CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

An HDL-parameterizable *m* x *n* systolic-array-based matrix multiplier for DNN applications

Master's Thesis in Embedded Electronic System Design

Ibrahim Tayyem

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

An HDL-parameterizable m x n systolic-array-based matrix multiplier for DNN applications

Ibrahim Tayyem

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

An HDL-parameterizable m x n systolic-array-based matrix multiplier for DNN applications
Ibrahim Tayyem

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Co-Supervisor: Mateo Vázquez Maceiras, Department of Computer Science and Engineering
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

An HDL-parameterizable m `x` n systolic-array-based matrix multiplier for DNN applications

Ibrahim Tayyem
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Deep Neural Networks (DNNs) and its applications have been employed in different platforms with different requirements and resource constrains. General Matrix Multiplication (GEMM) is a common way to compute convolution which represents the most computationally demanding operation in DNNs. The variation in convolutional layers' shapes and sizes results in different GEMM parameters. In this work, two versions of a matrix multiplier are presented. Both accelerators are designed using VHDL-93, are parameterizable at the Hardware Description Language (HDL) level and capable of multiplying m `x` n matrices using non-squared Systolic Arrays (SAs). v1 is capable of multiplying matrices of the size and shape of the SA and leave multiplying larger matrices as software overhead by re-feeding tiles of the same size to the accelerator. v2 on the other hand is capable of multiplying matrices that are larger than the used SA improving performance by on-chip feedback. On the other hand, resource utilization, specifically BRAM utilization in v1 is less than that in v2 giving the same SA shape since there is no need to save all tiles' parameters before the start of the execution. Thus available resources (higher DSP/MAC resources in v1 and higher BRAM in v2) make a specific version preferable to another to achieve the same acceleration.

Keywords: Machine Learning, Convolutional Neural Network, reconfigurable, Domain Specific Architecture, Systolic Array, Matrix Multiplication.

Acknowledgements

I would like to thank my supervisor Prof. Pedro Petersen Moura Trancoso for providing me this thesis opportunity and academic feedback throughout the thesis process, and co-supervisor Mateo Vázquez Maceiras for helpful discussions and providing the development board. I would also like to thank Prof. Per Larsson-Edefors for his examination of this thesis.

Ibrahim Tayyem, Gothenburg, August 2023

Contents

Acronyms

bfloat16 Brain Floating Point 3, 8

CNN Convolutional Neural Network 1, 2, 5, 7, 30, 34, 35, 41, 44, 45
CONV Convolutional 5, 6, 11, V, VI, VII

DNN Deep Neural Network 1, 2, 3, 5, 7, 8

FC Fully-Connected 5, 7, 11
FPGA Field Programmable Gate Array 14, 29, 45

GEMM General Matrix Multiplication 1, 2, 7
GPU Graphical Processing Unit 1

HDL Hardware Description Language v, 21, 45

ifmap Input Feature Map 5, 7, 8, 10, 36
ILA Integrated Logic Analyzer 14
im2col Image to Column 7, 30, 45, V, VI, VII
IS Input Stationary 1, 3, 10, 30

kn2row Kernel to Row 7

MAC Multiply Accumulate 3, 20, 21
MCMK Multiple Channel Multiple Kernel 7
MCSK Multiple Channel Single Kernel 7
ML Machine Learning 1, 9

NoC Network-on-Chip 3, 8, 11

ofmap Output Feature Map 5, 7
OS Output Stationary 1, 3, 10, 11

PE Processing Element 1, 3, 8, 9, 10, 11, 20, 21, 27, 36
POOL Pooling 5
psum Partial Sum 8, 10

ReLU Rectified Linear Unit 7
RS Row Stationary 1, 3, 11

SA Systolic Array v, 1, 2, 3, 9, 15, 18, 20, 22, 23, 24, 25, 26, 27, 30, 31, 33, 34, 36, 37, 38, 39, 42, 43, 44, 45, II

TPU Tensor Processing Unit 2, 3
TS Tunnel Stationary 3, 11

WS Weight Stationary 1, 2, 3, 9, 10, 11, 30

1

Introduction

With the end of Moore's Law, the search for new ways to improve performance in computing systems is increasing. For data-intensive applications such as Machine Learning (ML), the main approach is to use hardware accelerators or Graphical Processing Units (GPUs) to offload computations of resource-demanding tasks. A common branch in the field of ML are neural networks. Convolutional Neural Networks (CNNs) are a special kind of deep feedforward networks in which the output of one layer is fed to the next layer in an acyclic fashion and with no feedback [1]. Among numerical computations involved in the inference phase of a trained CNN lie convolutions as the most demanding operations [2]–[4]. The convolution operation can be performed using matrix multiplication [1], [3], [5]. One architectural methodology for mapping computations in hardware accelerators is Systolic Array (SA), used for accelerating General Matrix Multiplications (GEMMs), and kernel key in domains such as ML. SAs are simple and flexible architectures that can leverage the regular pattern of GEMM. However, while SAs define how computations are performed, there are still other aspects to analyze. These aspects include how to feed SAs the required data and how to control them. In all, there are multiple considerations when it comes to design hardware accelerators based on spatial architectures such as SAs.

1.1 Problem Statement

ML algorithms rely on updating numerical solutions via an iterative process rather than providing a full analytical formula [1]. However, exact formulas for computations done at the hardware level still need to be derived and mapped to the existing compute resources.

Most industrial hardware accelerators for Deep Neural Networks (DNNs) have architectures that perform parallel execution either in a systolic fashion or parallel vector units [6]. Although different low-level implementations exists between different accelerators, their data-mapping strategies can classify accelerators in categories depending upon dataflow. Different categories of DNN processing dataflows can be identified. Weight Stationary (WS), Input Stationary (IS), Output Stationary (OS) and Row Stationary (RS) dataflow are such categories [7], [8]. This project explores these dataflows and studies how data between Processing Elements (PEs) in a spatial architecture can be reused to minimize data movement and the associated performance and energy efficiency.

The main idea of reconfigurable spatial architectures is to keep the design flexible so that it can be scaled up whenever access to more resources is obtained without the need to be significantly redesigned. A configurable design at the RTL description using well-known synthesizable HDLs will not be constrained to a specific platform and may aid reconfiguration at runtime. Also, the ability to construct non-squared arrays will better utilize the available resources. Further, given a convolutional layer, the computation using GEMM involves constructing another matrices which contain redundant data. The dimensions of these matrices are related to the filter dimensions which in turn affects the number of parallel operations given a SA dimension. With this in mind and using a specific dataflow, this project aims to design the accelerator using a SA and explore the effect of different shapes and sizes on the performance.

1.2 Objective

The purpose of the project is to accelerate computations done in the inference phase of CNN-based ML frameworks. The project aims to develop hardware accelerator for ML and implement it into an FPGA. The accelerator is to be able to perform convolution through matrix-matrix multiplication through 2D systolic arrays. Further, the project will study the scalability of the design and the effect of different array sizes on the overall performance.

1.3 Related Work

Systolic arrays, as a methodology for mapping computations into hardware structures, were introduced by Kung in [9]. The paradigm has been proposed to solve general-purpose programmable systems as well as a variety of special-purpose resource-demanding applications including Deep Neural Networks. The Wrap computer [10], the Brown Systolic Array (B-SYS) [11] and the Geometric Arithmetic Parallel Processor (GAPP) [12] are general-purpose implementations of SAs. For special-purpose systems, the Princeton Nucleic Acid Comparator (P-NAC) [13], Systolic-CNN [14], SA-based MAC unit in [15] and Google's Tensor Processing Unit (TPU) [16] are such implementations. However, different implementations, mapping strategies and topologies of systolic arrays as well as different programming interfaces to special-purpose accelerators impose a huge challenge as full-system performance relies on both, the hardware and software stack. Gemmini [6] is an open-source full-stack Deep Neural Network (DNN) accelerator generator that supports both the hardware and software stack and enables full-system performance evaluation by tuning different hardware parameters. For the design of a SA accelerator, Systolic CNN Accelerator Simulator (SCALE-Sim) [17] is a tool that enables exploring different accelerator configurations and neural network architectures and provides performance and efficiency measures to aid the design of CNN accelerators.

Several spatial architectures that target DNNs have been implemented. The designs in TPU [16], NeuFlow [18] and nn-X [19] are all based on SAs that utilize WS dataflow. Chakradhar et al. [2] have implemented a configurable accelerator that adapts to different workloads by dynamically configuring its hardware and software

components. The design uses a systolic architecture based on OS dataflow. The designs in [20], [21] were also based on OS dataflow. The Reduced-precision matrix Multiplication Engine (RedMulE) [22] is an implementation that performs 16-bit floating-point matrix multiplication for targeting on-chip training and uses a semi-SA-based architecture. This design is symmetric regarding its inputs and can be used as WS or IS. Gupta et al. [23] have also implemented a systolic array accelerator for training purposes but using 16-bit wide fixed-point representation incorporating the importance of approximate computing in the field. The later design uses OS dataflow. The bulk of the compute power in TPU v4 [24] also consists of 2D MAC units based on a systolic array. Theses arrays perform operations based on Brain Floating Point (bfloat16) number representation. Shi et al. [25] used Winograd algorithm [26] to compute partitions using clusters of small SAs of size `4x4` and implemented the design using OS dataflow. Eyeriss [27] uses RS dataflow by implementing a hierarchical mesh Network-on-Chip (NoC) that allows for data reuse among different datatypes, i.e. weights, inputs and outputs. However, it requires a substantial effort for preparing the data and mapping the computations [28]. iFPNA [29] has a programmable engine that allows for choosing among different dataflows including IS, RS and Tunnel Stationary (TS) and supports different bit-widths. The proposed HeSA [4] is a design that targets compact CNN models with support for multiple dataflows and improve PE utilization in depthwise convolution layers compared to standard systolic arrays. SA-generators like [6] generates designs in the Chisel HDL targeting a specific platform for evaluation purposes and the array size is configured beforehand in the generator.

The SAs proposed in this thesis can be configured in the HDL code with the ability to construct non-squared arrays. Further, the design is independent of the scalar multiplication compute unit as DSPs can be replaced by generic MAC resources without affecting design abstraction beyond this level.

1.4 Thesis Outline

The contents of the thesis is organized as follows: Chapter 2 includes theoretical background required to follow the work. It starts with a brief summary of DNNs and the most common layer types. It then discusses how a convolutional layer can be calculated using matrix multiplication. The rest of the chapter provides introduction to DNN accelerators using spatial architectures, namely computational nodes called PEs and some common on-Chip Networks (NoCs) that serve such designs. The chapter ends with a brief discussion of dataflows used in the design of DNN accelerators. Chapter 3 describes the methodology followed in the thesis process as well as the system block design where the resulted custom IP were connected and tested. Chapter 4 presents the design of two versions of the accelerator. Chapter 5 provides results of the design. Chapter 6 concludes the work and suggest some possible future work.

2

Theory

This chapter starts with an overview of Deep Neural Networks (DNNs) and the different layers commonly seen in today's models. It also describes how the convolution operation can be transformed into matrix multiplication that can be implemented in different spatial architectures. Then, the background required for implementing some of DNN processing dataflows will be given through an overview of common on-Chip Networks (NoCs) in the field. Finally, different types of dataflows commonly used in the design of DNN's hardware accelerators will be explored.

2.1 Deep Neural Networks

The ability of machine learning by applying statistical methodologies on data and the need for extracting more interesting patterns from data and constructing higher abstraction has inspired the development of DNNs to include more layers between input and output layers [1], [30]. Convolutional Neural Networks (CNNs) are a common form of DNNs that targets 2D features and usually contains three types of layers: Convolutional (CONV) layer, Pooling (POOL) layer and Fully-Connected (FC) layer [31].

The Output Feature Map (ofmap) of one CONV layer is done by convolving Input Feature Map (ifmap) with filter weights (see section 2.2). The pooling layer is used to downsample feature maps, typically using `max` or `average` functions [31] and is essential to handle inputs of varying size, eg. to restrict the size of the input to upcoming layer [1]. FC layers follows CONV and POOL layers to classify the previously extracted features [31]. A non-linearity (activation) function is usually applied after CONV and POOL layers [1]. The next section gives more computational details.

2.2 CONV-GEMM Transformation

Many machine learning libraries implement a related function to the convolution operation, cross-correlation, but still call it convolution [1]. This function is the same as convolution but without flipping the kernel. Equation 2.1 shows a 2D version of the operation.

$$\begin{aligned} y(i,j) &= (x * w)(i,j) \\ &= \sum_m \sum_n x(i+m, j+n) w(m,n) \end{aligned} \tag{2.1}$$

The transformation of the convolution operation into matrix multiplication is done by constructing another matrix of the input. This matrix is sometimes called *Toeplitz Matrix* in literature [1], [5], [8], see figure 2.1.

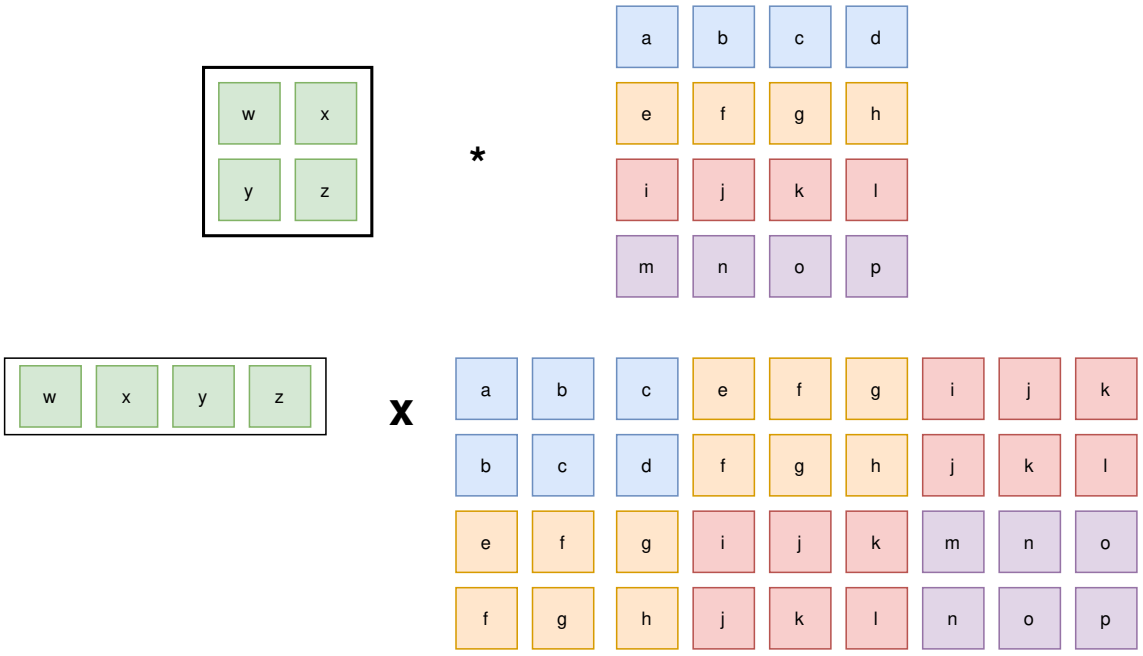


Figure 2.1: Transforming 2D convolution into matrix multiplication using *Toeplitz Matrix*.

In this thesis, the parameters shown in table 2.1 describing a CONV layer will be used.

Table 2.1: Parameters of a convolutional layer

Parameter	**Tensor**	**Description**
N	Input	Batch size
H_i	Input	Height
W_i	Input	Width
C_i	Input	# Channels
S	Filter	Vertical stride = Horizontal stride
H_f	Filter	Height
W_f	Filter	Width
C_i	Filter	# Channels
H_o	Output	Height = $(H_i - H_f + S)/S$
W_o	Output	Width = $(W_i - W_f + S)/S$
C_o	Output	# Channels = # 3D filters

Given the parameters in table 2.1, a CONV layer can be calculated using eq. 2.2 [3],

[8], [28].

$$\mathcal{O}[n][z][x][y] = \varphi\left(\mathcal{B}[z] + \sum_{k=0}^{C_i-1}\sum_{i=0}^{H_f-1}\sum_{j=0}^{W_f-1}\mathcal{I}[n][k][Sx+i][Sy+j] \times \mathcal{F}[z][k][i][j]\right) \tag{2.2}$$

$n \in [0, N)$, $z \in [0, C_o)$, $x \in [0, H_o)$, $y \in [0, W_o)$,

where $\mathcal{O}, \mathcal{I}, \mathcal{F}, \mathcal{B}$ are the ofmaps, ifmaps, filter weights and biases respectively. φ is the activation function which usually is a `ReLU` function or sometimes a `softmax` function where it is desired to represent a probability distribution over a discrete variable [1].

Eq. 2.2 is also valid for the calculation of FC layer, with some additional constraints [8] where $H_i = H_f$, $W_i = W_f$, $H_o = W_o = 1$ and $S = 1$.

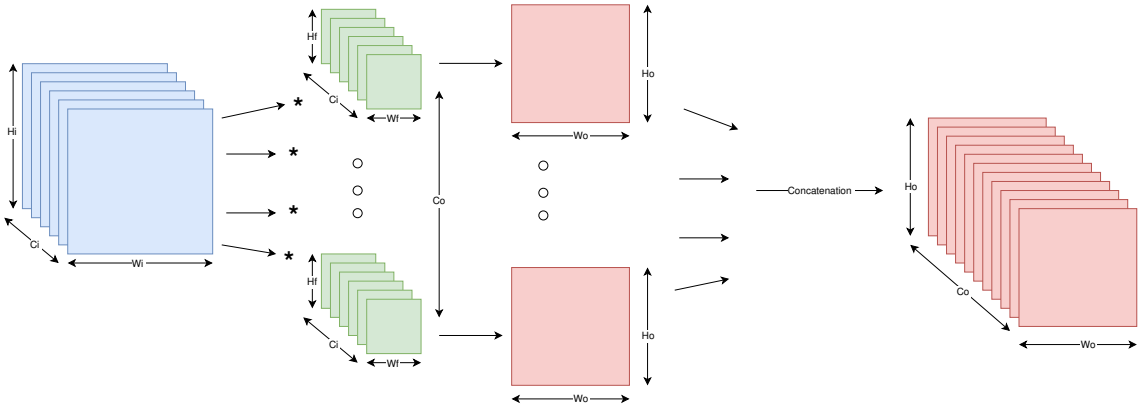


Figure 2.2: Multiple Channel Multiple Kernel (MCMK) Convolution. (Adapted from [3]).

