapter 5, to analyze their trade-offs compared to Weight Stationary (WS), Input Stationary (IS), and Output Stationary (OS);
- Implementation of larger accelerators arrangement: the 3x3 matrix used in this work is an initial step for DSE. We can extend the analyzes for larger arrays as 16x16, like in

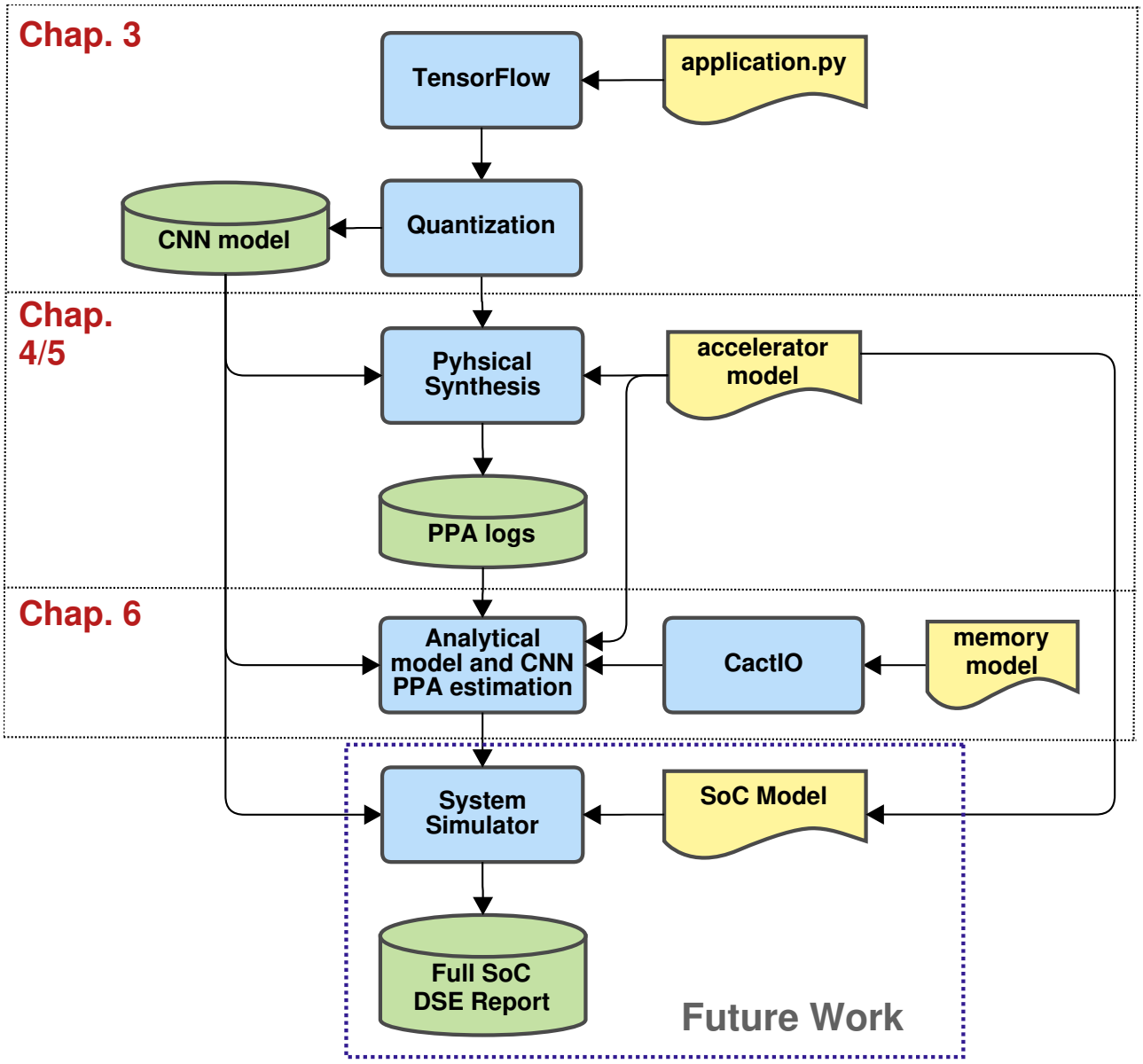


Figure 7.1: System Level DSE Flow.

[Udupa et al., 2020]. Larger arrays allow to improve the analysis of the array parallelism and the array utilization in terms of array percentage;

- Integration of Imagenet dataset on DSE flow: thus, it is possible to simulate the accelerators using more complex CNNs, like AlexNet and VGG16. Also, it is required to implement hardware pooling functions, like max-pooling and average pooling, to execute these CNNs.
- A benchmark approach to compare hardware accelerators: build an accelerator repository that allows project decisions regarding specific targets, such as low power, high throughput, and small area.

Accelerator Optimization. Optimize the accelerator design using low power techniques, pruning, quantization, and memory types.

- Application of low power techniques: evaluate, e.g., approximate computing [Armeniakos et al., 2022]. Also, other CNNs optimization techniques can be applied, like pruning [Bavikadi et al., 2022], and different types of CNN, like the all-convolutional neural networks [Benevenuti et al., 2021]. All-convolutional neural networks are candidates to reduce hardware area, once they do not present a fully-connected layer, which is the more expensive network component;
- Explore the use of different memories in the same project. For example, build accelerators that use both SRAM and DRAM. Also, explore memory hierarchies, like the use of cache memories.

Accelerator Prototyping. Prototype the proposed accelerators in FPGAs, considering the entire CNN.

- Analyzes and implementation of an entire CNN accelerated in hardware: this work requires analyzing how to connect the output from a layer to the input of the next layer of a CNN application using the hardware accelerators. These analyses include the use of memories, buffers, and memory hierarchy to connect these layers;
- FPGA prototyping of hardware accelerators: this work allows verification of the accelerators at a circuit level. Besides, prototyping and integrating the accelerators with microprocessors to explore analyzes of an entire system.

7.2 Summary of the publications produced during the Thesis

The following papers related to the Thesis were published:

- A TensorFlow and System Simulator Integration Approach to Estimate Hardware Metrics of Convolution Accelerators, L. R. Juracy, M. T. Moreira, A. M. Amory, and F. G. Moraes: published in LASCAS 2021 (Qualis B4);
- A High-level Modeling Framework for Estimating Hardware Metrics of CNN Accelerators, L. R. Juracy, M. T. Moreira, A. M. Amory, A., A. F. Hampel, and F. G. Moraes: published in TCASI 2021 (Qualis A1).

REFERENCES

[Ahmad and Pasha, 2020] Ahmad, A. and Pasha, M. A. (2020). FFConv: an FPGA-based accelerator for fast convolution layers in convolutional neural networks. *ACM Transactions on Embedded Computing Systems*, 19(2):1–24.

[Al-Jawfi, 2009] Al-Jawfi, R. (2009). Handwriting Arabic character recognition LeNet using neural network. *International Arab Journal of Information Technology*, 6(3):304–309.

[Alibaba, 2019] Alibaba (2019). Alibaba Hanguang 800. Source: https://techcrunch.com/2019/09/24/alibaba-unveils-hanguang-800-an-ai-inference-chip-it-says-significantly-increases-the-speed-of-machine-learning-tasks/, May 2022.