Eq. 2.2 (without the activation function and the bias term) for all batches computes one output (representing a Multiple Channel Single Kernel (MCSK) convolution) out of Multiple Channel Multiple Kernel (MCMK) output [3], see figure 2.2. This represents a nested loop that can be calculated in different ways that are extensions to compute 2D convolution using *Toeplitz Matrix*. The most common way of unrolling this loop is commonly called Image to Column (im2col) [32]–[34] approach. There are different implementations that are based on this approach ranging from batching input images to create one `column matrix` [34] to applying sub-tiles of the column matrix which improve performance if matched against the General Matrix Multiplication (GEMM) resources [33]. Another way to perform MCMK convolution that eliminates the need for data redundancy in im2col is the Kernel to Row (kn2row) algorithm proposed in [3].

2.3 DNN Accelerators

There are different operation precisions that have been used in DNN accelerators for multiplications and accumulations. For training CNNs, the precision required is

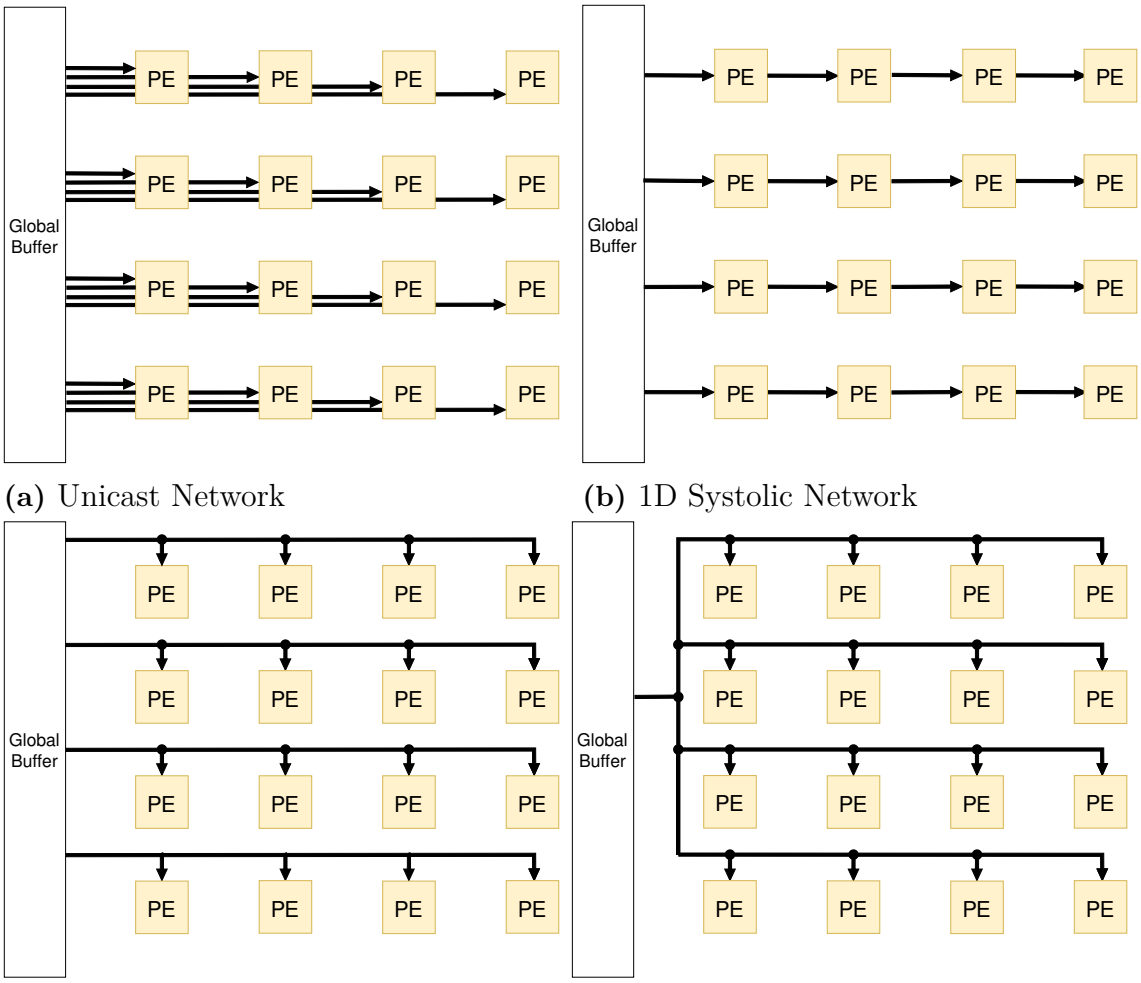
higher and operations are done sometimes with floating points [22] or with Brain Floating Point (bfloat16) [24]. However, for inference purposes as in this thesis work, a quantization scheme like in [35] using 8-bit integer multiplication and 32-bit integer accumulation would be sufficient. The design of DNN accelerators using spatial architectures consists roughly of Processing Elements (PEs) and a network interconnecting different nodes of PEs.

2.3.1 Processing Elements

A PE in spatial DNN accelerators consists mainly of a MAC unit, a local storage unit and sometimes a control part [4], [8], [28]. Different implementations of different dataflows result in different local storage needs, for example different pipeline requirements and data movement implementations result in different needs [2], [25], [36]. Further, the size of the local storage used in some designs takes into account the maximum filter size that can be directly used using that design as in [28]. In general, and since there is a need to reuse at least one datatype to accomplish the main functionality of the architecture and other two datatypes need to be moved in a pipeline, there will be need for local storage elements for all ifmaps, wights and Partial Sums (psums). Also, any delay elements needed will also account as extra registers.

2.3.2 On-Chip Networks

Figure 2.3 shows different Network-on-Chip (NoC) variants that can serve spatial architectures for processing DNNs. Since maximal achievable data reuse of all datatypes, i.e. filter weights, inputs and outputs cannot be achieved simultaneously [8], the design of DNN accelerators is usually done by mixing these topologies targeting different datatypes. The spatial data reuse of unicast networks and systolic networks is higher than multicast and broadcast networks [36], however that requires higher bandwidth between the global buffer and the PE as seen in the figure.



(a) Unicast Network

(b) 1D Systolic Network

(c) 1D Multicast Network

(d) Broadcast Network

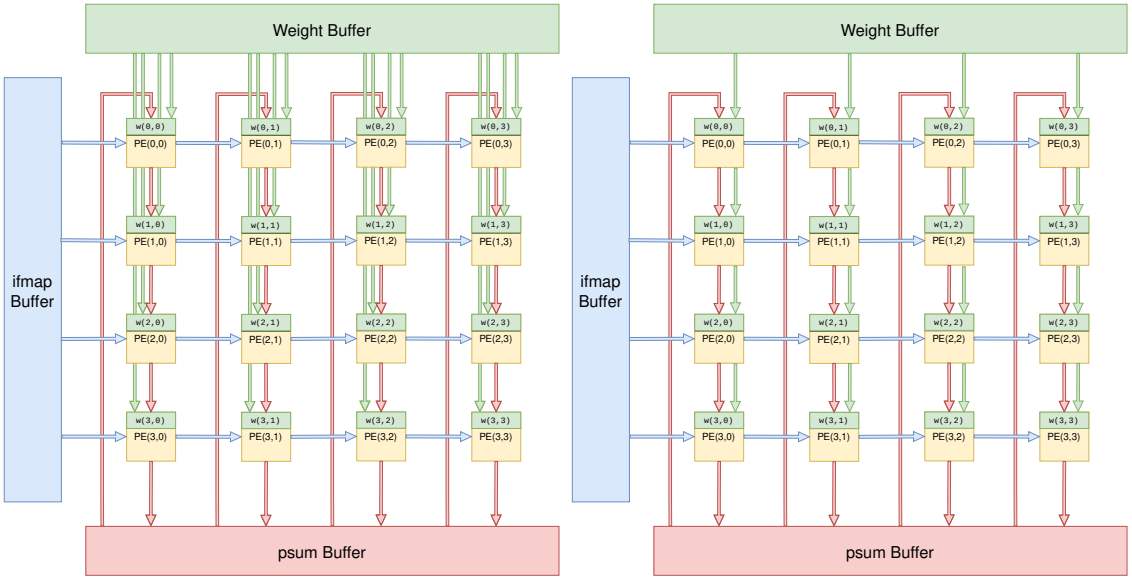
Figure 2.3: Common NoC designs. (Adapted from [36].)

2.4 Dataflows

Different dataflows in the design of hardware accelerators for Machine Learning (ML) applications have been used as discussed previously in section 1.3. The stationariness of dataflows discussed in this section is relative to the local memory storage inside the PE. Other dataflows can be constructed by targeting different datatypes at different memory hierarchy, for example the global buffer.

2.4.1 Weight Stationary

In Weight Stationary (WS) dataflow, the weights stay stationary at each PE during the time of calculating one execution of the array of PEs. This dataflow aims to maximize the reuse of weights stored locally at each PE [8]. Figure 2.4a shows an example of how WS dataflow can be implemented using a unicast network and a 1D systolic network, and figure 2.4b shows a WS dataflow using 2D Systolic Array (SA).



(a) **(b)**

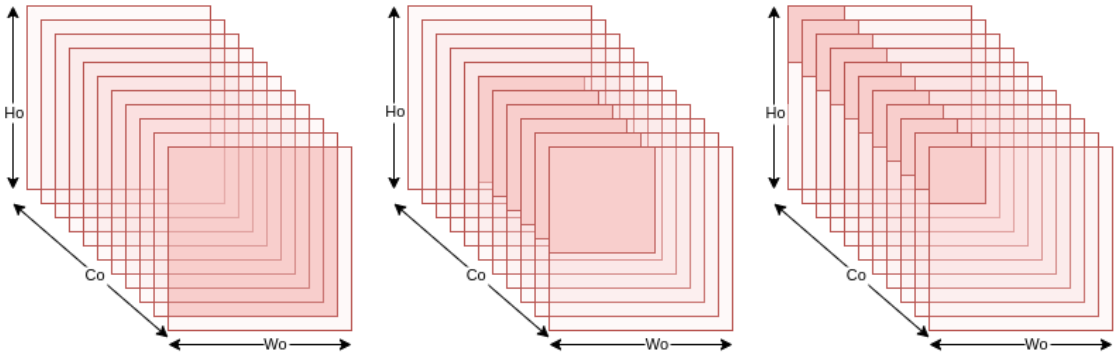
Figure 2.4: WS dataflow using: (a) 1D systolic network for the ifmaps and a unicast network for the weights and (b) 2D systolic network.

2.4.2 Input Stationary

The Input Stationary (IS) dataflow aims to maximize the reuse of ifmaps and the case is similar to that in WS dataflow seen in figure 2.4 replacing the weights with ifmaps.

2.4.3 Output Stationary

The Output Stationary (OS) dataflow aims to maximize the reuse of psums stored locally at each PE [8]. In contrast to WS and IS dataflows that depends on spatial accumulation, the OS dataflow resembles parallelism of SIMD in CPUs and GPUs as it uses temporal accumulation [37]. Three types of OS dataflow can be identified [7], see figure 2.5.



(a) **(b)** **(c)**

Figure 2.5: Different variants of OS dataflow. Colored regions represents parallel output regions. (Adapted from [8].)

The difference between theses types is the number of output channels and output activations where the variant in (a) mainly targets CONV layers and the variant in (c) targets FC layers [8]. Figure 2.6 shows an example implementation for the WS dataflow using 2D systolic network.

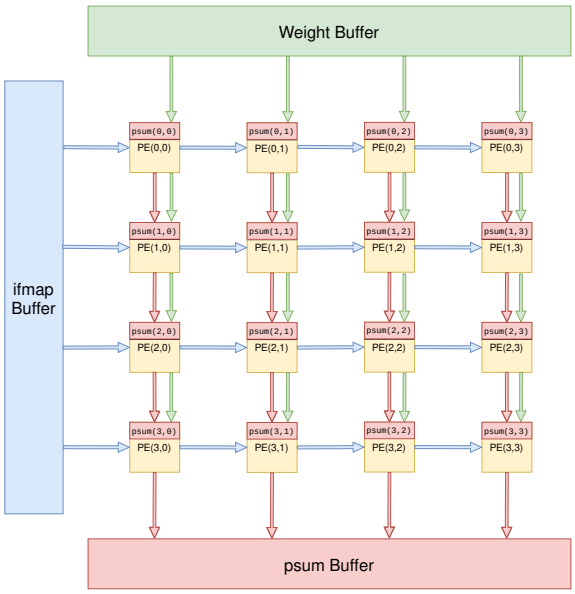


Figure 2.6: OS dataflow using 2D systolic network.

2.4.4 Other Dataflows

The Row Stationary (RS) dataflow first introduced in [7] aims to maximize data reuse across all datatypes. It divides the MACs into subsets of rows that run on the same PE and performs a 1D convolution for that row [38]. This is accomplished using the design of a hierarchical mesh NoC topology using different on-chip networks seen in figure 2.3 and adapt to different bandwidth requirements [36].

In the Tunnel Stationary (TS) dataflow, the input and weight matrices are divided into `1x1xN` units and the partial sums are accumulated over time for the same sliding

window [29]. Other dataflows can be constructed by building different network hierarchies mixing data reuse across different datatypes since the stationariness of a dataflow is relative to a specific level of memory hierarchy [8].

3

Methods

The work flow in the design can be divided into three categories: accelerator design and simulation, implementation on the development platform and MATLAB simulation, see figure 3.1.

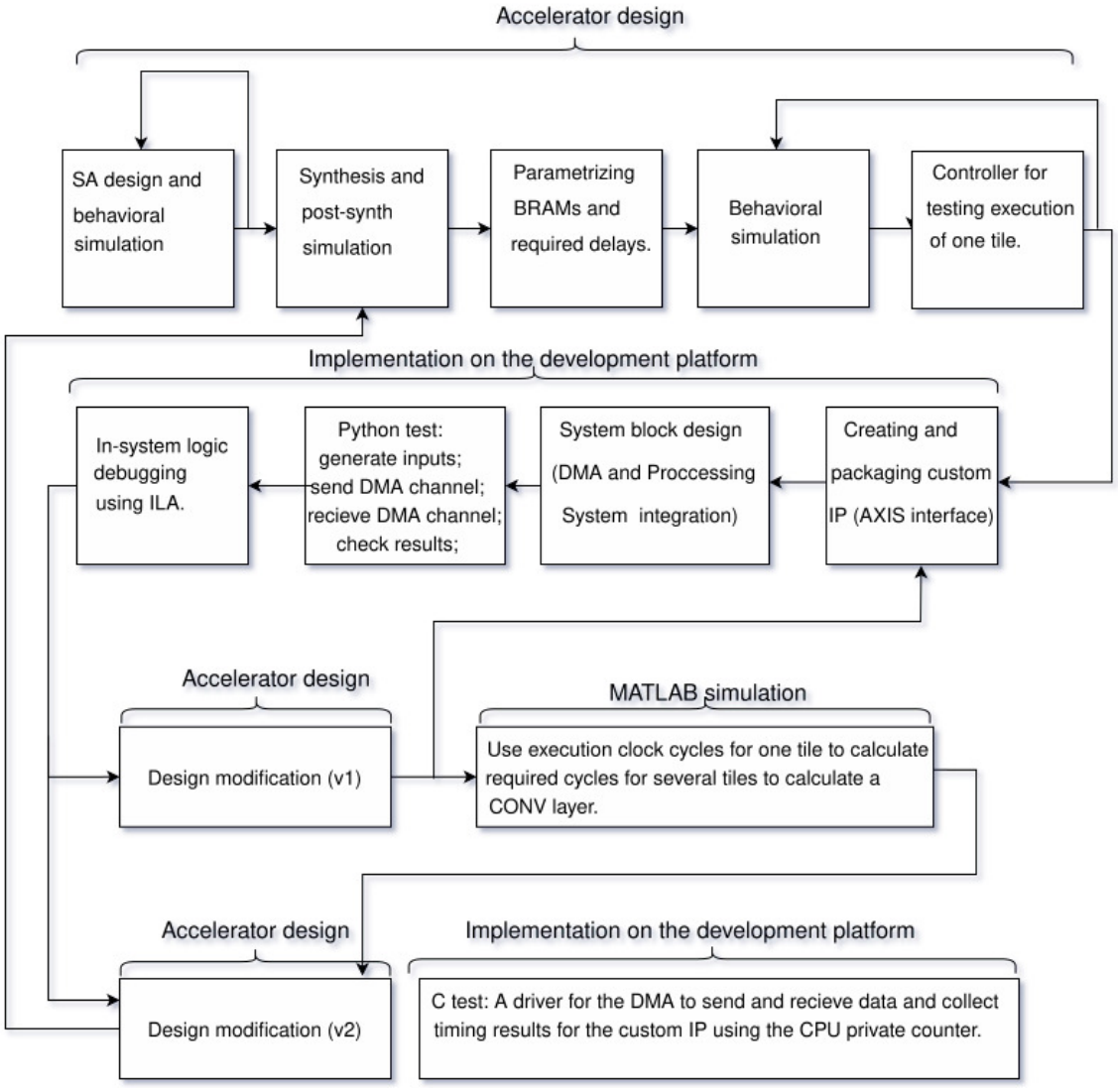


Figure 3.1: Design work flow.

3.1 Accelerator Design and Simulation

The design was written in VHDL-93. Simulation using QuestaSim and RTL analysis using Vivado were the main development tools throughout the thesis work. Later after implementation of the design on the development platform, the simulation results were verified against waveforms using Integrated Logic Analyzer (ILA) IP core [39].

3.2 Implementation on the development platform

3.2.1 Development System Overview

The development board is the PYNQ-Z2 with Zynq-7000 (XC7Z020) SoC [40]. This Soc contains a Processing System (PS) with dual core Cortex-A9 ARM processor and a Programmable Logic (PL) representing the FPGA part of the chip [40]. Arithmetic resources and specifically for MAC operations needed in the design exists the DSP48E1 [41]. For Block RAMs there exists RAMB18E1/RAMB36E1 [42]. Some of the available resources are summarized in table 3.1.

Table 3.1: Some resources available in the XC7Z020 SoC.

Resource	**# available**
Flip-Flops	106,400
Look-Up Tables (LUTs)	53,200
Block RAM (# 36 Kb Blocks)	4.9 Mb (140)
DSP Slices (18 x 25 MACs)	220
DMA Channels	8 (4 dedicated to Programmable Logic)

3.2.2 System implementation (Custom IP & System Block Design)

For testing on the development platform, a custom IP with AXIS [43] interface was created and packaged using Vivado [44]. The accelerator IP was then connected to the processing system of the XC7Z020 SoC through a DMA [45], see figure 3.2. After synthesis, implementation and bit stream generation, the system was loaded to the platform and functionality was first tested using python. In-system logic design debugging were done through ILA IP core [46] at the slave and master AXIS interfaces of the custom IP created for the prototype. Finally, one of the ARM cores available on the SoC was used to run a standalone (without operating system) `C` application to send and receive DMA transfers between the custom IP and the DDRAM and timing results for the execution of the accelerator were collected through Xilinx©Vitis using the ARM core private timer.