[Alom et al., 2018] Alom, M. Z., Taha, T. M., Yakopcic, C., Westberg, S., Sidike, P., Nasrin, M. S., Van Esesn, B. C., Awwal, A. A. S., and Asari, V. K. (2018). The history began from alexnet: A comprehensive survey on deep learning approaches. *Computing Research Repository*, abs/1803.01164(1):1–39.

[Amazon, 2018] Amazon (2018). AWS Inferentia. Source: https://aws.amazon.com/about-aws/whats-new/2018/11/announcing-amazon-inferentia-machine-learning-inference-microchip/, May 2022.

[Andri et al., 2017] Andri, R., Cavigelli, L., Rossi, D., and Benini, L. (2017). YodaNN: An architecture for ultralow power binary-weight CNN acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):48–60.

[Apple, 2022] Apple (2022). iPhone 11. Source: https://www.apple.com/iphone-11/, May 2022.

[Armeniakos et al., 2022] Armeniakos, G., Zervakis, G., Soudris, D., and Henkel, J. (2022). Hardware Approximate Techniques for Deep Neural Network Accelerators: A Survey. *ACM Computing Surveys*, *preprint*:1–36.

[Asanovic et al., 2016] Asanovic, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., et al. (2016). The rocket chip generator. Technical report, University of California. 11p.

[Bai et al., 2020] Bai, L., Lyu, Y., and Huang, X. (2020). A unified hardware architecture for convolutions and deconvolutions in CNN. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5.

[Baskin et al., 2021] Baskin, C., Liss, N., Schwartz, E., Zheltonozhskii, E., Giryes, R., Bronstein, A. M., and Mendelson, A. (2021). Uniq: Uniform noise injection for non-uniform quantization of neural networks. *ACM Transactions on Computer Systems*, 37(1-4):1–15.

[Bavikadi et al., 2022] Bavikadi, S., Dhavlle, A., Ganguly, A., Haridass, A., Hendy, H., Merkel, C., Reddi, V. J., Sutradhar, P. R., Joseph, A., and Dinakarrao, S. M. P. (2022). A Survey on Machine Learning Accelerators and Evolutionary Hardware Platforms. *IEEE Design & Test*, 39(3):91–116.

[Benevenuti et al., 2021] Benevenuti, F., Kastensmidt, F. L., de Oliveira, Á. B., Added, N., de Aguiar, V. Â. P., Medina, N. H., and Guazzelli, M. A. (2021). Robust Convolutional Neural Networks in SRAM-based FPGAs: a Case Study in Image Classification. *Journal of Integrated Circuits and Systems*, 16(2):1–12.

[Caffe, 2022] Caffe (2022). Caffe. Source: https://caffe.berkeleyvision.org/, May 2022.

[Cao et al., 2020] Cao, S., Deng, W., Bao, Z., Xue, C., Xu, S., and Zhang, S. (2020). SimuNN: A Pre-RTL Inference, Simulation and Evaluation Framework for Neural Networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(2):217–230.

[Cerebras, 2022] Cerebras (2022). Cerebras CS-1. Source: https://www.cerebras.net/technology/, May 2022.

[Chen et al., 2014] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. (2014). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284.

[Chen et al., 2020] Chen, X., Han, Y., and Wang, Y. (2020). Communication Lower Bound in Convolution Accelerators. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 529–541.

[Chen et al., 2016a] Chen, Y.-H., Emer, J., and Sze, V. (2016a). Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379.

[Chen et al., 2016b] Chen, Y.-H., Krishna, T., Emer, J. S., and Sze, V. (2016b). Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-state Circuits*, 52(1):127–138.

[Chen et al., 2019] Chen, Y.-H., Yang, T.-J., Emer, J., and Sze, V. (2019). Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308.

[Courbariaux et al., 2016] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *Computing Research Repository*, abs/1602.02830(1):1–11.

[CS231n, 2022] CS231n (2022). Convolutional Neural Networks (CNNs / ConvNets). Source: https://cs231n.github.io/convolutional-networks/, May 2022.

[Dally et al., 2020] Dally, W. J., Turakhia, Y., and Han, S. (2020). Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57.

[Das et al., 2020] Das, S., Roy, A., Chandrasekharan, K. K., Deshwal, A., and Lee, S. (2020). A Systolic Dataflow Based Accelerator for CNNs. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5.

[Digital, 2022] Digital, W. (2022). Western Digital Machine Learning Accelerator. Source: https://link.westerndigital.com/welcome/mcs-bulletin/mcs-bulletin-events/machine-learning-accelerator.html?_ga=2.249821190.143199995.1570669759-1671111829.1570669759, May 2022.

[Domingues, 2020] Domingues, A. R. P. (2020). ORCA: A Self-Adaptive, Multiprocessor System-On-Chip Platform. Master's thesis, PUCRS. 112p.

[Du et al., 2017] Du, L., Du, Y., Li, Y., Su, J., Kuan, Y.-C., Liu, C.-C., and Chang, M.-C. F. (2017). A reconfigurable streaming deep convolutional neural network accelerator for Internet of Things. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(1):198–208.

[Du et al., 2015] Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y., and Temam, O. (2015). ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the ACM International Symposium on Computer Architecture (ISCA)*, pages 92–104.

[Facebook, 2022a] Facebook (2022a). Facebook Horizon. Source: https://www.oculus.com/horizon-worlds/, May 2022.

[Facebook, 2022b] Facebook (2022b). Facebook Kings Canyon. Source: https://engineering.fb.com/data-center-engineering/accelerating-infrastructure/, May 2022.

[Ferianc et al., 2021] Ferianc, M., Fan, H., Manocha, D., Zhou, H., Liu, S., Niu, X., and Luk, W. (2021). Improving Performance Estimation for Design Space Exploration for Convolutional Neural Network Accelerators. *MDPI Electronics*, 10(4):1–14.

[Fujitsu, 2018] Fujitsu (2018). Fujitsu Deep Learning Unit. Source: https://www.fujitsu.com/global/Images/deep-learning-unit.pdf, May 2022.

[Fujiwara et al., 2013] Fujiwara, H., Yabuuchi, M., Morimoto, M., Tanaka, K., Tanaka, M., Maeda, N., Tsukamoto, Y., and Nii, K. (2013). A 20nm 0.6 V 2.1 μW/MHz 128kb SRAM with no half select issue by interleave wordline and hierarchical bitline scheme. In *Proceedings of the IEEE Symposium on VLSI Circuits (VLSI)*, pages 118–119.

[Gem5, 2022] Gem5 (2022). Gem5. Source: http://gem5.org/, May 2022.

[Genc et al., 2021] Genc, H., Kim, S., Amid, A., Haj-Ali, A., Iyer, V., Prakash, P., Zhao, J., Grubb, D., Liew, H., Mao, H., et al. (2021). Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 769–774.

[Gerogiannis et al., 2022] Gerogiannis, G., Birbas, M., Leftheriotis, A., Mylonas, E., Tzanis, N., and Birbas, A. (2022). Deep Reinforcement Learning Acceleration for Real-Time Edge Computing Mixed Integer Programming Problems. *IEEE Access*, 10(1):18526–18543.