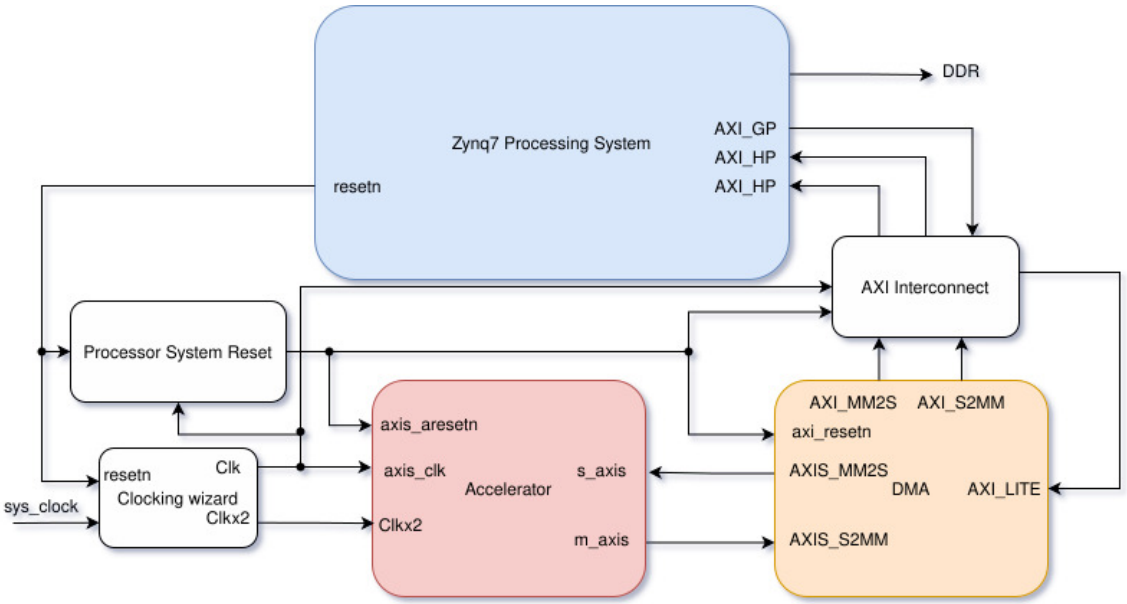


Figure 3.2: System block design.

3.3 MATLAB simulation

For calculation of execution cycles required for a convolutional layer using different Systolic Array (SA) shapes and sizes, an exhaustive testing on hardware for all combinations provided in this thesis would be difficult and time-consuming. Also, as a reference value to compare some hardware test-cases with the calculated results, a MATLAB simulation was done to calculate and plot the result. Other alternatives such as MATPLOTLIB python library could have been used. Given a specific number of DSP/MAC resources, all SA shapes combinations that don't exceed resources is found. The SA execution cycles for all combinations were then calculated along with the required tiles for all combinations for finally calculating the total execution cycles for each layer. A detailed description of the algorithm used can be found in listing 3.1.

```
weight matrix, (MxK);
ifmap matrix, (KxN);
DSP/MAC resources, L;
Combinations, combs = {(SA_Rows, SA_Columns): SA_Rows * SA_Columns
    <= L };

v1:
for all i in SA_Rows
    for all j in SA_Columns
        cycles(i, j) = max(SA_Rows(i), SA_Columns(j)) + 2 (SA_Rows(
    i) + SA_Columns(j)) + SA_Columns(j);
        tiles(i, j) = ceil(K/SA_Rows(i)) * ceil(M/SA_Columns(j)) *
    ceil(N/SA_Columns(j));
        layer_cycles(i, j) = tiles(i, j) * cycles(i, j);
    end for
end for

```

```
v2:
for all i in SA_Rows
    for all j in SA_Columns
        tiles(i, j) = ceil(K/SA_Rows(i)) * ceil(M/SA_Columns(j));
        layer_cycles(i, j) = tiles(i, j) * (max(SA_Rows(i),
   SA_Columns(j)) + 2 (SA_Rows(i) + N)) + M * ceil(N/SA_Columns(j))
   ;
    end for
end for
```

Listing 3.1: Pseudocode for the algorithm used to calculate execution cycles using MATLAB.

The convolutional layers that were used in this setup are summarized in appendix A, tables B.1 to B.4.

4

Design

This chapter presents diagrams of the final design. VHDL code and documentation can be found in Gitlab repository `https://git.chalmers.se/tayyemi/datx05_sub`. Design-specific equations that were extracted throughout this work and used in the VHDL code can be found in appendix A.

4.1 Design Overview

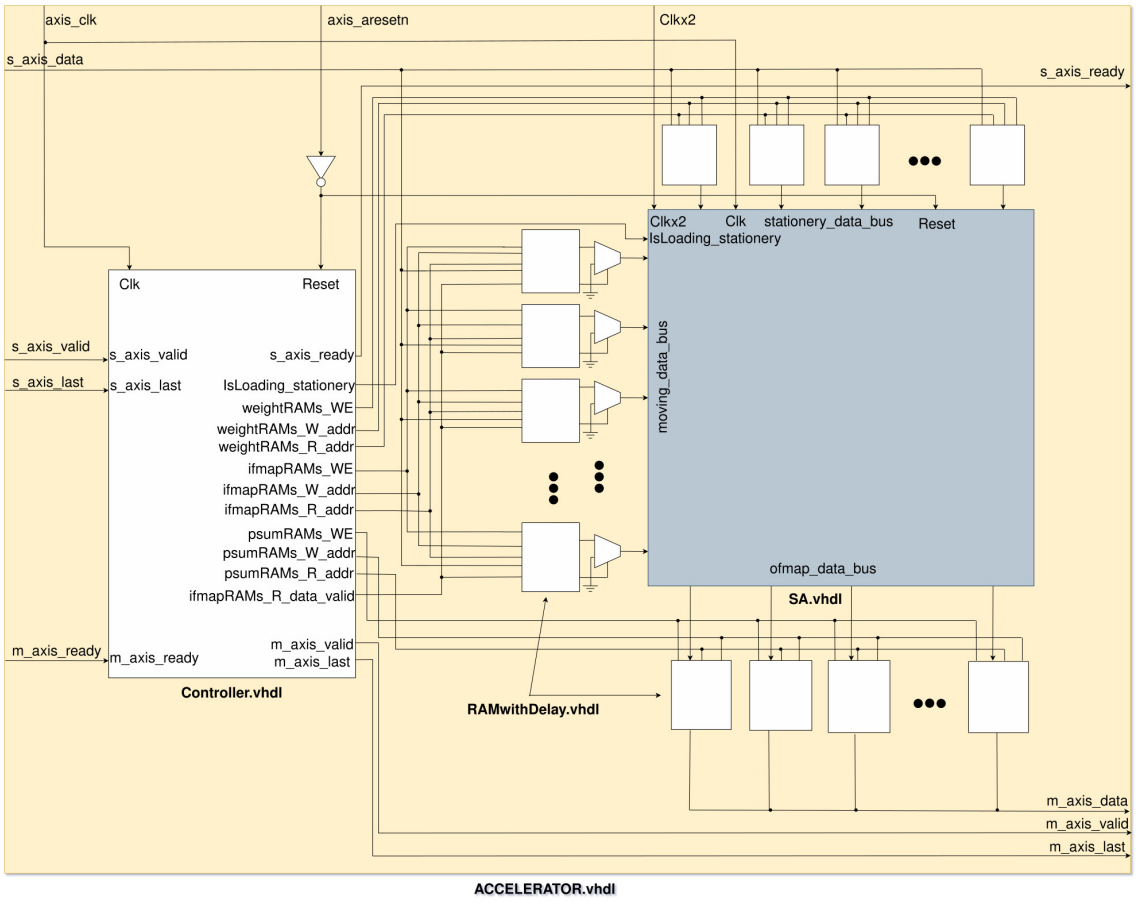


Figure 4.1: Top level module of Accelerator v1.

The top level interface of the accelerator was introduced previously in figure 3.2. Figure 4.1 shows the main components that constitute the accelerator. The design consists of the `SA`, components called `RAMwithDelay`, and the `Controller`. The difference between both versions resides mainly in the logic behind the controller component. Also, additional multiplexers were added in the Systolic Array (SA) component for v2. The accelerator is capable of multiplying `m x n` matrices using `m x n` SA with one matrix stays stationary during the execution of the SA. The following sections present each component in the top level as well as different design components used inside each one. Design modules are also documented in the Gitlab repository `https://git.chalmers.se/tayyemi/datx05_sub`.

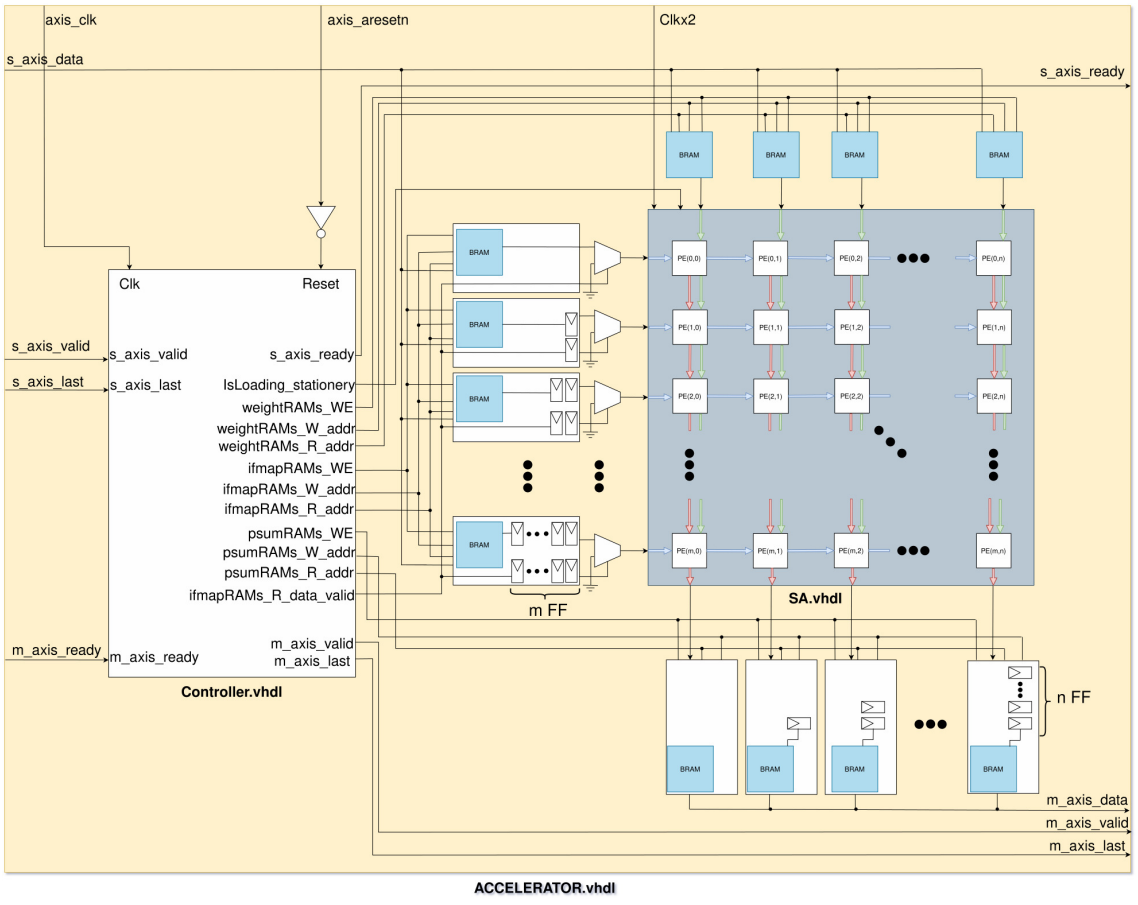


Figure 4.2: Top level module of Accelerator v1 showing delay elements and processing elements components.

Different SA shapes yield different synthesized delay elements inside *RAMwithDelay* components. This is illustrated in figures 4.2 and 4.3 for Accelerator v1 and v2 respectively.

VHDL code parameters that allow generation of different shapes and sizes of the SA can be found in appendix A table A.1.

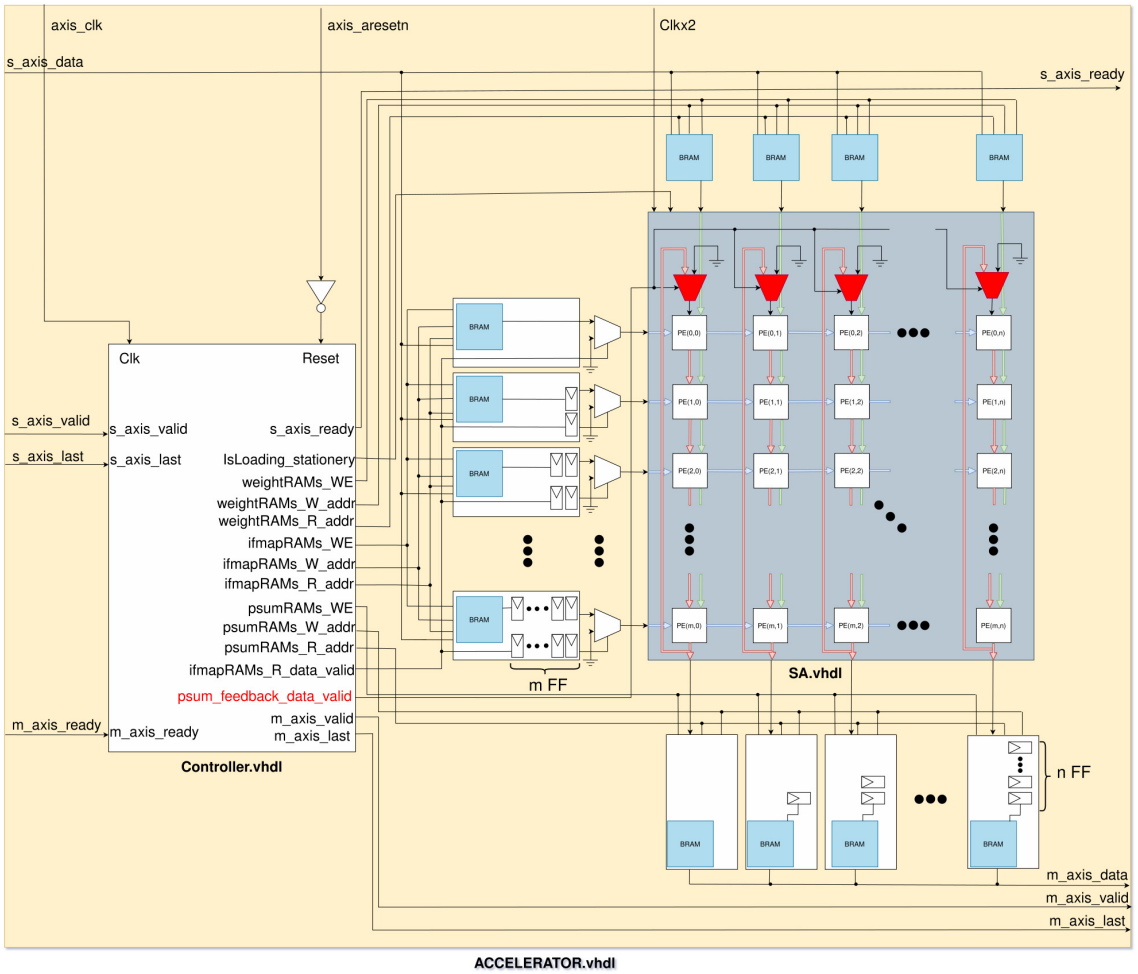


Figure 4.3: Top level module of Accelerator v2.

4.2 Systolic Array

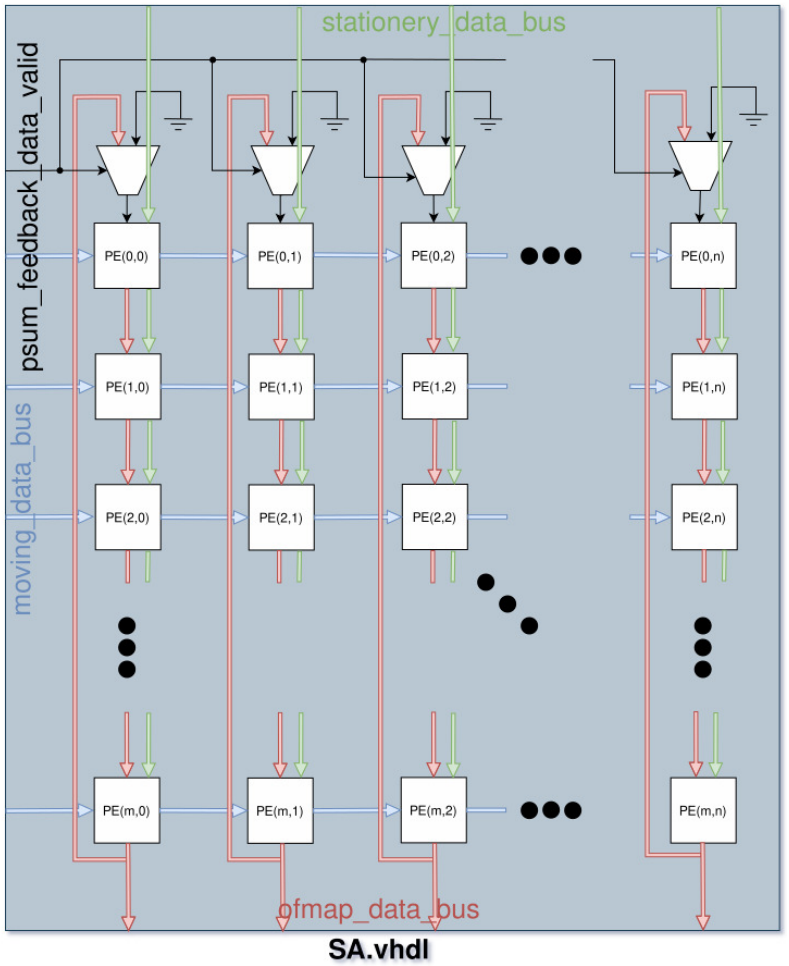


Figure 4.4: `SA` in `Accelerator v2`. `Clk`, `Clkx2`, `Reset` and `IsLoading_stationary` input ports are not seen in the figure. For `Accelerator v1`, the multiplexers with thier select signal from `psum_feedback_data_valid` input port are excluded.

The SA consists of Processing Elements (PEs) and the interconnections between them as well as the connections between the SA interface and PEs at the border.

4.2.1 Processing Element

The PE design is shown in figure 4.5. It multiplies `stationary` and `moving` inputs and adds that to the `psum` input. The Multiply Accumulate (MAC) operation runs at a clock frequency double the rate of the incoming inputs. The reason for that is that for the operation in row `i+1` to be correct, the operation in row `i` must be performed at least one clock cycle before that in row `i+1` so that the output from row `i` is at the `psum` input of row `i+1` at the next clock cycle for the incoming inputs.

To keep HDL source code portable and scalable, inferencing is used for the MAC operation and for the sake of evaluation on the development platform, the attributes needed to synthesize using DSPs are used so that for each inferred MAC unit a DSP is used. The output register seen in figure 4.5 is not synthesized and the functionality is accomplished using the `PREG` at the output of the DSP48E1 [41].

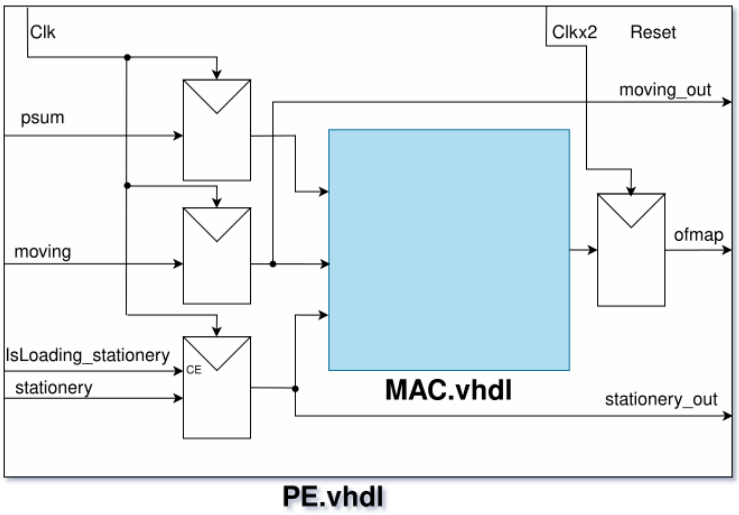


Figure 4.5: Processing Element.

4.2.2 Interconnections

Nine types of interconnections have been identified to make the design parametrizable for (`m x n`) SA as shown in figure 4.6. The PEs at the borders have interconnections to the corresponding spatial interface. For example, in position (`0,0`), the interconnections are with the left `moving_data_bus` and the top `stationary_data_bus` and in position (`m,0`), the interconnections are with the left `moving_data_bus` and the bottom `ofmap_data_bus`. The PEs at the borders but not at the corners have interconnections with the PEs next to them at both sides horizontally, but at one side vertically, for example in position (`0,j`) the horizontal interconnections are with PEs in (`0,j-1`) and (`0,j+1`) which responsible for propagating the `moving` part and the PEs bottom connections are between the `ofmap` output and the `psum` input of the row below. Another vertical interconnections exist to propagete the `stationary` part when loading data at the start of the execution. The PEs at the corners have in turn similar vertical interconnections but at one side horizontally with the PEs next to them. The PEs at the center, (`i,j`), have interconnections at all sides but without any connection to the interface. Additional internal signal is fed from the `ofmap` output of the last row back to `psum_data_bus` of the first row through multiplexers. The select signal of the multiplexers is a signal from the controller to choose between these feedback signals and a grounded input (logic 0) depending upon if the execution is in an intial phase (`TILE = 0`) or not (TILE > 0).

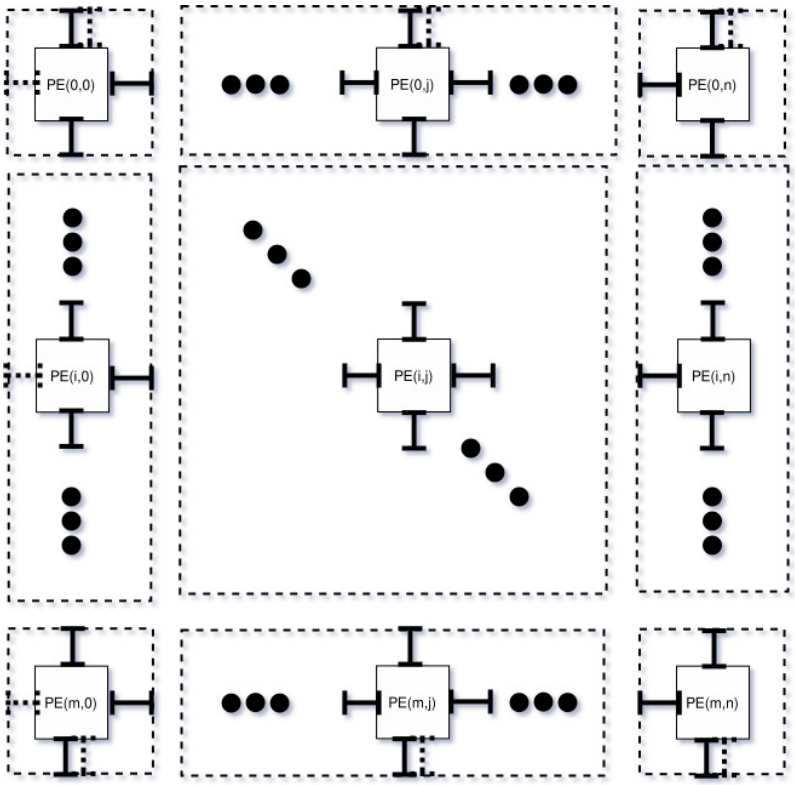


Figure 4.6: Nine types of interconnections in between PEs and between PEs and the SA interface.

4.3 Global buffers

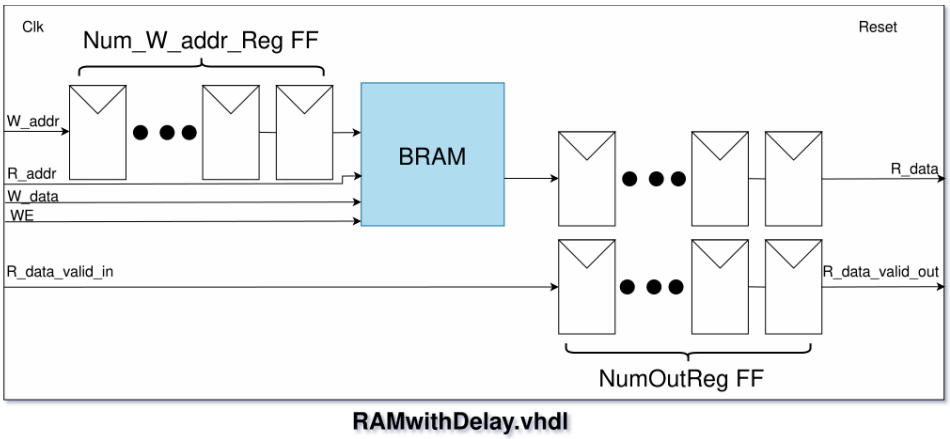


Figure 4.7: RAMwithDelay module.

An important feature that is needed to synchronize SA input feeding and output generation is to be able to derive a formula for clock cycles delay and parametrize that delay depending on the position in the SA. Global buffers and the associated delay is accomplished in the design through a module called `RAMwithDelay`. Input

registers at the `WRITE` line (`W_addr` in the figure) is used in `psumRAMs` and output registers is used in `ifmapRAMs`. No input or output registers exist for `weightRAMs`. The `R_data_valid_in` is a control signal that has similar clock cycles delay and is used as select signal for the multiplexers at the input of the SA to choose either between actual input form the block RAMs or a grounded input indicating zeros, see equation 4.1.

$$\begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \\ b_{2,0} & b_{2,1} & b_{2,2} \\ b_{3,0} & b_{3,1} & b_{3,2} \end{bmatrix} \times \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \tag{4.1}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & a_{0,3} & a_{0,2} & a_{0,1} & a_{0,0} \\ 0 & 0 & 0 & 0 & a_{1,3} & a_{1,2} & a_{1,1} & a_{1,0} & 0 \\ 0 & 0 & 0 & \underbrace{a_{2,3} \quad a_{2,2} \quad a_{2,1} \quad a_{2,0}}_{SA\_Columns} & & & & 0 & 0 \end{pmatrix}$$

$$\underbrace{\qquad}_{SA\_Columns - 1} \qquad \underbrace{\qquad}_{SA\_Rows - 1}$$

Equation 4.1 is an example for multiplying two matrices with SA shape as the shape of `a matrix`, (`SA_Rows`, `SA_Columns`) = (3, 4) in Accelerator v1. `b matrix` is transposed and enters the SA and stays stationary there. The design uses systolic network also for `weightRAMs`. This means that it needs *SA_Rows* clock cycles for the stationary matrix (`b matrix`) to be loaded. This can easily be modified to require only a clock cycle instead at the expense of increasing the bandwidth and `weightRAMs` utilization. `a matrix` is flipped horizontally and enters the SA from left as shown in the (3 x 9) matrix. The required clock cycles for this part is seen on this matrix. Additional one clock cycle is needed for BRAM READ operation.

$$\begin{aligned} v1\_cycles\_READ\_input &= SA\_Rows + (SA\_Columns - 1 \\ &\quad + SA\_Columns + SA\_Rows - 1) + 1 \\ &= 2 \cdot (SA\_Rows + SA\_Columns) - 1 \end{aligned} \tag{4.2}$$

In Accelerator v2, and using (`SA_Rows`, `SA_Columns`) = (2, 2) the 2x2 tile of `b matrix` is also given transposed as before and requires *SA_Rows* clock cycles to load the `stationary` matrix. The first tile of `a matrix` would be as the following (see equation 4.4 for needed clock cycles for one tile in v2):

$$\begin{bmatrix} b_{0,0} & b_{1,0} \\ b_{0,1} & b_{1,1} \end{bmatrix} \times \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \tag{4.3}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & a_{0,3} & a_{0,2} & a_{0,1} & a_{0,0} \\ 0 & 0 & 0 & \underbrace{a_{1,3} \quad a_{1,2} \quad a_{1,1} \quad a_{1,0}}_{N} & & & & 0 \end{pmatrix}$$

$$\underbrace{\qquad}_{N - 1} \qquad \underbrace{\qquad}_{SA\_Rows - 1}$$

$$\begin{aligned} v2\_tile\_cycles\_READ\_input &= SA\_Rows + (N - 1 \\ &\quad + N + SA\_Rows - 1) + 1 \\ &= 2 \cdot (SA\_Rows + N) - 1 \end{aligned} \tag{4.4}$$

4.4 The Controller

The controller in the first version of the accelerator supports multiplying one tile of the size of the SA. For multiplying larger matrices, a software interface that uses the output of one tile and feed that back to the accelerator is needed. The state machine used and the documentation for the controller as well as other parts of the design can be found in appendix A. Vectors of both matrices to be multiplied shares the s_axis_data and during the `Get/Get_extra` states, data are loaded into block RAMs. In v1, the `psum` input of the first row is always grounded. The output is read from `psumRAMs` to `m_axis_data` during the `Give` state.

To write matrices (ifmaps and weights) data coming on the s_axis_data line to the block RAMs, a number of clock cycles that equals the maximum of (`SA_Rows`, `SA_Columns`) is needed. First in the `Get` state, a number of elements in both matrices that equals the minimum of (`SA_Rows`, `SA_Columns`) is written and after that the rest of the larger matrix is written in the `Get_extra` state. An additional clock cycle for BRAM operation is needed as before.

$$\begin{aligned} cycles\_WRITE\_input &= v1\_cycles\_WRITE\_input \\ &= v2\_tile\_cycles\_WRITE\_input \\ &= \max(SA\_Rows, SA\_Columns) + 1 \end{aligned} \tag{4.5}$$

Finally for reading `ofmaps` from `psumRAMS` at the `GIVE` state the required cycles are explained in equation 4.6 and 4.7.

$$v1\_cycles\_READ\_output = SA\_Columns \tag{4.6}$$

$$v2\_cycles\_READ\_output = M \cdot \lceil N/SA\_Columns \rceil \tag{4.7}$$

Using equations 4.2, 4.4 and 4.5 we can calculate the required clock cycles needed for the execution of both versions of the accelerator, see equation 4.8 and 4.9

$$\begin{aligned} v1\_cycles = \max(SA\_Rows, SA\_Columns) + \\ 2 \cdot (SA\_Rows + SA\_Columns) + SA\_Columns \end{aligned} \tag{4.8}$$

$$\begin{aligned} v2\_cycles = v2\_TILES \cdot (\max(SA\_Rows, SA\_Columns) + 2 \cdot (SA\_Rows + N)) + \\ M \cdot \lceil N/SA\_Columns \rceil \end{aligned} \tag{4.9}$$

5

Results

The results obtained in this chapter use 8-bit integers on the inputs. The operation is 8-bit multiplication with 32-bit accumulation. This gives 32-bit signed integers at the output. The methodology used is the one described in the methodology chapter and the setup is as explained in figure 3.2. Additional details are mentioned in each of the following sections.

5.1 Simulation

In this section, the simulation results of both versions of the accelerator are shown using example matrices. Figures 5.1 and 5.2 shows that for 2x3 and 3x2 Systolic Arrays (SAs) respectively in Accelerator v1. For Accelerator v2, simulation result of one tile using 2x2 SA as a partial result of multiplying two matrices of 4x4 sizes can be seen in figure 5.3. This illustrates the functionality of the design with squared as well as non-squared systolic arrays for signed multiplication. In these simulations, the actual matrices to be multiplied are given first followed by the waveform viewed using gtkwave. Simulation data are from QuestaSim as mentioned previously. Similar waveforms were obtained using ILA IP [39] in Vivado.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 9 & 12 & 15 \\ 19 & 26 & 33 \\ 29 & 40 & 51 \end{bmatrix} \tag{5.1}$$

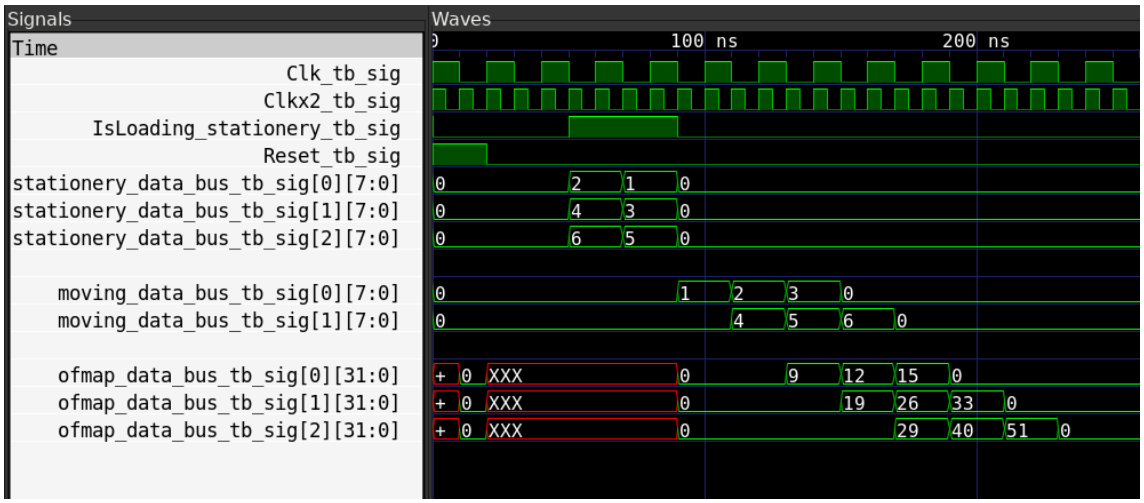


Figure 5.1: Simulation results of 2x3 SA in Accelerator v1.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 22 & 28 \\ 49 & 64 \end{bmatrix} \tag{5.2}$$

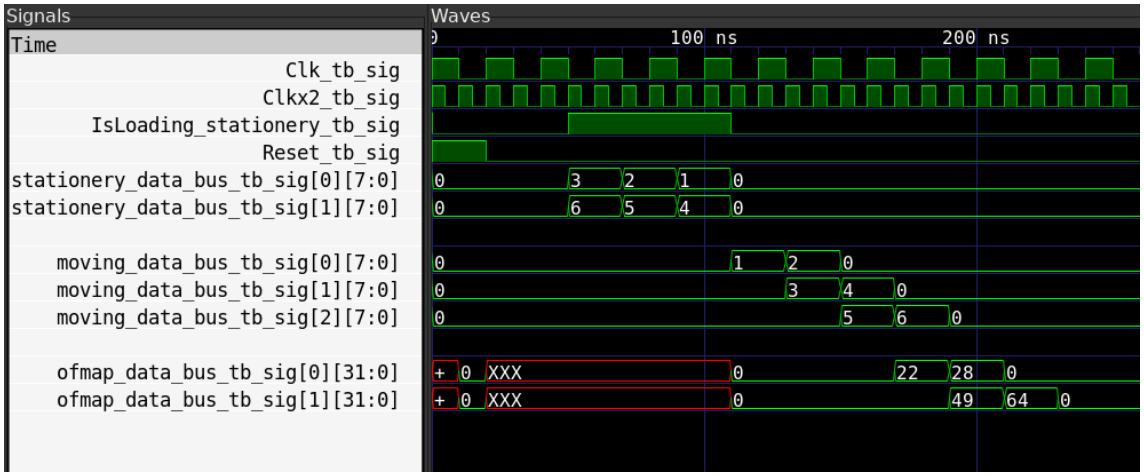


Figure 5.2: Simulation results of 3x2 SA in Accelerator v1.