[Giri et al., 2020] Giri, D., Chiu, K., Guglielmo, G. D., Mantovani, P., and Carloni, L. P. (2020). ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning. In *Proceedings of the IEEE Design, Automation Test in Europe Conference (DATE)*, pages 1049–1054.

[Gokhale et al., 2014] Gokhale, V., Jin, J., Dundar, A., Martini, B., and Culurciello, E. (2014). A 240 g-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 682–687.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. 781p.

[Google, 2022a] Google (2022a). Cloud TPU. Source: https://cloud.google.com/tpu/, May 2022.

[Google, 2022b] Google (2022b). Google Assistant, your own personal Google. Source: https://assistant.google.com, May 2022.

[Haine et al., 2017] Haine, T., Nguyen, Q.-K., Stas, F., Moreau, L., Flandre, D., and Bol, D. (2017). An 80-MHz 0.4 V ULV SRAM macro in 28nm FDSOI achieving 28-fJ/bit access energy with a ULP bitcell and on-chip adaptive back bias generation. In *Proceedings of the IEEE European Solid State Circuits Conference (ESSCIRC)*, pages 312–315.

[Hao et al., 2019] Hao, C., Zhang, X., Li, Y., Huang, S., Xiong, J., Rupnow, K., Hwu, W.-m., and Chen, D. (2019). FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1–6.

[Haykin, 2009] Haykin, S. S. (2009). *Neural networks and learning machines*. Pearson Education, third edition. 906p.

[Heidorn et al., 2020] Heidorn, C., Hannig, F., and Teich, J. (2020). Design space exploration for layer-parallel execution of convolutional neural networks on CGRAs. In *Proceedings of the ACM SIGBED/EDAA Software and Compilers for Embedded Systems (SCOPES)*, pages 26–31.

[Hsiao and Chang, 2020] Hsiao, S.-F. and Chang, H.-J. (2020). Sparsity-Aware Deep Learning Accelerator Design Supporting CNN and LSTM Operations. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4.

[Hsiao et al., 2020] Hsiao, S.-F., Chen, K.-C., Lin, C.-C., Chang, H.-J., and Tsai, B.-C. (2020). Design of a Sparsity-Aware Reconfigurable Deep Learning Accelerator Supporting Various Types of Operations. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(3):376–387.

[Huang et al., 2021] Huang, B., Huan, Y., Chu, H., Xu, J., Liu, L., Zheng, L., and Zou, Z. (2021). IECA: An In-Execution Configuration CNN Accelerator With 30.55 GOPS/mm^2 Area Efficiency. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(11):4672–4685.

[Huawei, 2019] Huawei (2019). Huawei Ascend 910. Source: https://www.huawei.com/en/press-events/news/2019/8/huawei-ascend-910-most-powerful-ai-processor, May 2022.

[IBM, 2022] IBM (2022). IBM Watson. Source: https://www.ibm.com/us-en/marketplace/deep-learning-platform, May 2022.

[Intel, 2022] Intel (2022). Intel Nervana. Source: https://www.intel.com.br/content/www/br/pt/analytics/artificial-intelligence/overview.html, May 2022.

[Jiao et al., 2020] Jiao, Y., Han, L., Jin, R., Su, Y.-J., Ho, C., Yin, L., Li, Y., Chen, L., Chen, Z., Liu, L., et al. (2020). 7.2 A 12nm Programmable Convolution-Efficient Neural-Processing-Unit Chip Achieving 825TOPS. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, pages 136–140.

[Jouppi et al., 2014] Jouppi, N. P., Kahng, A. B., Muralimanohar, N., and Srinivas, V. (2014). Cacti-IO: Cacti with off-chip power-area-timing models. *IEEE Transactions on Very Large Scale Integration Systems*, 23(7):1254–1267.

[Juracy et al., 2021a] Juracy, L. R., Moreira, M. T., Amory, A. M., and Moraes, F. G. (2021a). A TensorFlow and System Simulator Integration Approach to Estimate Hardware Metrics of Convolution Accelerators. In *Proceedings of the IEEE Latin America Symposium on Circuits and System (LASCAS)*, pages 217–230.

[Juracy et al., 2021b] Juracy, L. R., Moreira, M. T., de Morais Amory, A., Hampel, A. F., and Moraes, F. G. (2021b). A High-Level Modeling Framework for Estimating Hardware Metrics of CNN Accelerators. *IEEE Transactions on Circuits and Systems – I*, 68(11):4783–4795.

[Karbachevsky et al., 2021] Karbachevsky, A., Baskin, C., Zheltonozhskii, E., Yermolin, Y., Gabbay, F., Bronstein, A. M., and Mendelson, A. (2021). Early-stage neural network hardware performance analysis. *MDPI Sustainability*, 13(2):1–20.

[Keras, 2022] Keras (2022). Layer activation functions. Source: https://keras.io/api/layers/activations/, May 2022.

[Kim et al., 2020] Kim, S., Wang, J., Seo, Y., Lee, S., Park, Y., Park, S., and Park, C. S. (2020). Transaction-level Model Simulator for Communication-Limited Accelerators. *Computing Research Repository*, abs/2007.14897(1):1–11.

[Krizhevsky et al., 2017] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90.

[Kwon et al., 2019] Kwon, H., Chatarasi, P., Pellauer, M., Parashar, A., Sarkar, V., and Krishna, T. (2019). Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 754–768.

[Kwon et al., 2018a] Kwon, H., Pellauer, M., and Krishna, T. (2018a). Maestro: An open-source infrastructure for modeling dataflows within deep learning accelerators. *Computing Research Repository*, abs/1805.02566(1):1–5.

[Kwon et al., 2018b] Kwon, H., Samajdar, A., and Krishna, T. (2018b). Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM Special Interest Group on Programming Languages Notices*, 53(2):461–475.

[Li et al., 2019] Li, H., Bhargav, M., Whatmough, P. N., and Wong, H.-S. P. (2019). On-chip memory technology design space explorations for mobile deep neural network accelerators. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1–6.

[Lin and Arslan, 2021] Lin, W. and Arslan, T. (2021). A Column Streaming-Based Convolution Engine and Mapping Algorithm for CNN-based Edge AI accelerators. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–6.

[Liu et al., 2020a] Liu, B., Chen, X., Han, Y., Wang, Y., Li, J., Xu, H., and Li, X. (2020a). Search-free Accelerator for Sparse Convolutional Neural Networks. In *Proceedings of the ACM/IEEE Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 524–529.

[Liu et al., 2020b] Liu, B., Chen, X., Han, Y., and Xu, H. (2020b). Swallow: A Versatile Accelerator for Sparse Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4881–4893.

[Lu et al., 2017] Lu, W., Yan, G., Li, J., Gong, S., Han, Y., and Li, X. (2017). Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564.

[Manasi and Sapatnekar, 2021] Manasi, S. D. and Sapatnekar, S. S. (2021). DeepOpt: Optimized scheduling of CNN workloads for ASIC-based systolic deep learning accelerators. In *Proceedings of the ACM/IEEE Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 235–241.

[Mediatek, 2022] Mediatek (2022). Mediatek APU. Source: https://www.mediatek.com/technology/artificial-intelligence, May 2022.

[Microsoft, 2022] Microsoft (2022). Project Brainwave. Source: https://www.microsoft.com/en-us/research/project/project-brainwave/, May 2022.

[Moolchandani et al., 2021] Moolchandani, D., Kumar, A., and Sarangi, S. R. (2021). Accelerating CNN inference on ASICs: A survey. *Journal of Systems Architecture*, 113(1):1–26.

[Muñoz-Martínez et al., 2020] Muñoz-Martínez, F., Abellán, J. L., Acacio, M. E., and Krishna, T. (2020). STONNE: A Detailed Architectural Simulator for Flexible Neural Network Accelerators. *Computing Research Repository*, abs/2006.07137(1):1–8.

[NVIDIA, 2022a] NVIDIA (2022a). NVDLA. Source: http://nvdla.org/, May 2022.

[NVIDIA, 2022b] NVIDIA (2022b). TensorRT. Source: https://developer.nvidia.com/tensorrt, May 2022.

[NXP, 2022] NXP (2022). NXP S32V234 MPU. Source: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/s32v2-vision-mpus-/vision-processor-for-front-and-surround-view-camera-machine-learning-and-sensor-fusion:S32V234, May 2022.

[Parashar et al., 2019] Parashar, A., Raina, P., Shao, Y. S., Chen, Y.-H., Ying, V. A., Mukkara, A., Venkatesan, R., Khailany, B., Keckler, S. W., and Emer, J. (2019). Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315.

[Park and Chung, 2020] Park, S.-S. and Chung, K.-S. (2020). CENNA: Cost-Effective Neural Network Accelerator. *Electronics*, 9(1):1–19.

[PyTorch, 2022] PyTorch (2022). PyTorch. Source: https://pytorch.org/, May 2022.

[Qualcomm, 2019] Qualcomm (2019). Qualcomm Snapdragon. Source: https://developer.qualcomm.com/blog/accelerate-your-device-ai-qualcomm-artificial-intelligence-ai-engine-snapdragon, May 2022.

[Renesas, 2022] Renesas (2022). Renesas e-AI. Source: https://www.renesas.com/jp/en/solutions/key-technology/e-ai.html, May 2022.

[Ryu et al., 2022] Ryu, S., Kim, H., Yi, W., Kim, E., Kim, Y., Kim, T., and Kim, J.-J. (2022). BitBlade: Energy-Efficient Variable Bit-Precision Hardware Accelerator for Quantized Neural Networks. *IEEE Journal of Solid-State Circuits*, 1(1):1–11.

[Samajdar et al., 2020] Samajdar, A., Joseph, J. M., Zhu, Y., Whatmough, P., Mattina, M., and Krishna, T. (2020). A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 58–68.

[Samajdar et al., 2018] Samajdar, A., Zhu, Y., Whatmough, P., Mattina, M., and Krishna, T. (2018). SCALE-sim: Systolic CNN accelerator. *Computing Research Repository*, abs/1811.02883(1):1–11.

[Samsung, 2019] Samsung (2019). Samsung Exynos. Source: https://www.eetimes.com/document.asp?doc_id=1334340, May 2022.

[ServiceNow, 2022] ServiceNow (2022). Enterprise Chatbot – Virtual Agent. Source: https://assistant.google.com, May 2022.

[Shao et al., 2014] Shao, Y. S., Reagen, B., Wei, G.-Y., and Brooks, D. (2014). Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the ACM International Symposium on Computer Architecture (ISCA)*, pages 97–108.

[Shao et al., 2016] Shao, Y. S., Xi, S. L., Srinivasan, V., Wei, G.-Y., and Brooks, D. (2016). Co-designing accelerators and soc interfaces using gem5-aladdin. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12.

[Shivapakash et al., 2020] Shivapakash, S., Jain, H., Hellwich, O., and Gerfers, F. (2020). A Power Efficient Multi-Bit Accelerator for Memory Prohibitive Deep Neural Networks. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *Computing Research Repository*, abs/1409.1556(1):1–14.

[Sohrabizadeh et al., 2021] Sohrabizadeh, A., Bai, Y., Sun, Y., and Cong, J. (2021). Enabling Automated FPGA Accelerator Optimization Using Graph Neural Networks. *Computing Research Repository*, abs/2111.08848(1):1–12.

[Son et al., 2013] Son, Y. H., Seongil, O., Ro, Y., Lee, J. W., and Ahn, J. H. (2013). Reducing memory access latency with asymmetric DRAM bank organizations. In *Proceedings of the ACM International Symposium on Computer Architecture (ISCA)*, pages 380–391.

[Spagnolo et al., 2020] Spagnolo, F., Perri, S., Frustaci, F., and Corsonello, P. (2020). Reconfigurable Convolution Architecture for Heterogeneous Systems-on-Chip. In *Proceedings of the IEEE Mediterranean Conference on Embedded Computing (MECO)*, pages 1–5.

[Strom, 2015] Strom, N. (2015). Scalable distributed DNN training using commodity GPU cloud computing. In *Proceedings of the International Speech Communication Association (ISCA)*, pages 1488–1492.

[Tang and Xie, 2018] Tang, T. and Xie, Y. (2018). Mlpat: A power area timing modeling framework for machine learning accelerators. In *Proceedings of the IEEE International Workshop on Domain Specific System Architecture (DOSSA)*, pages 1–3.

[Tavakoli et al., 2020] Tavakoli, M. R., Sayedi, S. M., and Khaleghi, M. J. (2020). A High Throughput Hardware CNN Accelerator Using a Novel Multi-Layer Convolution Processor. In *Proceedings of the IEEE Iranian Conference on Electrical Engineering (ICEE)*, pages 1–6.

[TensorFlow, 2022] TensorFlow (2022). TensorFlow. Source: https://www.tensorflow.org/, May 2022.

[Tesla, 2019] Tesla (2019). Autopilot and Full Self-Driving Capability. Source: https://analyticsindiamag.com/under-the-hood-of-teslas-ai-chip-that-takes-the-driverless-battle-to-nvidias-home-turf/, May 2022.

[Tesla, 2022] Tesla (2022). Autopilot. Source: https://www.tesla.com, May 2022.

[Texas, 2022] Texas (2022). Texas Instruments Sitara. Source: http://www.ti.com/tool/SITARA-MACHINE-LEARNING, May 2022.

[Toshiba, 2019] Toshiba (2019). Toshiba Visconti 5. Source: https://toshiba.semicon-storage.com/ap-en/company/news/news-topics/2019/01/automotive-20190107-1.html, May 2022.

[Udupa et al., 2020] Udupa, P., Mahale, G., Chandrasekharan, K. K., and Lee, S. (2020). Accelerating Depthwise Convolution and Pooling Operations on z-First Storage CNN Architectures. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5.

[Venkatesan et al., 2019] Venkatesan, R. et al. (2019). MAGNet: A Modular Accelerator Generator for Neural Networks. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 1–8.

[Wu et al., 2019] Wu, Y. N., Emer, J. S., and Sze, V. (2019). Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 1–8.

[Xian et al., 2020] Xian, Z., Li, H., and Li, Y. (2020). Weight Isolation-Based Binarized Neural Networks Accelerator. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4.

[Xiang et al., 2018] Xiang, T., Feng, Y., Ye, X., Tan, X., Li, W., Zhu, Y., Wu, M., Zhang, H., and Fan, D. (2018). Accelerating CNN algorithm with fine-grained dataflow architectures. In *Proceedings of the IEEE International Conference on Smart City (SmartCity)*, pages 243–251.