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \times \begin{bmatrix} -1 & -2 & -3 & -4 \\ -5 & -6 & -7 & -8 \\ -9 & -10 & -11 & -12 \\ -13 & -14 & -15 & -16 \end{bmatrix} = \begin{bmatrix} -62 & -68 & -74 & -80 \\ -174 & -196 & -218 & -240 \\ -286 & -324 & -362 & -400 \\ -398 & -452 & -506 & -560 \end{bmatrix} \tag{5.3}$$

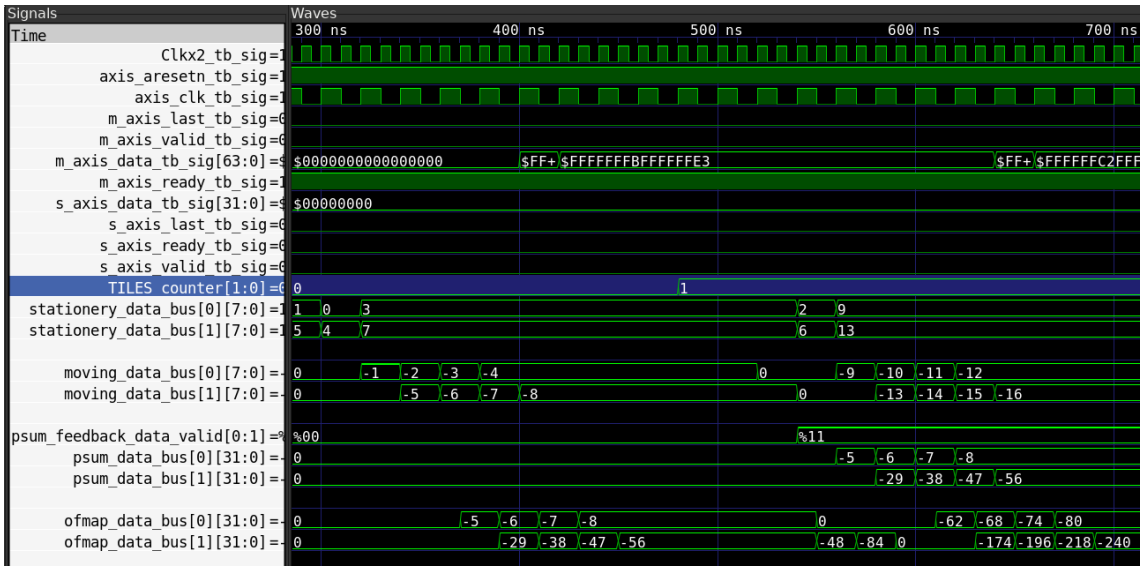


Figure 5.3: Top level simulation of M4K4N4 2x2 SA in Accelerator v2. Values are in hexadecimal on `axis_data` buses. Signals from the SA module is also seen on the figure which shows the first two tiles calculating the top rows of the result. Results for these signals are in signed decimal.

5.2 Utilization

Targeting the XC7Z020 SoC, some example SA shapes were synthesized and the results are summarized in tables 5.1 and 5.2. Attributes to control the logic inference of DSPs and BRAMs were used. The results of theses resources reflects the design and conforms with figures 4.2 and 4.3. Taking the 5x3 case for example, the systolic array is constructed using 15 Processing Elements (PEs) which each contains a DSP. The BRAMs are distributed as: 3 `weightRAMs`, 3 `psumRAMs` and 5 `ifmapRAMs`. Sometimes the tool does not follow synthesis attributes and solve the problem with less DSPs as in the case of 14x14 in table 5.1 but it fall back to follow that when it

runs out of BRAM resources as in table 5.2.

Table 5.1: Post-synthesis utilization summary for some example shapes in Accelerator v1.

SA shape (SA_Rows x SA_Columns)	**LUTs** (53,200)[1]	**Flip-Flops** (106,400)[1]	**Block RAM (RAMB18)** (280)[1]	**DSPs** (220)[1]
2x2	24 (0.05%)	149 (0.14%)	6 (2.14%)	4 (1.82%)
5x3	65 (0.12%)	709 (0.67%)	11 (3.93%)	15 (6.82%)
4x4	38 (0.07%)	713 (0.67%)	12 (4.29)	16 (7.27%)
6x4	93 (0.17%)	1140 (1.07%)	14 (5.00%)	24 (10.91%)
14x14	282 (0.43%)	8750 (8.22%)	40 (14.29%)	182 (82.73%)
22x10	436 (0.82%)	10963 (10.30%)	42 (15%)	220 (100%)

[1] Values in parenthesis are the total available resources in the XC7Z020.

Table 5.2: Post-synthesis utilization summary for some example shapes in Accelerator v2.

MxKxN	**SA shape** (SA_Rows x SA_Columns)	**LUTs** (53,200)[1]	**Flip-Flops** (106,400)[1]	**Block RAM (RAMB18)** (280)[1]	**DSPs** (220)[1]
4x4x4	2x2	184 (0.35%)	367 (0.34%)	6 (2.14%)	4 (1.82%)
200x200x200	5x3	1654 (3.11%)	1583 (1.49%)	140 (50%)	16 (7.27%)
100x1000x100	4x4	1535 (2.89%)	1681 (1.58%)	160 (57.14%)	17 (7.73%)
256x512x256	4x4	2774 (5.21%)	2012 (1.89%)	256 (91.43%)	16 (7.27%)
128x1024x128	14x14	5231 (9.83%)	11077 (10.41%)	280 (100%)	196 (89.09%)
128x1024x128	22x10	4087 (7.68%)	12033 (11.31%)	208 (74.29%)	220 (100%)

[1] Values in parenthesis are the total available resources in the XC7Z020.

5.3 Custom IP

The accelerator was wrapped in a custom IP in Vivado with AXIS-4 [43] interface and connected to the rest of the system as in figure 3.2. Both accelerators have the same interface. Figures 5.4 and 5.5 are the resulting user interface to the accelerators. The customization parameters are also shown in the figures.

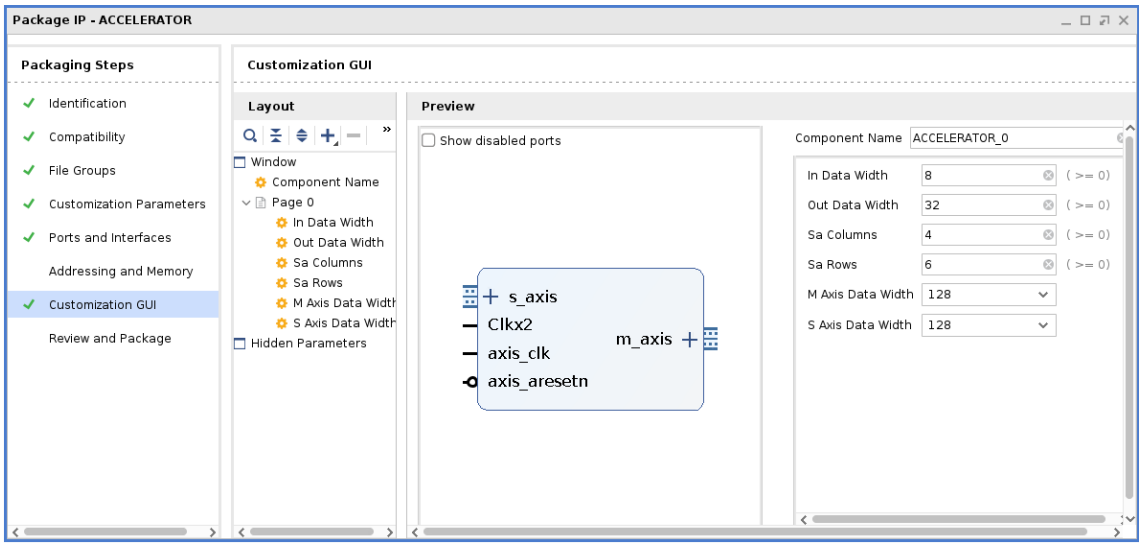


Figure 5.4: Custom IP of the prototype in v1.

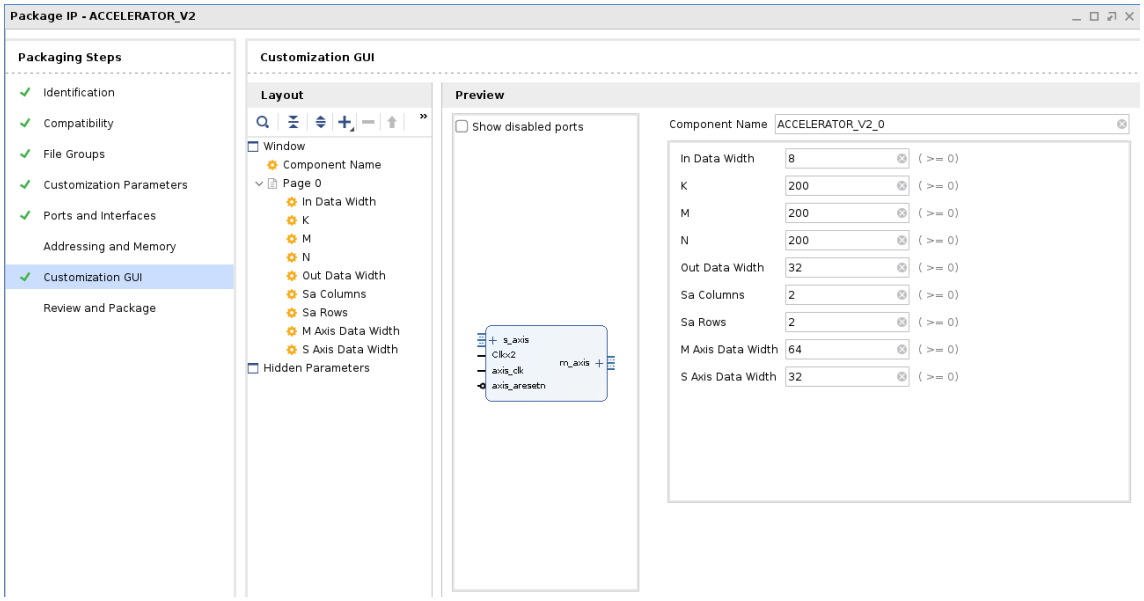


Figure 5.5: Custom IP of the prototype in v2. Notice the extra customization parameters, `M`, `K` and `N`.

The system shown in figure 3.2 represents example deployment case but the IP can be used in different setup as any other IP with AXIS interface. The user of this IP should make sure that the product of `SA_Rows` and `SA_Columns` does not exceed MAC resources. Further, the bit width of the inputs to the available scalar multiplication resources specific to each FPGA determines the maximum `IN_DATA_WIDTH` that can be chosen. For example, the DSP48E1 available in the Zynq-7000 SoC has 18 x 25 MACs and `IN_DATA_WIDTH` cannot exceed that. Other customization parameters such as `M, K, N` have direct relation to increase BRAMs utilization and the equations in table A.1 can be used for estimation so that the utilization does not exceed the available resources.

5.4 Execution Cycles

Using the previously extracted equations from the design, see subsection 4.3 and appendix A, the following represents the time required to execute both versions of the accelerators using example SA shapes and convolutional layers from some common nets, namely AlexNet, ResNet18, ResNet50 and VGG16. All CNN parameters that were used in this analysis can be found in appendix B. Since the design is symmetric regarding both input ports, the `stationary_data_bus` and `moving_data_bus`, it can be used as Input Stationary (IS) and Weight Stationary (WS). In the following analysis and experiments, the design was used as WS. Using this dataflow and squared SA shapes, the total execution time for all convolutional layers of the four CNNs is found. The optimal shape that results in less execution time is also found for each convolutional layer of each CNN. A comparison for the performance gain from executing the optimal shape versus the largest squared matrix that fits in the resources available on the development board (14x14) will be presented for all nets. Results for AlexNet will be included in this section while the results for ResNet18, ResNet50 and VGG16 can be found in appendix C.

5.4.1 Accelerator v1

Given a specific number of DSP/MAC resources, a convolutional layer can be calculated using the Image to Column (im2col) approach using different shapes of the SA. Different shapes results in different number of required tiles, see equation A.15. Therefore, the execution of a layer using different shapes requires different number of clock cycles. To find the optimal SA shape that result in the minimum required clock cycles, the algorithm mentioned in listing 3.1 was used.

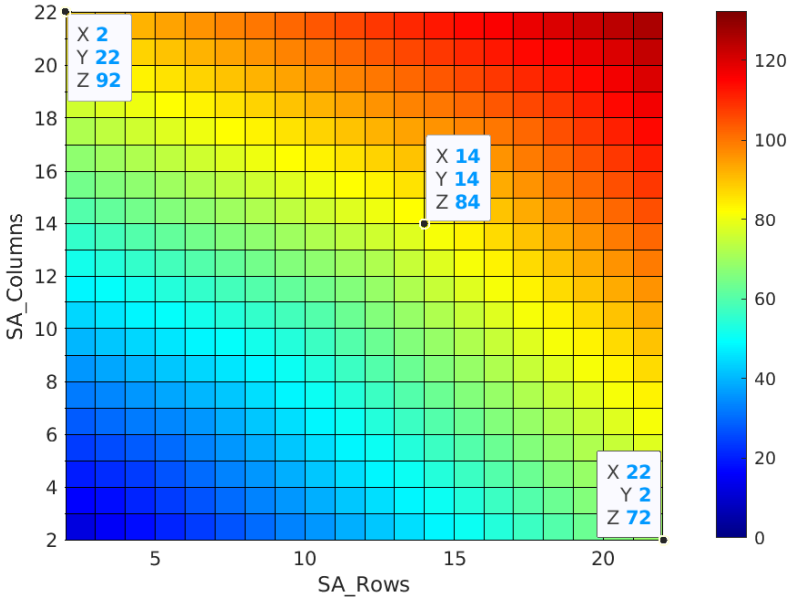


Figure 5.6: SA execution cycles in Accelerator v1. The color bar indicates the number of system clock cycles.

Table 5.3: Measured execution time of Accelerator v1 including DMA transfers at 100 MHz for different SA shapes using INT8 inputs and INT32 outputs.

SA shape	Measured time inc. DMA [us]	Calculated time (Accelerator only) [us]	Difference (DMA transfer+) [us] (%)
2 x 2	3.91	0.12	3.79 (96.93)
8 x 8	4.59	0.48	4.11 (89.54)
14 x 14	6.19	0.84	5.35 (86.43)
11 x 20	8.56	1.02	7.54 (88.08)

Table 5.4: Measured execution time of Accelerator v1 including DMA transfers at 200 MHz for different SA shapes using INT8 inputs and INT32 outputs.

SA shape	Measured time inc. DMA [us]	Calculated time (Accelerator only) [us]	Difference (DMA transfer+) [us] (%)
2 x 2	2.72	0.06	2.66 (97.79)
8 x 8	3.2	0.24	2.96 (92.50)
14 x 14	3.88	0.42	3.46 (89.17)
11 x 20	5.38	0.51	4.87 (90.52)

Figure 5.6 shows a plot of the cycles required to execute one tile. We notice that the cycles increases rapidly if we move in the direction of squared matrices. This is symmetric here since no other parameters rather than the `SA_Rows` and `SA_Columns` involved in the clock cycles required to execute one tile, see equation 4.8. Table 5.3 and 5.4 show results on hardware, also for one tile, at clock frequency of 100 MHz and 200 MHz respectively. We notice in this case that the time required for DMA transfers and other software overhead represents about 90% of the execution time of one tile. Since the execution of the accelerator is in the order of tenths of microseconds, this portion of difference decreases when executing several tiles by increasing the size of the sent packet, see table 5.14.

Table 5.5: AlexNet time execution at 100 MHz with SA shape (14x14) for different convolutional layers.

M x K x N	Required tiles	Time [ms]
96 x 363 x 3025	39494	33.17
256 x 2400 x 729	173204	145.49
384 x 2304 x 169	60060	50.45
384 x 3456 x 169	89908	75.52
256 x 3456 x 169	61009	51.24
Total		355.88

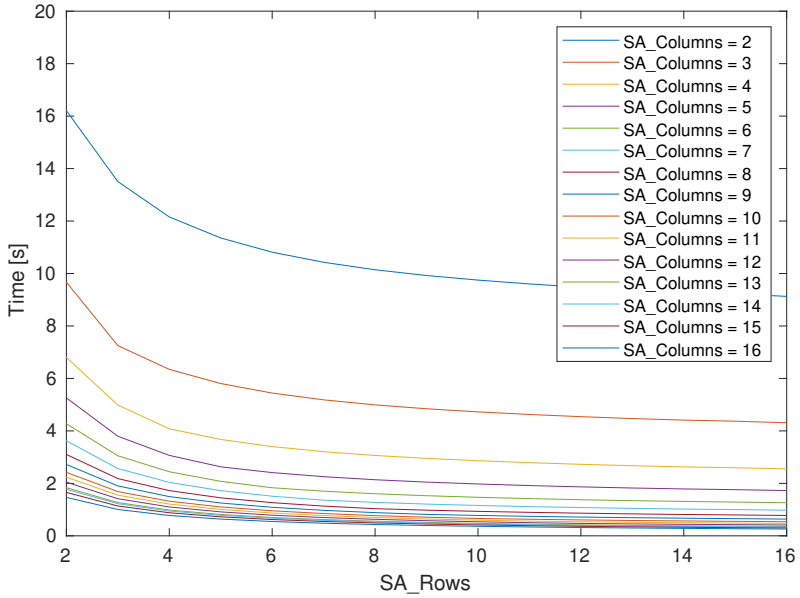


Figure 5.7: AlexNet total execution time at 100 MHz clock frequency with constant `SA_Columns` and variable `SA_Rows` in Accelerator v1.

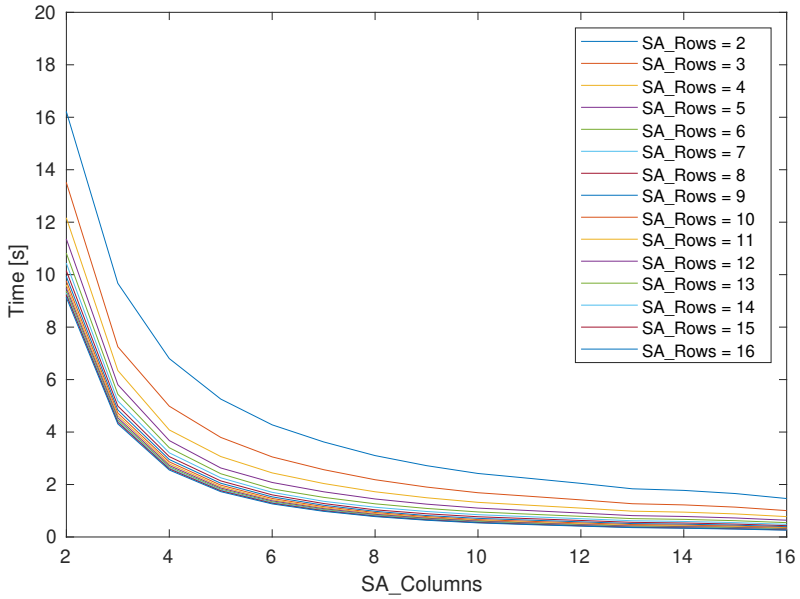
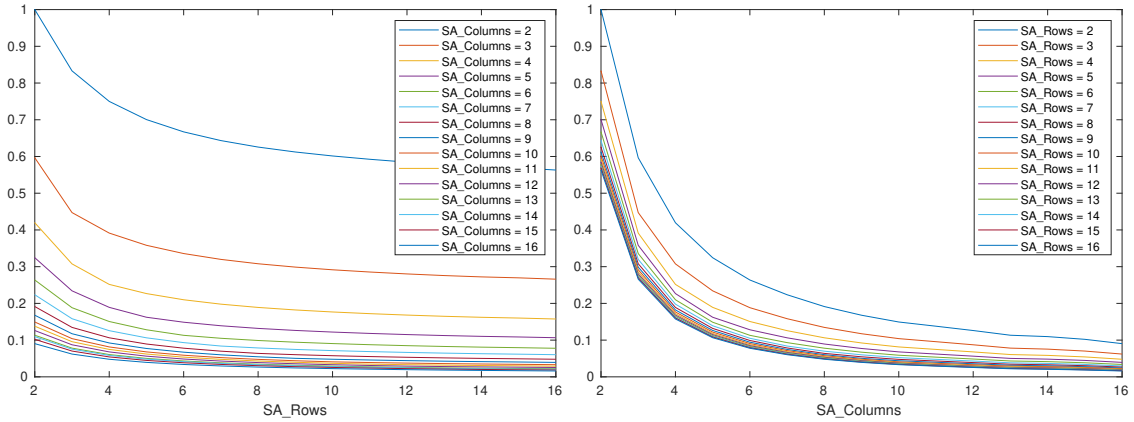


Figure 5.8: AlexNet total execution time at 100 MHz clock frequency with constant `SA_Rows` and variable `SA_Columns` in Accelerator v1.