[Xilinx, 2018] Xilinx (2018). Xilinx xDNN. Source: https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf, May 2022.

[Xilinx, 2021] Xilinx (2021). Vitis AI. Source: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html, May 2022.

[Yang et al., 2020] Yang, X., Gao, M., Liu, Q., Setter, J., Pu, J., Nayak, A., Bell, S., Cao, K., Ha, H., Raina, P., et al. (2020). Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 369–383.

[Ye et al., 2021] Ye, H., Hao, C., Jeong, H., Huang, J., and Chen, D. (2021). ScaleHLS: Achieving Scalable High-Level Synthesis through MLIR. *Computing Research Repository*, abs/2107.11673(1):1–15.

[Zacharopoulos et al., 2022] Zacharopoulos, G., Ejjeh, A., Jing, Y., Yang, E.-Y., Jia, T., Brumar, I., Intan, J., Huzaifa, M., Adve, S., Adve, V., et al. (2022). Trireme: Exploring Hierarchical Multi-Level Parallelism for Domain Specific Hardware Acceleration. *Computing Research Repository*, abs/2201.08603(1):1–20.

[Zhang et al., 2015] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., and Cong, J. (2015). Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium On Field-Programmable Gate Arrays (FPGA)*, pages 161–170.

[Zhang et al., 2021] Zhang, X., Ye, H., and Chen, D. (2021). Being-ahead: Benchmarking and Exploring Accelerators for Hardware-Efficient AI Deployment. *Computing Research Repository*, abs/2104.02251(1):1–12.

[Zhao et al., 2020] Zhao, Y., Li, C., Wang, Y., Xu, P., Zhang, Y., and Lin, Y. (2020). DNN-Chip Predictor: An Analytical Performance Predictor for DNN Accelerators with Various Dataflows and Hardware Architectures. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1593–1597.

APPENDIX A – 2D CONVOLUTION MODEL IN URSA

This Appendix details the conv2d implementation in URSA.

```
#include <sstream>
#include <iomanip>
#include <iostream>

//simulator API
#include <TConv2dArray.h>

TConv2dArray::TConv2dArray(std::string name,
        //signals
        int _a_buffer[][N],
        int _b_buffer[][N]) : TimedModel(name) {

    int x,y;

    for (x = 0; x < N; x++){
                for (y = 0; y < N; y++){
                        _PE[x][y]  = new TPE(this->GetName() + ".PE" + to_string(x) + to_string(y),
                                                                x, y,
                                                                _start,_shift_acc,_shift_out);

                        _z_buffer[x][y] = 0;
                        _a_buffer[x][y] = 0;
                        _b_buffer[x][y] = 0;
                }
        }

        for (x = 0; x < N; x++){
                for (y = 1; y < N; y++){
                        _PE[x][y]->SetTPEAInput(_PE[x][y-1]->GetTPEAOutput());
                }
        }

        for (x = 1; x < N; x++){
                for (y = 0; y < N; y++){
                        _PE[x][y]->SetTPEBInput(_PE[x-1][y]->GetTPEBOutput());
                        _PE[x][y]->SetTPEZInput(_PE[x-1][y]->GetTPEZOutput());
                }
        }
}

TConv2dArray::~TConv2dArray(){
}

void TConv2dArray::Reset(){
        int x,y;

        _systolic_array_state = Conv2dArrayState::INIT_ARRAY;
        _cont_row = 0;
        _cont_column = 0;

        for (x = 0; x < N; x++){
                for (y = 0; y < N; y++){
                        _PE[x][y]->Reset();
                        _z_buffer[x][y] = 0;
                }
```

```
        }

        // Initialize matrixs
        for (x = 0; x < N; x++){
                for (y = 0; y < N; y++){
                        _a_buffer[x][y] = (x*2) + 5 + y;
                        _b_buffer[x][y] = (y+1) + x;
                }
        }
}

void TConv2dArray::InitArray(){
        int x,y;

        switch(_systolic_array_state){
                case Conv2dArrayState::INIT_ARRAY:{
                        _start = 0;

                        for (x = 0; x < N; x++) {
                                _PE[x][0]->SetAInputValue(_a_buffer[x][_cont_column]);
                                _PE[x][0]->ShiftTPEAInput();
                                _PE[0][x]->SetBInputValue(_b_buffer[_cont_row][x]);
                                _PE[0][x]->ShiftTPEBInput();
                        }
                        _cont_column = _cont_column + 1;
                        _cont_row = _cont_row + 1;

                        for (x = 0; x < N; x++){
                                for (y = 1; y < N; y++){
                                        _PE[x][y]->ShiftTPEAInput();
                                }
                        }

                        for (x = 1; x < N; x++){
                                for (y = 0; y < N; y++){
                                        _PE[x][y]->ShiftTPEBInput();
                                }
                        }

                        if (_cont_column <= N)
                                _systolic_array_state = Conv2dArrayState::START_MULT;
                        else {
                                _systolic_array_state = Conv2dArrayState::SHIFT_OUT;
                                _start = 0;
                                _cont_column = 0;
                                _cont_row = 0;
                                _shift_acc = 1;
                                _shift_out = 0;
                        }
                }break;
                default: break;
        }
}

void TConv2dArray::StartMult(){
        switch(_systolic_array_state){
                case Conv2dArrayState::START_MULT:{
                        _start = 1;
                        _systolic_array_state = Conv2dArrayState::INIT_ARRAY;
                } default: break;
```

```
        }
}

void TConv2dArray::ShiftOut(){
        int y;

        switch(_systolic_array_state){
                case Conv2dArrayState::SHIFT_OUT:{
                        _start = 0;
                        _shift_out = 1;
                        _shift_acc = 0;

                        if (_cont_row < N){
                                _shift_out = 1;

                                for (y = 0; y < N; y++) {
                                        _z_buffer[N-(_cont_row+1)][y] = _PE[N-1][y]->GetZOutputValue();
                                }
                        }

                        _cont_row++;

                        if (_cont_row == N)
                                _systolic_array_state = Conv2dArrayState::END_OP;
                }break;
                default: break;
        }
}

void TConv2dArray::EndOp(){
        int x,y;

        switch(_systolic_array_state){
                        case Conv2dArrayState::END_OP:{
                        printf("systolic_array: accumulator result!\n");
                        for (x = 0; x < N; x++){
                                for (y = 0; y < N; y++){
                                        printf("%d ", _PE[x][y]->GetMACResult());
                                }
                                printf("\n");
                        }

                        printf("systolic_array: shifted out result!\n");
                        for (x = 0; x < N; x++){
                                for (y = 0; y < N; y++){
                                        printf("%d ", _z_buffer[x][y]);
                                }
                                printf("\n");
                        }

                        printf("Done!\n");
                        while(1);
                }break;
                default: break;
        }
}

std::string TConv2dArray::GetName() {
        return ".systolic_array";
}
```

```
SimulationTime TConv2dArray::Run() {
        int x,y;

        this->EndOp();
        this->ShiftOut();
        this->InitArray();
        this->StartMult();

    for (x = N-1; x >= 0; x--){
                for (y = N-1; y >= 0; y--){
                        _PE[x][y]->Run();
                }
        }

    return 1;
}
```

APPENDIX B – DSE TABLES

This Appendix presents the complete set of results related to the DSE exploration, for the physical synthesis and analytical flows. These Tables show the PPA information and its breakdown regarding accelerator core and output buffer, the necessary buffer capacity to store all input values, the number of memory reads, and the number of memory writes. The column **synt.flow** corresponds to the physical synthesis flow (from Section 6.1), while the **analytic** column is the results from the analytical flow (from Section 6.3). The column |**error**| is the obtained error for each obtained value. The Tables are separated by memory type (Tables B.1 to B.3 regard SRAM while Tables B.4 to B.6 regard DRAM), and each Table presents the values for the five Accelerators described in Chapter 4.

Table B.1: Cifar10 CNN layer 0 analytic results for SRAM memory type.

layer 0 (sram)															
dataflow	**ws**			**ws buf**			**is slice**			**is slice buf**			**os**		
results	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**
core area (μm^2)	14,563.9680	14,563.9680	0.00000	13,399.0464	13,399.0464	0.0000	15,425.3376	15,425.3376	0.0000	13,886.3616	13,886.3616	0.0000	14,938.3488	14,938.3488	0.0000
output buffer area (μm^2)	0	0	0.00000	37,860.7680	38,266.0000	1.0703	0.0000	0.0000	0.0000	40,674.6624	40,859.0000	0.4532	0.0000	0	0.0000
acc total area (μm^2)	14,563.9680	14,563.9680	0.00000	51,259.8144	51,665.0464	0.7905	15,425.3376	15,425.3376	0.0000	54,561.0240	54,745.3616	0.3379	14,938.3488	14,938.3488	0.0000
input capacity (Bits)	0	0	0.00000	0	0	0.0000	7168	7,168	0.0000	7168	7,168	0.0000	0	0	0.0000
number of cycles	236,888	236,592	0.12495	236,887	236,592	0.1245	142,875.0000	140,529.0000	1.6420	136,125.0000	126,129.0000	7.3433	605,942.0000	591,264	2.4223
core power (mW)	0.9498	0.9498	0.00000	0.9870	0.9870	0.0000	2.0400	2.0400	0.0000	2.0969	2.0970	0.0048	1.0380	1.038	0.0000
buf power (mW)	0	0	0.00000	1.33	1.328832	0.0878	0.0000	0	0.0000	1.945	1.9586688	0.7028	0.0000	0	0.0000
acc power (mW)	0.9498	0.9498	0.00000	2.317	2.315832	0.0504	2.0400	2.04	0.0000	4.0420	4.0556688	0.3382	1.0380	1.038	0.0000
core energy (fJ)	224,996.2224	224,715.0816	0.12495	548,867.1790	547,907.3245	0.1749	291,465.0000	261,827.8800	10.1683	550,217.2500	511,537.4501	7.0299	628,967.7960	613,732.032	2.4223
input memory reads	71,056	71,056	0.00000	71,056	71,056	0.0000	7,198.0000	6,523.0000	9.3776	7,198.0000	6,523.0000	9.3776	194,735.0000	194,704	0.0159
input memory writes	0	0	0.00000	0	0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory reads	7,200	7,200	0.00000	0	0	0.0000	7,200.0000	7,200.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory writes	10,800	10,800	0.00000	3,600	3,600	0.0000	10,800.0000	10,800.0000	0.0000	3,600.0000	3,600.0000	0.0000	3,600.0000	3,600	0.0000
sram read energy (nJ)	1,061.1905	1,061.1905	0.00000	963.5549	963.5549	0.0000	195.2441	186.0907	4.6882	97.6085	88.4551	9.3776	2,640.7040	2,640.283592	0.0159
sram write energy (nJ)	145.9847	145.9847	0.00000	48.6616	48.6616	0.0000	145.9847	145.9847	0.0000	48.6616	48.6616	0.0000	48.6616	48.66156	0.0000
total energy (nJ)	1,207.4002	1,207.3999	0.00002	1,012.7653	1,012.7644	0.0001	341.5202	332.3372	2.6889	146.8203	137.6282	6.2607	2,689.9945	2,689.558884	0.0162

Table B.2: Cifar10 CNN layer 1 analytic results for SRAM memory type.

layer 1 (sram)															
dataflow	**ws**			**ws buf**			**is slice**			**is slice buf**			**os**		
results	**synt. flow**	**analytic**	**||error|| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**
core area (μm^2)	15,037.5744	14,563.9680	3.1495	13,777.0176	13,399.0464	2.7435	15,702.4512	15,425.3376	1.7648	14,156.9472	13,886.3616	1.9113	15,336.2304	14,938.3488	2.5944
output buffer area (μm^2)	0	0	0.0000	8,627.5680	8,646.6000	0.2206	0.0000	0.0000	0.0000	38,007.8112	38,171.0000	0.4294	0.0000	0	0.0000
acc. total area (μm^2)	15,037.5744	14,563.9680	3.1495	22,404.5856	22,045.6464	1.6021	15,702.4512	15,425.3376	1.7648	52,164.7584	52,057.3616	0.2059	15,336.2304	14,938.3488	2.5944
input capacity (Bits)	0	0	0.0000	0	0	0.0000	74240	74,240	0.0000	74240	74,240	0.0000	0	74,240	0.0000
number of cycles	642,152	647,680	0.8609	642,151	647,680	0.8610	291,364.0000	313,072.0000	7.4505	278,134.0000	266,032.0000	4.3511	1,407,206.0000	1,359,616	3.3819
core power (mW)	0.8683	0.9498	9.3862	0.9203	0.9870	7.2476	1.9200	2.0400	6.2500	1.906	2.0970	10.0210	1.0230	1.038	1.4663
buf power (mW)	0	0	0.0000	0.3257	0.3255114752	0.0579	0.0000	0	0.0000	1.824	1.836927488	0.7087	0.0000	0	0.0000
acc power (mW)	0.8683	0.9498	9.3862	1.246	1.312511475	5.3380	1.9200	2.04	6.2500	3.7300	3.933927488	5.4672	1.0230	1.038	1.4663
core energy (fJ)	557,580.5816	615,166.4640	10.3278	800,120.1460	850,087.4323	6.2450	559,418.8800	609,747.8400	8.9967	1,037,439.8200	1,046,550.5975	0.8782	1,439,571.7380	1,411,281.408	1.9652
input memory reads	192,544	192,544	0.0000	192,544	192,544	0.0000	12,480.0000	11,696.0000	6.2821	12,480.0000	11,696.0000	6.2821	452,255.0000	452,192	0.0139
input memory writes	0	0	0.0000	0	0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory reads	23,520	23,520	0.0000	0	0	0.0000	23,520.0000	23,520.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory writes	25,088	25,088	0.0000	1,568	1,568	0.0000	25,088.0000	25,088.0000	0.0000	1,568.0000	1,568.0000	0.0000	1,568.0000	1,568	0.0000
sram read energy (nJ)	2,929.9359	2,929.9359	0.0000	2,610.9929	2,610.9929	0.0000	488.1780	477.5466	2.1778	169.2350	158.6036	6.2821	6,132.8039	6,131.949616	0.0139
sram write energy (nJ)	339.1170	339.1170	0.0000	21.1948	21.1948	0.0000	339.1170	339.1170	0.0000	21.1948	21.1948	0.0000	21.1948	21.1948128	0.0000
total energy (nJ)	3,269.6105	3,269.6680	0.0018	2,632.9878	2,633.0378	0.0019	827.8544	817.2733	1.2781	191.4673	180.8450	5.5479	6,155.4383	6,154.55571	0.0143