Figures 5.7-5.8 shows the total execution time at clock frequency of 100 MHz calculated for all convolutional layers of AlexNet. We notice that increasing the number of `SA_Columns` decreases the execution time faster than increasing the number of `SA_Rows`. While the execution of one tile is approximately symmetric between `SA_Rows` and `SA_Columns` (figure 5.6 and eq. 4.8) the number of required tiles, $v1\_TILES \propto \lceil 1/SA\_Columns \rceil^2$ while $v1\_TILES \propto \lceil 1/SA\_Rows \rceil$ (eq. A.15)

which explains the asymmetry found in figures 5.7-5.8. A breakdown of the results of different layers in the case of SA shape 14x14 exists in table 5.5.



(a) **(b)**

Figure 5.9: AlexNet normalized execution cycles with (a) constant `SA_Columns` and variable `SA_Rows` and (b) constant `SA_Rows` and variable `SA_Columns` in Accelerator v1.

Table 5.6: AlexNet optimal SA shape for different convolutional layers given 220 DSPs. Time is at 100 MHz clock frequency.

M x K x N	**Optimal SA shape**	**Required tiles**	**Time [ms]**
96 x 363 x 3025	11 x 20	25080	25.58
256 x 2400 x 729	10 x 22	97920	105.75
384 x 2304 x 169	10 x 22	33264	35.92
384 x 3456 x 169	10 x 22	49824	53.80
256 x 3456 x 169	10 x 22	33216	35.87
Total			256.94

Table 5.6 shows the optimal SA shape that results in less execution time given the DSP resources on the development board to execute AlexNet convolutional layers. We notice in figure 5.9 that even with slightly higher resources than that available on the development board (225) to use a squared SA that result in full utilization of DSPs (15x15), the non-squared 10x22 case results in less execution time. Results for ResNet18, ResNet50 and VGG16 can be found in appendix C.1.

Table 5.7: Total time execution at 100 MHz with SA shape (14x14) versus the optimal found shapes for different nets' convolutional layers in Accelerator v1.

SA shape	**14x14**	**Optimal shape**	**Performance gain**
Net	**Time [ms]**	**Time [ms]**	[%]
AlexNet	355.89	256.94	27.80
ResNet18	796.49	551.74	30.73
ResNet50	1648.85	1151.30	30.18
VGG16	4946.28	3563.93	27.95

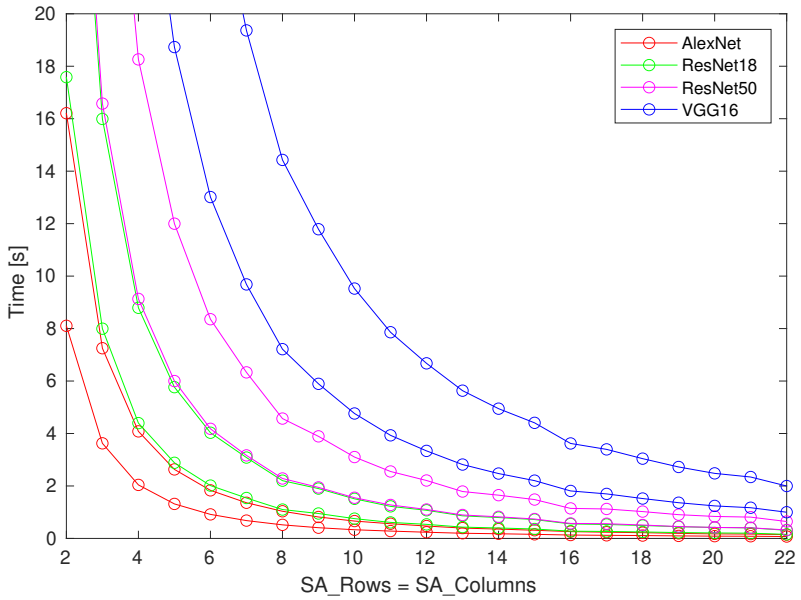


Figure 5.10: Total execution time for all convolutional layers for some common nets using squared SA shapes with system clock frequency of 100 MHz (upper curve) and 200 MHz (lower curve) for each net.

Figure 5.10 shows the execution time at 100 MHz and 200 MHz for squared SA shapes. We notice that different nets have different SA- square's side value at which the slope of the curve is minus one. At these values, increasing the used resources results in less contribution to the time minimization. However, this depends on the application since the time shown is in the order of seconds and the time saved from several executions may be significant to the application and argument for increasing the used resources. A comparison of the total execution time at 100 MHz with SA shape (14x14) versus the optimal found shapes can be found in table 5.7.

If we expand the studied range in figures 5.7- 5.9 and C.1- C.6 and starting from a squared SA consider doubling SA_Columns first, then SA_Rows and finally both of them we notice different behavior between different CNNs and across different ranges, see figure 5.11.

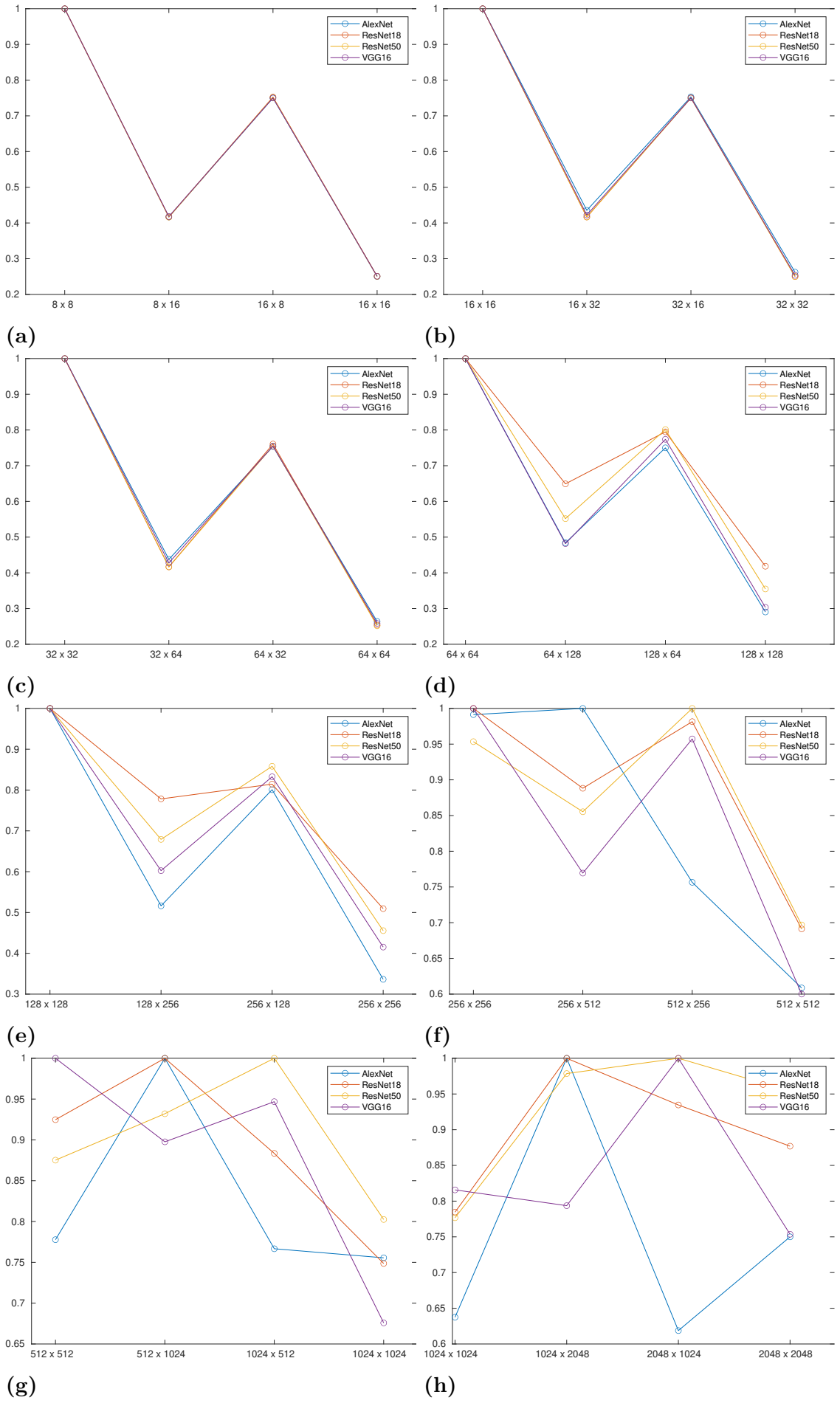


Figure 5.11: Normalized execution cycles for different CNNs across different ranges.

5.4.2 Accelerator v2

In the second version, the `N` dimension of the `MxKxN` arrangement is now packed inside the accelerator. The required internal tiles (internal in the accelerator compared to the external tiles for Accelerator v1 done at the software level) is therefore less than in v1, see equation A.16. As soon as one tile is loaded in the PEs' RegFile, Input Feature Maps (ifmaps) enter thereafter the PEs taking care of the `N` dimension, see equation 4.3.

Table 5.8: AlexNet time execution at 100 MHz with SA shape (14x14) for different convolutional layers.

M x K x N	**Required tiles**	**Time [ms]**
96 x 363 x 3025	182	11.29
256 x 2400 x 729	3268	49.15
384 x 2304 x 169	4620	17.60
384 x 3456 x 169	6916	26.33
256 x 3456 x 169	4693	17.86
Total		122.25

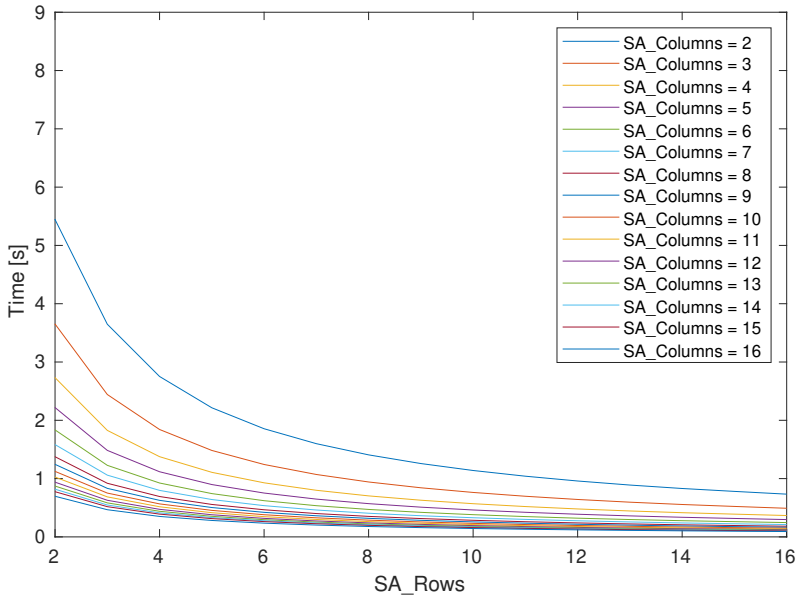


Figure 5.12: AlexNet total execution time at 100 MHz clock frequency with constant `SA_Columns` and variable `SA_Rows` in Accelerator v2.

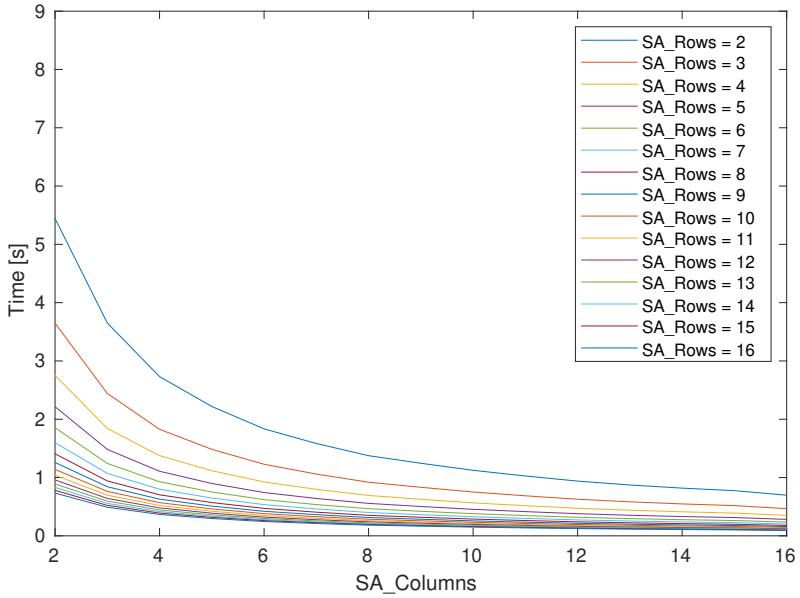
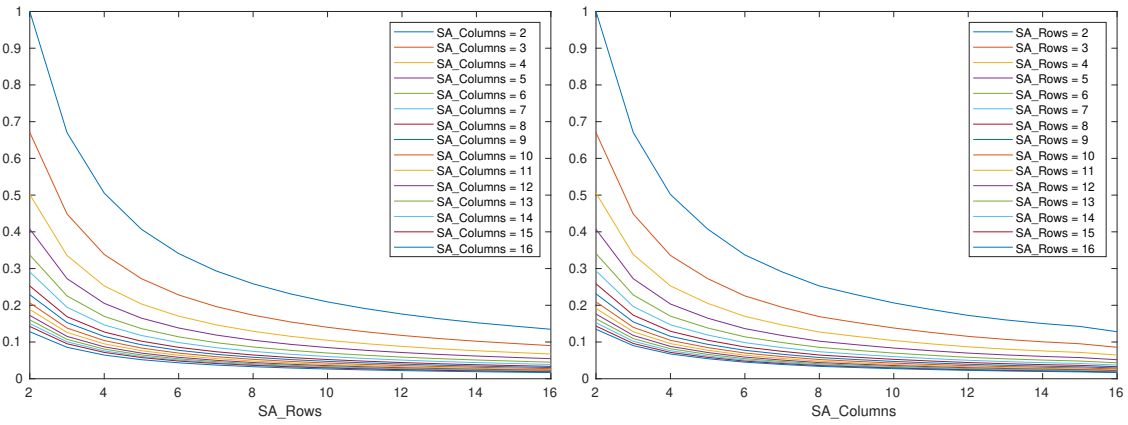


Figure 5.13: AlexNet total execution time at 100 MHz clock frequency with constant `SA_Rows` and variable `SA_Columns` in Accelerator v2.

Figures 5.12-5.13 show the total execution time at clock frequency of 100 MHz calculated for all convolutional layers of AlexNet. We notice that the case now is approximately symmetric for the execution time as a function of `SA_Rows` and `SA_Columns` with advantage for increasing `SA_Columns` to decrease execution time, see eq. 4.9 and A.16. A breakdown of the results of different layers in the case of SA shape 14x14 exists in table 5.8.



(a) **(b)**

Figure 5.14: AlexNet normalized execution cycles with (a) constant `SA_Columns` and variable `SA_Rows` and (b) constant `SA_Rows` and variable `SA_Columns` in Accelerator v2.

Table 5.9: AlexNet optimal SA shape for different convolutional layers given 220 DSPs. Time is for 100 MHz clock frequency.

M x K x N	**Optimal SA shape**	**Required tiles**	**Time [ms]**
96 x 363 x 3025	11 x 20	165	10.19
256 x 2400 x 729	11 x 20	2847	42.79
384 x 2304 x 169	10 x 22	4158	15.83
384 x 3456 x 169	10 x 22	6228	23.69
256 x 3456 x 169	11 x 20	4095	15.58
Total			108.11

Table 5.10: AlexNet optimal SA shape for different convolutional layers given 225 (15 x 15) DSPs. Time is for 100 MHz clock frequency.

M x K x N	**Optimal SA shape**	**Required tiles**	**Time [ms]**
96 x 363 x 3025	14 x 16	156	9.68
256 x 2400 x 729	14 x 16	2752	41.45
384 x 2304 x 169	14 x 16	3960	15.16
384 x 3456 x 169	14 x 16	5928	22.68
256 x 3456 x 169	14 x 16	3952	15.12
Total			104.12

Table 5.9 and 5.10 show the optimal SA shape that results in less execution time given the DSP resources on the development board (220) and 225 respectively, to execute AlexNet convolutional layers. We notice in figure 5.14 that even with slightly higher resources than that available on the development board (225) so that the total DSP/MAC resources $= N^2$ and can be arranged in a square so that $N = SA\_Rows = SA\_Columns$, the squared SA is not optimal. Assuming 225 DSP resources to use a squared SA that result in full utilization of DSPs (15x15), the non-squared 14x16 case results in less execution time. However, and as stated previously, the approximately symmetric result tends towards squared shapes and therefore the time difference with 15x15 is negligible but still the 11x20 case is optimal considering less resources. Results for ResNet18, ResNet50 and VGG16 can be found in appendix C.2.

Table 5.11: Total time execution at 100 MHz with SA shape (14x14) versus the optimal found shapes for different nets' convolutional layers in Accelerator v2.