Table B.3: Cifar10 CNN layer 2 analytic results for SRAM memory type.

layer 2 (sram)															
dataflow	**ws**			**ws buf**			**is slice**			**is slice buf**			**os**		
results	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**
core area (μm^2)	15,002.4864	14,563.9680	2.9230	13,762.6560	13,399.0464	2.6420	15,696.2496	15,425.3376	1.7260	14,162.1696	13,886.3616	1.9475	15,753.2064	14,938.3488	5.1726
output buffer area (μm^2)	0	0	0.0000	1,990.5504	1,990.6000	0.0025	0.0000	0.0000	0.0000	32,648.1600	32,795.0000	0.4498	0.0000	0	0.0000
acc total area (μm^2)	15,002.4864	14,563.9680	2.9230	15,753.2064	15,389.6464	2.3078	15,696.2496	15,425.3376	1.7260	46,810.3296	46,681.3616	0.2755	15,753.2064	14,938.3488	5.1726
input capacity (Bits)	0	0	0.0000	0	0	0.0000	295936	295,936	0.0000	295936	295,936	0.0000	0	0	0.0000
number of cycles	766,152	804,864	5.0528	766,151	804,864	5.0529	244,980.0000	267,744.0000	9.2922	235,494.0000	232,032.0000	1.4701	1,036,742.0000	999,936	3.5502
core power (mW)	0.8812	0.9498	7.7848	0.9076	0.9870	8.7483	1.7290	2.0400	17.9873	1.8	2.0970	16.5000	1.0310	1.038	0.6790
buf power (mW)	0	0	0.0000	0.1234	0.1233626112	0.0303	0.0000	0.0000	0.0000	1.424	1.435372032	0.7986	0.0000	0	0.0000
acc power (mW)	0.8812	0.9498	7.7848	1.031	1.110362611	7.6976	1.7290	2.0400	17.9873	3.2240	3.532372032	9.5649	1.0310	1.038	0.6790
core energy (fJ)	675,133.1424	764,459.8272	13.2310	789,901.6810	893,690.8927	13.1395	42,3570.4200	535,361.2800	26.3925	759,232.6560	819,623.3473	7.9542	1,068,881.0020	1,037,933.568	2.8953
input memory reads	229,440	229,440	0.0000	229,440	229,440	0.0000	21,376.0000	21,088.0000	1.3473	21,376.0000	21,088.0000	1.3473	333,119.0000	332,992	0.0381
input memory writes	0	0	0.0000	0	0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory reads	17,856	17,856	0.0000	0	0	0.0000	17,856.0000	17,856.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory writes	18,432	18,432	0.0000	576	576	0.0000	18,432.0000	18,432.0000	0.0000	576.0000	576.0000	0.0000	576.0000	576	0.0000
sram read energy (nJ)	3,353.4574	3,353.4574	0.0000	3,111.3211	3,111.3211	0.0000	532.0055	528.1001	0.7341	289.8692	285.9638	1.3473	4,517.2602	4,515.538016	0.0381
sram write energy (nJ)	249.1472	249.1472	0.0000	7.7858	7.7858	0.0000	249.1472	249.1472	0.0000	7.7858	7.7858	0.0000	7.7858	7.7858496	0.0000
total energy (nJ)	3,603.2797	3,603.3691	0.0025	3,119.8969	3,120.0007	0.0033	781.5763	777.7827	0.4854	298.4143	294.5693	1.2885	4,526.1149	4524.361799	0.0387

Table B.4: Cifar10 CNN layer 0 analytic results for DRAM memory type.

layer 0 (dram)															
dataflow	**ws**			**ws buf**			**is slice**			**is slice buf**			**os**		
results	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**
core area (μm^2)	14,563.9680	14,563.9680	0.0000	13,399.0464	13,399.0464	0.0000	15,425.3376	15,425.3376	0.0000	13,886.3616	13,886.3616	0.0000	14,938.3488	14,938.3488	0.0000
output buffer area (μm^2)	0	0	0.0000	37,860.7680	37,933.0000	0.1908	0.0000	0.0000	0.0000	40,674.6624	40,859.0000	0.4532	0.0000	0	0.0000
acc total area (μm^2)	14,563.9680	14,563.9680	0.0000	51,259.8144	51,332.0464	0.1409	15,425.3376	15,425.3376	0.0000	54,561.0240	54,745.3616	0.3379	14,938.3488	14,938.3488	0.0000
input capacity (Bits)	0	0	0.00000	0	0	0.0000	7168	7168	0.0000	7168	7,168	0.0000	0	0	0.0000
number of cycles	449,915	451,584	0.3710	449,911	451,584	0.3719	182,694.0000	170,898.0000	6.4567	155,694.0000	14,5698.0000	6.4203	1,190,102.0000	1,175,328	1.2414
core power (mW)	0.7443	0.7443	0.0000	0.8040	0.8040	0.0000	1.7940	1.7940	0.0000	2.006	2.0060	0.0000	0.8766	0.8766	0.0000
buf power (mW)	0	0	0.0000	1.104	1.328832	20.3652	0.0000	0	0.0000	1.835	1.9586688	6.7394	0.0000	0	0.0000
acc power (mW)	0.7443	0.7443	0.0000	1.908	1.906664	0.0700	1.7940	1.794	0.0000	3.8410	3.8379679	0.0800	0.8766	0.8766	0.0000
core energy (fJ)	334,871.7345	336,113.9712	0.3710	858,430.1880	963,152.8059	12.1993	327,753.0360	306,591.0120	6.4567	598,020.6540	577,644.3148	3.4073	1,043,243.4132	1,030,292.525	1.2414
input memory reads	71,008	71,056	0.0676	71,008	71,056	0.0676	6,523.0000	6,523.0000	0.0000	6,523.0000	6,523.0000	0.0000	194,720.0000	194,704	0.0082
input memory writes	0	0	0.0000	0	0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory reads	7,200	7,200	0.0000	0	0	0.0000	7,200.0000	7,200.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory writes	10,800	10,800	0.0000	3,600	3,600	0.0000	10,800.0000	10,800.0000	0.0000	3,600.0000	3,600.0000	0.0000	3,600.0000	3,600	0.0000
sram read energy (nJ)	12,772.8524	12,780.6917	0.0614	11,596.9556	11,604.7949	0.0676	2,241.2266	2,241.2266	0.0000	1,065.3298	1,065.3298	0.0000	31,801.4757	31,798.86258	0.0082
sram write energy (nJ)	1,795.7376	1,795.7376	0.0000	598.5792	598.5792	0.0000	1,795.7376	1,795.7376	0.0000	598.5792	598.5792	0.0000	598.5792	598.5792	0.0000
total energy (nJ)	1,4568.9248	1,4576.7654	0.0538	12,196.3932	12,204.3372	0.0651	4,037.2920	4,037.2708	0.0005	1,664.5071	1,664.4867	0.0012	32,401.0981	32,398.47207	0.0081