SA shape	**14x14**	**Optimal shape**	**Performance gain**
Net	**Time [ms]**	**Time [ms]**	[%]
AlexNet	122.25	108.11	11.57
ResNet18	283.70	242.17	14.64
ResNet50	587.61	502.57	14.47
VGG16	1690.55	1463.18	13.45

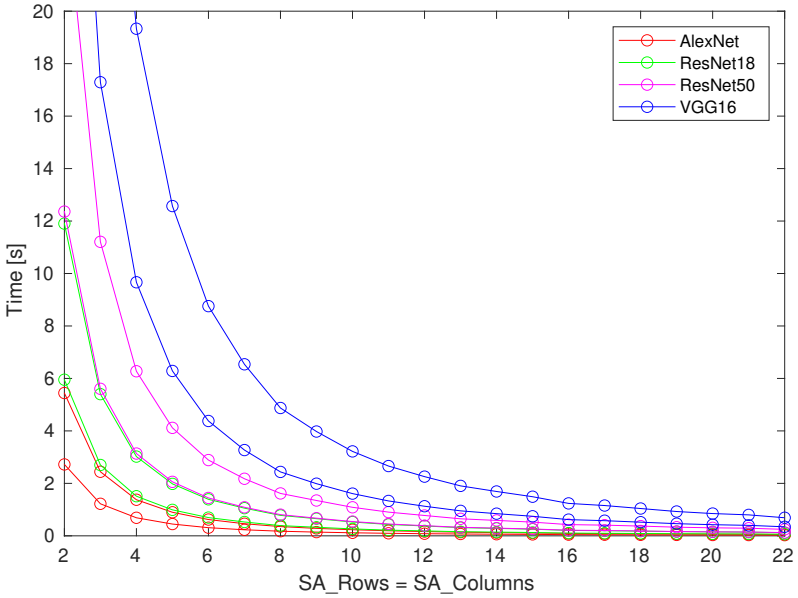


Figure 5.15: Total execution time for all convolutional layers for some common nets using squared SA shapes with system clock frequency of 100 MHz (upper curve) and 200 MHz (lower curve) for each net.

Figure 5.15 shows the execution time at 100 MHz and 200 MHz for squared SA shapes. Similar to the case in the first version, we notice that for different nets there are SA shapes after which increasing the size slowly decreases the execution time. However, the curves are translated and pushed to the left indicating less execution time for the same SA size compared to v1. A comparison of the total execution time at 100 MHz with SA shape (14x14) versus the optimal found shapes for v2 can be found in table 5.11.

Table 5.12: Total time execution at 100 MHz using the optimal found SA shapes in v1 and v2 for different nets' convolutional layers.

SA shape	**v1**	**v2**	**Performance gain**
Net	**Time [ms]**	**Time [ms]**	[%]
AlexNet	256.94	108.11	57.92
ResNet18	551.74	242.17	56.11
ResNet50	1151.30	502.57	56.35
VGG16	3563.93	1463.18	58.94

Table 5.13: Total time execution at 100 MHz using (14x14) SA shape in v1 and v2 for different nets' convolutional layers.

SA shape	**v1**	**v2**	**Performance gain**
Net	**Time [ms]**	**Time [ms]**	[%]
AlexNet	355.89	122.25	65.65
ResNet18	796.49	283.70	64.38
ResNet50	1648.85	587.61	64.36
VGG16	4946.28	1690.55	65.82

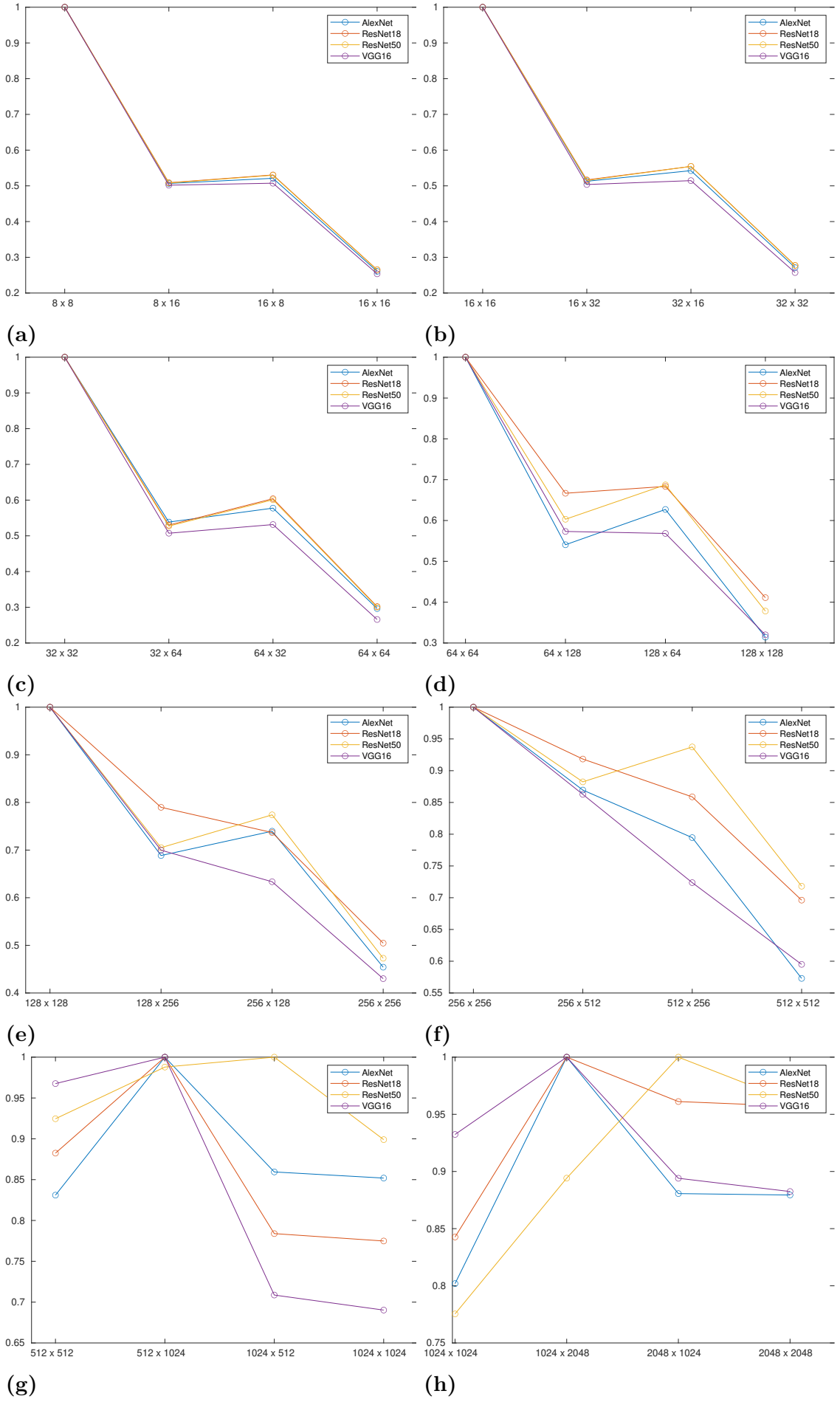


Figure 5.16: Normalized execution cycles for different CNNs across different ranges.

5.5 Comparison between v1 & v2 on hardware

Table 5.14: Measured execution time of Accelerator v1 including DMA transfers at 100 MHz for different SA shapes at `MxKxN` = 40x40x40 using INT8 inputs and INT32 outputs.

Required tiles (external)	**SA shape**	**Measured time inc. DMA [us]**	**Calculated time (*eq.*A.15·*eq.*4.8) (Accelerator only) [us]**	**Difference (DMA transfer+) [us] (%)**
8000	2 x 2	1045.25	960.00	85.25 (8.16)
1000	4 x 4	256.99	240.00	16.99 (6.61)
343	6 x 6	136.01	123.48	12.53 (9.21)
125	8 x 8	66.8	60.00	6.8 (10.20)
27	14 x 14	38.57	22.68	15.89 (41.20)
16	10 x 22	62.75	17.28	45.47 (72.46)
16	11 x 20	57.56	16.32	41.24 (71.65)

Table 5.15: Measured execution time of Accelerator v2 including DMA transfers at 100 MHz for different SA shapes at `MxKxN` = 40x40x40 using INT8 inputs and INT32 outputs.

Required tiles (internal)	**SA shape**	**Measured time inc. DMA [us]**	**Calculated time (*eq.*4.9) (Accelerator only) [us]**	**Difference (DMA transfer+) [us] (%)**
400	2 x 2	358.20	352.00	6.20 (1.73)
100	4 x 4	106.39	96.00	10.39 (9.77)
49	6 x 6	52.78	50.82	1.96 (3.71)
25	8 x 8	41.04	28.00	13.04 (31.77)
9	14 x 14	18.27	12.18	6.09 (33.33)
8	10 x 22	20.97	10.56	10.23 (48.78)
8	11 x 20	27.79	10.56	17.23 (62.00)

Table 5.14 and 5.15 show the execution time for a test case layer with MxKxN = 40x40x40 measured on hardware along with the corresponding calculated one using the extracted equations from the design for system clock frequency of 100 MHz and 200 MHz respectively. The result for squared SA shapes is plotted in figure 5.17 where the upper curve and lower curve of each color represents the measured and the calculated execution time respectively. We notice that the calculated values follow the measured results and that increasing the number of tiles gives higher percentage of the total execution time being at the expense of the accelerator. This indicates better use case than executing one tile where the execution of the accelerator in the

range of tenths of microseconds and the DMA transfers in microseconds. As can be seen in the case of 2x2 in table 5.14 and 5.15 where the number of tiles is the highest in each table, the accelerator time percentage is over 90%. In table 5.16, where the results in milliseconds using 10,000 tiles, the calculated and the measured accelerator time at this precision represents 100% of the total time.

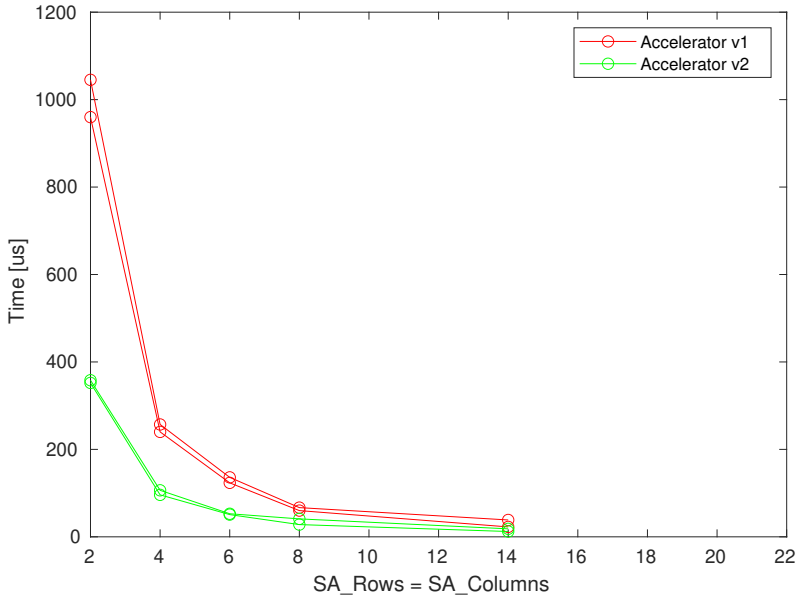


Figure 5.17: Execution time for a test case layer with `MxKxN` = 40x40x40 using squared SA shapes at system clock frequency of 100 MHz using INT8 inputs and INT32 outputs for Accelerator v1 & v2. Upper curve of each color represents the measured value on hardware while the lower curve is the calculated value using previously extracted equations.

Table 5.16: Measured execution time of Accelerator v2 including DMA transfers at 100 MHz for 2x2 SA at `MxKxN` = 200x200x200.

Required tiles (internal)	**SA shape**	**Measured time inc. DMA [ms]**	**Calculated time (Accelerator only) [ms]**	**Difference (DMA transfer+) [ms] (%)**
10000	2 x 2	40.80	40.80	0.00 (00.00)

5.6 Summary of Results

The proposed design in v1 showed that increasing the number of `SA_Columns` decreases the execution time faster than that when increasing the number of `SA_Rows`. This is due to the number of the required tiles, see eq. A.15. In v2, this asymmetry is solved by packing the `N` dimension of the `MxKxN` arrangement inside the design so that `SA_Rows` has the same inverse proportionality as `SA_Columns` to the number of tiles, see eq. A.16. However, when considering larger SA shapes, this behavior

is different between different CNNs and at different ranges where sometime increasing `SA_Rows` improves the performance more than in the case of `SA_Columns`, see figure 5.11 and 5.16.

Given 220 DSPs, the optimal SA shapes for different layers in the studied CNNs were found to be (10x22) or (11x20) in (84-100)% of the cases in both versions of the accelerator. Performance gain of executing the optimal SA shapes versus the largest possible squared shape is around (28-31)% in v1 and (12-15)% in v2. The performance gain in v2 compared to v1 using the optimal found SA shapes is (56-59)% and using the squared shape is (64-66)%.

6

Conclusion

The thesis work presented the development of two versions of a matrix multiplier. Both are parameterizable at the Hardware Description Language (HDL) code with the ability to construct non-squared as well as squared Systolic Arrays (SAs). The work showed the effect of different SA shapes and sizes on the execution time. It showed that non-squared SAs result in higher performance compared to the equivalent largest squared SA that can be synthesized given a specific number of MAC resources. Further, non-squared SAs allow higher rates of utilization of the available resources. The work also showed that increasing SA- square's side value beyond a specific value does not provide significant performance improvements and that this point is different for different Convolutional Neural Networks (CNNs). This allows for wise resource utilization and better estimation of the expected performance gain when increasing the used resources. Finally, different CNNs has different performance improvements as a result of increasing the number of rows or columns of the SA and the function curve of the execution time oscillates at higher SA sizes.

6.1 Future work

This work is a good candidate for partial reconfiguration. Instead of using one single configuration to calculate different matrix sizes and shapes which increases the required tiles, reconfiguring part of the FPGA resources into the shape and size that adapt to the matrices' parameters provide better utilization and should increase performance. This reconfiguration can be done during the execution of another configured part of the FPGA adding extra performance. Further, this work can be a fundamental building block for synthesizing a stack of accelerators of small sizes, for example, of the 2D size of a CNN filter, to further analyze the performance outcomes when directly calculating the Toeplitz Matrix skipping the `Image to Column (im2col)` approach.

Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[2] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A Dynamically Configurable Coprocessor for Convolutional Neural Networks," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 247–257, 2010, ISSN: 0163-5964. DOI: 10.1145/1816038.1815993. [Online]. Available: https://doi.org/10.1145/1816038.1815993.

[3] A. Vasudevan, A. Anderson, and D. Gregg, *Parallel Multi Channel Convolution using General Matrix Multiplication*, 2017. arXiv: 1704.04428 [cs.CV].

[4] R. Xu, S. Ma, Y. Wang, Y. Guo, D. Li, and Y. Qiao, "Heterogeneous Systolic Array Architecture for Compact CNNs Hardware Accelerators," *IEEE Trans. on Parallel and Distributed Syst.*, vol. 33, no. 11, pp. 2860–2871, 2022. DOI: 10.1109/TPDS.2021.3129647.

[5] A. F. Peterson, S. L. Ray, and R. Mittra, "Algorithms for the Solution of Linear Systems of Equations," in *Computational Methods for Electromagnetics*. 1998, pp. 143–186. DOI: 10.1109/9780470544303.ch4.

[6] H. Genc, S. Kim, A. Amid, *et al.*, "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration," in *2021 58th ACM/IEEE Des. Automation Conf. (DAC)*, 2021, pp. 769–774. DOI: 10.1109/DAC18074.2021.9586216.

[7] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *2016 ACM/IEEE 43rd Annu. Int. Symp. on Comput. Architecture (ISCA)*, 2016, pp. 367–379. DOI: 10.1109/ISCA.2016.40.

[8] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient Processing of Deep Neural Networks*. Springer International Publishing, 2020. DOI: 10.1007/978-3-031-01766-7. [Online]. Available: https://doi.org/10.1007%2F978-3-031-01766-7.

[9] H. T. Kung, "Why systolic architectures?" *Comput.*, vol. 15, no. 1, pp. 37–46, 1982. DOI: 10.1109/MC.1982.1653825.

[10] M. Annaratone, E. Arnould, T. Gross, *et al.*, "The Warp Computer: Architecture, Implementation, and Performance," *IEEE Trans. on Comput.*, vol. C-36, no. 12, pp. 1523–1538, 1987. DOI: 10.1109/TC.1987.5009502.

[11] R. Hughey and D. Lopresti, "Architecture of a programmable systolic array," in *[1988] Proc.. Int. Conf. on Systolic Arrays*, 1988, pp. 41–49. DOI: 10.1109/ARRAYS.1988.18043.

[12] R. Morley and T. Sullivan, "A massively parallel systolic array processor system," in *[1988] Proc.. Int. Conf. on Systolic Arrays*, 1988, pp. 217–225. DOI: 10.1109/ARRAYS.1988.18062.

[13] Lopresti, "P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences," *Comput.*, vol. 20, no. 12, pp. 98–99, 1987. DOI: 10.1109/MC.1987.1663629.

[14] A. Dua, Y. Li, and F. Ren, "Systolic-CNN: An OpenCL-defined Scalable Run-time-flexible FPGA Accelerator Architecture for Accelerating Convolutional Neural Network Inference in Cloud/Edge Computing," in *2020 IEEE 28th Annu. Int. Symp. on Field-Programmable Custom Computing Mach.s (FCCM)*, 2020, pp. 231–231. DOI: 10.1109/FCCM48280.2020.00064.

[15] P. Lahari, S. S. Yellampalli, and R. Vaddi, "Systolic Array based Multiply Accumulation Unit for IoT Edge Accelerators," in *2021 IEEE Int. Symp. on Smart Electron. Syst. (iSES)*, 2021, pp. 220–223. DOI: 10.1109/iSES52644.2021.00058.

[16] N. P. Jouppi, C. Young, N. Patil, *et al.*, *In-Datacenter Performance Analysis of a Tensor Processing Unit*, 2017. DOI: 10.48550/ARXIV.1704.04760. [Online]. Available: https://arxiv.org/abs/1704.04760.

[17] A. Samajdar, Y. Zhu, P. N. Whatmough, M. Mattina, and T. Krishna, "SCALE-Sim: Systolic CNN Accelerator," *CoRR*, vol. abs/1811.02883, 2018. arXiv: 1811.02883. [Online]. Available: http://arxiv.org/abs/1811.02883.

[18] P.-H. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun, and E. Culurciello, "NeuFlow: Dataflow vision processing system-on-a-chip," in *2012 IEEE 55th Int. Midwest Symp. on Circuits and Syst. (MWSCAS)*, 2012, pp. 1044–1047. DOI: 10.1109/MWSCAS.2012.6292202.

[19] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks," in *2014 IEEE Conf. on Comput. Vision and Pattern Recognition Workshops*, 2014, pp. 696–701. DOI: 10.1109/CVPRW.2014.106.

[20] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A Convolutional Network Accelerator," in *Proc. of the 25th Edition on Great Lakes Symp. on VLSI*, ser. GLSVLSI '15, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2015, pp. 199–204, ISBN: 9781450334747. DOI: 10.1145/2742060.2743766. [Online]. Available: https://doi.org/10.1145/2742060.2743766.