Table B.5: Cifar10 CNN layer 1 analytic results for DRAM memory type.

layer 1 (dram)															
dataflow	**ws**			**ws buf**			**is slice**			**is slice buf**			**os**		
results	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**
core area (μm^2)	15,037.5744	14,563.9680	3.1495	13,777.0176	13,399.0464	2.7435	15,702.4512	15,425.3376	1.7648	14,156.9472	13,886.3616	1.9113	15,336.2304	14,938.3488	2.5944
output buffer area (μm^2)	0	0	0.0000	8,627.5680	8,646.6000	0.2206	0.0000	0.0000	0.0000	38,007.8112	38,171.0000	0.4294	0.0000	0	0.0000
acc total area (μm^2)	15,037.5744	14,563.9680	3.1495	22,404.5856	22,045.6464	1.6021	15,702.4512	15,425.3376	1.7648	52,164.7584	52,057.3616	0.2059	15,336.2304	14,938.3488	2.5944
input capacity (Bits)	0	0	0.0000	0	0	0.0000	74240	74,240	0.0000	74240	74,240	0.0000	0	74,240	0.0000
number of cycles	1,218,251	1,245,184	2.2108	1,218,247	1,245,184	2.2111	363,202.0000	373,248.0000	2.7660	313,222.0000	301,120.0000	3.8637	2,763,878.0000	2,716,096	1.7288
core power (mW)	0.7135	0.7443	4.3167	0.7845	0.8040	2.4857	1.7020	1.7940	5.4054	1.809	2.0060	10.8900	0.8690	0.8766	0.8746
buf power (mW)	0	0	0.0000	0.2785	0.3255114752	16.8802	0.0000	0	0.0000	1.727	1.836927488	6.3652	0.0000	0	0.0000
acc power (mW)	0.7135	0.7443	4.3167	1.063	1.0821797504	1.8000	1.7020	1.794	5.4054	3.5360	3.72975167	5.4800	0.8690	0.8766	0.8746
core energy (fJ)	869,222.0885	926,790.4512	6.6230	1,294,996.5610	1,406,449.6167	8.6064	618,169.8040	669,606.9120	8.3209	1,107,552.9920	1,157,182.3252	4.4810	2,401,809.9820	2,380,929.754	0.8694
input memory reads	192,032	192,544	0.2666	192,032	192,544	0.2666	11,696.0000	11,696.0000	0.0000	11,696.0000	11,696.0000	0.0000	452,224.0000	452,192	0.0071
input memory writes	0	0	0.0000	0	0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory reads	23,520	23,520	0.0000	0	0	0.0000	23,520.0000	23,520.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory writes	25,088	25,088	0.0000	1,568	1,568	0.0000	25,088.0000	25,088.0000	0.0000	1,568.0000	1,568.0000	0.0000	1,568.0000	1,568	0.0000
sram read energy (nJ)	35,203.7371	35,287.3564	0.2375	31,362.4742	31,446.0935	0.2666	5,751.4419	5,751.4419	0.0000	1,910.1790	1,910.1790	0.0000	73,856.7715	73,851.54525	0.0071
sram write energy (nJ)	4,171.4319	4,171.4319	0.0000	260.7145	260.7145	0.0000	4,171.4319	4,171.4319	0.0000	260.7145	260.7145	0.0000	260.7145	260.714496	0.0000
total energy (nJ)	39,376.0382	39,459.7151	0.2125	31,624.4837	31,708.2145	0.2648	9,923.4920	9,923.5434	0.0005	2,172.0011	2,172.0507	0.0023	74,119.8878	74,114.64067	0.0071

Table B.6: Cifar10 CNN layer 2 analytic results for DRAM memory type.

layer 2 (dram)															
dataflow	**ws**			**ws buf**			**is slice**			**is slice buf**			**os**		
results	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**	**synt. flow**	**analytic**	**|error| (%)**
core area (μm^2)	15,002.4864	14,563.9680	2.9230	13,762.6560	13399.0464	2.6420	15,696.2496	15,425.3376	1.7260	14,162.1696	13,886.3616	1.9475	15,320.4000	14,938.3488	2.4937
output buffer area (μm^2)	0	0	0.0000	1,990.5504	1,990.6000	0.0025	0.0000	0.0000	0.0000	32,648.1600	32,795.0000	0.4498	0.0000	0	0.0000
acc total area (μm^2)	15,002.4864	14,563.9680	2.9230	15,753.2064	15,389.6464	2.3078	15,696.2496	15,425.3376	1.7260	46,810.3296	46,681.3616	0.2755	15,320.4000	14,938.3488	2.4937
input capacity (Bits)	0	0	0.0000	0	0	0.0000	295,936	295,936	0.0000	295936	295,936	0.0000	0	0	0.0000
number of cycles	1,448,331	1,572,864	8.5984	1,448,327	1,572,864	8.5987	335,586.0000	349,440.0000	4.1283	298,758.0000	295,296.0000	1.1588	2,035,910.0000	1,998,720	1.8267
core power (mW)	0.7141	0.7443	4.2291	0.7534	0.8040	6.7162	1.4520	1.7940	23.5537	1.588	2.0060	26.3224	0.8807	0.8766	0.4655
buf power (mW)	0	0	0.0000	0.115	0.1233626112	7.2718	0.0000	0	0.0000	1.277	1.435372032	12.4019	0.0000	0	0.0000
acc power (mW)	0.7141	0.7443	4.2291	0.8684	0.9189060223	5.8200	1.4520	1.794	23.5537	2.8650	3.279355519	14.4600	0.8807	0.8766	0.4655
core energy (fJ)	1,034,253.1671	1,170,682.6752	13.1911	1,257,727.1668	1,458,615.2661	15.9723	487,270.8720	626,895.3600	28.6544	855,941.6700	1,016,223.3956	18.7258	1,793,025.9370	1,752,077.952	2.2837
input memory reads	227,392	229,440	0.9006	227,392	229,440	0.9006	21,088.0000	21,088.0000	0.0000	21,088.0000	21,088.0000	0.0000	333,056.0000	332,992	0.0192
input memory writes	0	0	0.0000	0	0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory reads	17,856	17,856	0.0000	0	0	0.0000	17,856.0000	17,856.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0	0.0000
ofmap memory writes	18,432	18,432	0.0000	576	576	0.0000	18,432.0000	18,432.0000	0.0000	576.0000	576.0000	0.0000	576.0000	576	0.0000
sram read energy (nJ)	40,053.6581	40,388.1354	0.8351	37,137.4340	37,471.9114	0.9006	6,360.2951	6,360.2951	0.0000	3,444.0711	3,444.0711	0.0000	54,394.3729	54,383.92045	0.0192
sram write energy (nJ)	3,064.7255	3,064.7255	0.0000	95.7727	95.7727	0.0000	3,064.7255	3,064.7255	0.0000	95.7727	95.7727	0.0000	95.7727	95.772672	0.0000
total energy (nJ)	43,119.4179	43,454.0316	0.7760	37,234.4644	37,569.1426	0.8988	9,425.5079	9,425.6475	0.0015	3,540.6997	3,540.8600	0.0045	54,491.9386	54,481.4452	0.0193

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br