[21] Z. Du, R. Fasthuber, T. Chen, *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *2015 ACM/IEEE 42nd Annu. Int. Symp. on Comput. Architecture (ISCA)*, 2015, pp. 92–104. DOI: 10.1145/2749469.2750389.

[22] Y. Tortorella, L. Bertaccini, D. Rossi, L. Benini, and F. Conti, "RedMulE: A Compact FP16 Matrix-Multiplication Accelerator for Adaptive Deep Learning on RISC-V-Based Ultra-Low-Power SoCs," in *2022 Des., Automation & Test in Europe Conf. & Exhibition (DATE)*, 2022, pp. 1099–1102. DOI: 10.23919/DATE54114.2022.9774759.

[23] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, *Deep Learning with Limited Numerical Precision*, 2015. arXiv: 1502.02551 [cs.LG].

[24] N. P. Jouppi, G. Kurian, S. Li, *et al.*, *TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings*, 2023. arXiv: 2304.01433 [cs.AR].

[25] F. Shi, H. Li, Y. Gao, B. Kuschner, and S.-C. Zhu, "Sparse Winograd Convolutional Neural Networks on Small-scale Systolic Arrays," *Proc. of the 2019 ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2018.

[26] S. Winograd, *Arithmetic complexity of computations*. (Regional conference series in applied mathematics: 33). Society for ind. and appl. math., 1980, ISBN: 0898711630. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=cat09075a&AN=clpc.oai.edge.chalmers.folio.ebsco.com.fs00001000.f04e7be4.86be.442e.a361.fa8ab61131c1&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[27] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE J. of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017. DOI: 10.1109/JSSC.2016.2616357.

[28] Y. Zeng, H. Sun, J. Katto, and Y. Fan, "Accelerating Convolutional Neural Network Inference Based on a Reconfigurable Sliced Systolic Array," in *2021 IEEE Int. Symp. on Circuits and Syst. (ISCAS)*, 2021, pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401287.

[29] C. Chen, X. Liu, H. Peng, H. Ding, and C.-J. R. Shi, "Ifpna: A flexible and efficient deep neural network accelerator with a programmable data flow engine in 28nm cmos," in *ESSCIRC 2018 - IEEE 44th Eur. Solid State Circuits Conf. (ESSCIRC)*, 2018, pp. 170–173. DOI: 10.1109/ESSCIRC.2018.8494327.

[30] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA, USA: MIT Press, 2012.

[31] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey," *Proc. of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020. DOI: 10.1109/JPROC.2020.2976475.

[32] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth international workshop on frontiers in handwriting recognition*, Suvisoft, 2006.

[33] S. Chetlur, C. Woolley, P. Vandermersch, *et al.*, *Cudnn: Efficient primitives for deep learning*, 2014. arXiv: 1410.0759 [cs.NE].

[34] J. Gu, Y. Liu, Y. Gao, and M. Zhu, "Opencl caffe: Accelerating and enabling a cross platform machine learning framework," in *Proc. of the 4th Int. Workshop on OpenCL*, ser. IWOCL '16, Vienna, Austria: Association for Computing Machinery, 2016, ISBN: 9781450343381. DOI: 10.1145/2909437.2909443. [Online]. Available: https://doi.org/10.1145/2909437.2909443.

[35] B. Jacob, S. Kligys, B. Chen, *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *2018 IEEE/CVF Conf. on Comput. Vision and Pattern Recognition*, 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286.

[36] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *IEEE J. on*

Emerging and Select. Topics in Circuits and Syst., vol. 9, no. 2, pp. 292–308, 2019. DOI: 10.1109/JETCAS.2019.2910232.

[37] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proc. of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017. DOI: 10.1109/JPROC.2017.2761740.

[38] Y.-H. Chen, J. Emer, and V. Sze, "Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 12–21, 2017. DOI: 10.1109/MM.2017.54.

[39] Xilinx Inc., *System Integrated Logic Analyzer v1.1 LogiCORE IP Product Guide (PG261)*, English, version v1.1, February 4, 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg261-system-ila.

[40] Xilinx Inc., *Zynq-7000 SoC Data Sheet (DS190)*, English, version v1.11.1, July 2, 2018. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview.

[41] Xilinx Inc., *7 Series DSP48E1 Slice User Guide (UG479)*, English, version v1.10, March 27, 2018. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1.

[42] Xilinx Inc., *Xilinx 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide for HDL Designs (UG768)*, English, version v14.7, October 2, 2013. [Online]. Available: https://www.xilinx.com/htmldocs/xilinx14_7/7series_hdl.pdf.

[43] ARM, *AMBA 4 AXI4-Stream Protocol Specification*, English, version 1.0, 2010. [Online]. Available: https://zipcpu.com/doc/axi-stream.pdf.

[44] Xilinx Inc., *Vivado Design Suite User Guide: Creating and Packaging Custom IP (UG1118)*, English, version v2022.2, November 2, 2022. [Online]. Available: https://docs.xilinx.com/r/2022.2-English/ug1118-vivado-creating-packaging-custom-ip/Creating-and-Packaging-Custom-IP.

[45] Xilinx Inc., *AXI DMA v7.1 LogiCORE IP Product Guide (PG021)*, English, version v7.1, April 27, 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/pg021_axi_dma.

[46] Xilinx Inc., *Vivado Design Suite User Guide Programming and Debugging (UG908)*, English, version v2022.2, October 19, 2022. [Online]. Available: https://docs.xilinx.com/r/2022.2-English/ug908-vivado-programming-debugging/Introduction.

[47] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proc. of the 25th Int. Conf. on Neural Inform. Process. Syst. - Volume 1*, ser. NIPS'12, Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

A

Appendix 1 - Design parameters

The equations in this chapter are design-specific and were used in the VHDL code. These equations can be used to help choosing customization parameters as described in section 5.3.

Table A.1: Parameters of the `ACCELERATOR v1` and `v2` with the minimum required for one execution.

Parameter	`ACCELERATOR v1`	`ACCELERATOR v2`	**Description**
SA_Rows	user specific		# rows in the systolic array.
SA_Columns	user specific		# columns in the systolic array.
IN_DATA_WIDTH	user specific/platform dependent		Data width of elements in both input matrices.
OUT_DATA_WIDTH	user specific/platform dependent		Data width of elements in SA output.
s_axis_data_width	Eq. A.2		slave axis interface input data width.
m_axis_data_width	Eq. A.1		master axis interface output data width.
M	-	user specific	# rows in the `stationery` matrix.
K	-	user specific	# columns in the `stationery` matrix = # rows in the `moving` matrix.
N	-	user specific	# columns in the `moving` matrix.
weight_RAM_DEPTH	Eq. A.4	Eq. A.10	RAM depth of `stationery` data.
weight_RAM_addr_width	Eq. A.3	Eq. A.9	Address width of weight_RAM.
ifmap_RAM_DEPTH	Eq. A.6	Eq. A.12	RAM depth of `moving` data.
ifmap_RAM_addr_width	Eq. A.5	Eq. A.11	Address width of ifmap_RAM.
psum_RAM_DEPTH	Eq. A.8	Eq. A.14	RAM depth of partial sums.
psum_RAM_addr_width	Eq. A.7	Eq. A.13	Address width of psum_RAM.
TILES	-	Eq. A.16	# tiles of the size of the SA that is required to calculate corresponding M,K values. (N doesn't increase the number of tiles.)

$$m\_axis\_data\_width = 2^{\lceil \log_2 (SA\_Columns \cdot OUT\_DATA\_WIDTH) \rceil} \tag{A.1}$$

$$s\_axis\_data\_width = 2^{\lceil \log_2 ((SA\_Rows+SA\_Columns)\cdot IN\_DATA\_WIDTH)\rceil} \tag{A.2}$$

$$v1\_weight\_RAM\_addr\_width = \lceil \log_2 (SA\_Rows)\rceil \tag{A.3}$$

$$v1\_weight\_RAM\_DEPTH = 2^{\lceil \log_2 (SA\_Rows)\rceil} \tag{A.4}$$

$$v1\_ifmap\_RAM\_addr\_width = \lceil \log_2 (SA\_Columns)\rceil \tag{A.5}$$

$$v1\_ifmap\_RAM\_DEPTH = 2^{\lceil \log_2 (SA\_Columns)\rceil} \tag{A.6}$$

$$v1\_psum\_RAM\_addr\_width = \lceil \log_2 (SA\_Columns)\rceil \tag{A.7}$$

$$v1\_psum\_RAM\_DEPTH = 2^{\lceil \log_2 (SA\_Columns)\rceil} \tag{A.8}$$

$$v2\_weight\_RAM\_addr\_width = \lceil \log_2 (K \cdot \lceil M/SA\_Columns\rceil)\rceil \tag{A.9}$$

$$v2\_weight\_RAM\_DEPTH = 2^{\lceil \log_2 (K\cdot\lceil M/SA\_Columns)\rceil\rceil} \tag{A.10}$$

$$v2\_ifmap\_RAM\_addr\_width = \lceil \log_2 (N \cdot \lceil K/SA\_Rows\rceil)\rceil \tag{A.11}$$

$$v2\_ifmap\_RAM\_DEPTH = 2^{\lceil \log_2 (N\cdot\lceil K/SA\_Rows\rceil)\rceil} \tag{A.12}$$

$$v2\_psum\_RAM\_addr\_width = \lceil \log_2 (M \cdot \lceil N/SA\_Columns\rceil)\rceil \tag{A.13}$$

$$v2\_psum\_RAM\_DEPTH = 2^{\lceil \log_2 (M\cdot\lceil N/SA\_Columns\rceil)\rceil} \tag{A.14}$$

$$v1\_TILES = \lceil K/SA\_Rows\rceil \cdot \lceil M/SA\_Columns\rceil \cdot \lceil N/SA\_Columns\rceil \tag{A.15}$$

$$v2\_TILES = \lceil K/SA\_Rows\rceil \cdot \lceil M/SA\_Columns\rceil \tag{A.16}$$

B

Appendix 2 - CNN parameters

This chapter contains all convolutional layers' parameters and the corresponding Image to Column (im2col) parameters used in this thesis.

Table B.1: AlexNet [47] Convolutional (CONV) layers.

Input			Filter			Output			S	Pad	im2col matrix		
H_i	W_i	C_i	H_f	W_f	C_f	H_o	W_o	C_o			M	K	N
227	227	3	11	11	3	55	55	96	4	0	96	363	3025
27	27	96	5	5	96	27	27	256	1	2	256	2400	729
13	13	256	3	3	256	13	13	384	1	1	384	2304	169
13	13	384	3	3	384	13	13	384	1	1	384	3456	169
13	13	384	3	3	384	13	13	256	1	1	256	3456	169

Table B.2: ResNet18 CONV layers.

Input			Filter			Output			S	Pad	im2col matrix		
H_i	W_i	C_i	H_f	W_f	C_f	H_o	W_o	C_o			M	K	N
256	256	3	7	7	3	128	128	64	2	3	64	147	16384
64	64	64	3	3	64	64	64	64	1	1	64	576	4096
64	64	64	3	3	64	64	64	64	1	1	64	576	4096
64	64	64	3	3	64	64	64	64	1	1	64	576	4096
64	64	64	3	3	64	64	64	64	1	1	64	576	4096
64	64	64	3	3	64	32	32	128	2	1	128	576	1024
32	32	128	3	3	128	32	32	128	1	1	128	1152	1024
32	32	128	3	3	128	32	32	128	1	1	128	1152	1024
32	32	128	3	3	128	32	32	128	1	1	128	1152	1024
32	32	128	3	3	128	16	16	256	2	1	256	1152	256
16	16	256	3	3	256	16	16	256	1	1	256	2304	256
16	16	256	3	3	256	16	16	256	1	1	256	2304	256
16	16	256	3	3	256	16	16	256	1	1	256	2304	256
16	16	256	3	3	256	8	8	512	2	1	512	2304	64
8	8	512	3	3	512	8	8	512	1	1	512	4608	64
8	8	512	3	3	512	8	8	512	1	1	512	4608	64
8	8	512	3	3	512	8	8	512	1	1	512	4608	64

Table B.3: ResNet50 CONV layers.

Input			Filter			Output			S	Pad	im2col matrix		
	H_i		H_f	W_f	C_f	H_o	W_o	C_o			M	K	N
256	256	3	7	7	3	128	128	64	2	3	64	147	16384
64	64	64	1	1	64	64	64	64	1	0	64	64	4096
64	64	64	3	3	64	64	64	64	1	1	64	576	4096
64	64	64	1	1	64	64	64	256	1	0	256	64	4096
64	64	256	1	1	256	64	64	64	1	0	64	256	4096
64	64	64	3	3	64	64	64	64	1	1	64	576	4096
64	64	64	1	1	64	64	64	256	1	0	256	64	4096
64	64	256	1	1	256	64	64	64	1	0	64	256	4096
64	64	64	3	3	64	64	64	64	1	1	64	576	4096
64	64	64	1	1	64	64	64	256	1	0	256	64	4096
64	64	256	1	1	256	64	64	128	1	0	128	256	4096
64	64	128	3	3	128	32	32	128	2	1	128	1152	1024
32	32	128	1	1	128	32	32	512	1	0	512	128	1024
32	32	512	1	1	512	32	32	128	1	0	128	512	1024
32	32	128	3	3	128	32	32	128	1	1	128	1152	1024
32	32	128	1	1	128	32	32	512	1	0	512	128	1024
32	32	512	1	1	512	32	32	128	1	0	128	512	1024
32	32	128	3	3	128	32	32	128	1	1	128	1152	1024
32	32	128	1	1	128	32	32	512	1	0	512	128	1024
32	32	512	1	1	512	32	32	128	1	0	128	512	1024
32	32	128	3	3	128	32	32	128	1	1	128	1152	1024
32	32	128	1	1	128	32	32	512	1	0	512	128	1024
32	32	512	1	1	512	32	32	256	1	0	256	512	1024
32	32	256	3	3	256	16	16	256	2	1	256	2304	256
16	16	256	1	1	256	16	16	1024	1	0	1024	256	256
16	16	1024	1	1	1024	16	16	256	1	0	256	1024	256
16	16	256	3	3	256	16	16	256	1	1	256	2304	256
16	16	256	1	1	256	16	16	1024	1	0	1024	256	256
16	16	1024	1	1	1024	16	16	256	1	0	256	1024	256
16	16	256	3	3	256	16	16	256	1	1	256	2304	256
16	16	256	1	1	256	16	16	1024	1	0	1024	256	256
16	16	1024	1	1	1024	16	16	256	1	0	256	1024	256
16	16	256	3	3	256	16	16	256	1	1	256	2304	256

Continued on next page

Table B.3: ResNet50 CONV layers. (Continued)

16	16	256	1	1	256	16	16	1024	1	0	1024	256	256
16	16	1024	1	1	1024	16	16	256	1	0	256	1024	256
16	16	256	3	3	256	16	16	256	1	1	256	2304	256
16	16	256	1	1	256	16	16	1024	1	0	1024	256	256
16	16	1024	1	1	1024	16	16	256	1	0	256	1024	256
16	16	256	3	3	256	16	16	256	1	1	256	2304	256
16	16	256	1	1	256	16	16	1024	1	0	1024	256	256
16	16	1024	1	1	1024	16	16	512	1	0	512	1024	256
16	16	512	3	3	512	8	8	512	2	1	512	4608	64
8	8	512	1	1	512	8	8	2048	1	0	2048	512	64
8	8	2048	1	1	2048	8	8	512	1	0	512	2048	64
8	8	512	3	3	512	8	8	512	1	1	512	4608	64
8	8	512	1	1	512	8	8	2048	1	0	2048	512	64
8	8	2048	1	1	2048	8	8	512	1	0	512	2048	64
8	8	512	3	3	512	8	8	512	1	1	512	4608	64
8	8	512	1	1	512	8	8	2048	1	0	2048	512	64

Table B.4: VGG16 CONV layers.

Input			Filter			Output			S	Pad	im2col matrix		
H_i	W_i	C_i	H_f	W_f	C_f	H_o	W_o	C_o			M	K	N
224	224	3	3	3	3	224	224	64	1	1	64	27	50176
224	224	64	3	3	64	224	224	64	1	1	64	576	50176
112	112	64	3	3	64	112	112	128	1	1	128	576	12544
112	112	128	3	3	128	112	112	128	1	1	128	1152	12544
56	56	128	3	3	128	56	56	256	1	1	256	1152	3136
56	56	256	3	3	256	56	56	256	1	1	256	2304	3136
56	56	256	3	3	256	56	56	256	1	1	256	2304	3136
28	28	256	3	3	256	28	28	512	1	1	512	2304	784
28	28	512	3	3	512	28	28	512	1	1	512	4608	784
28	28	512	3	3	512	28	28	512	1	1	512	4608	784
14	14	512	3	3	512	14	14	512	1	1	512	4608	196
14	14	512	3	3	512	14	14	512	1	1	512	4608	196
14	14	512	3	3	512	14	14	512	1	1	512	4608	196

C

Appendix 3 - Timing Results

This chapter contains timing results obtained for ResNet18, ResNet50 and VGG16 similar to what was represented earlier for AlexNet in the result chapter.

C.1 Accelerator v1

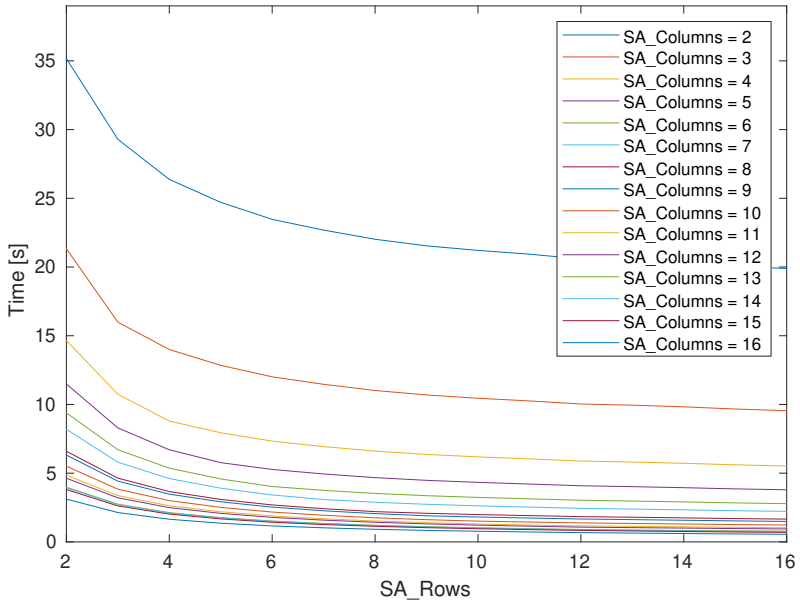


Figure C.1: ResNet18 total execution time at 100 MHz clock frequency with constant `SA_Columns` and variable `SA_Rows` in Accelerator v1.

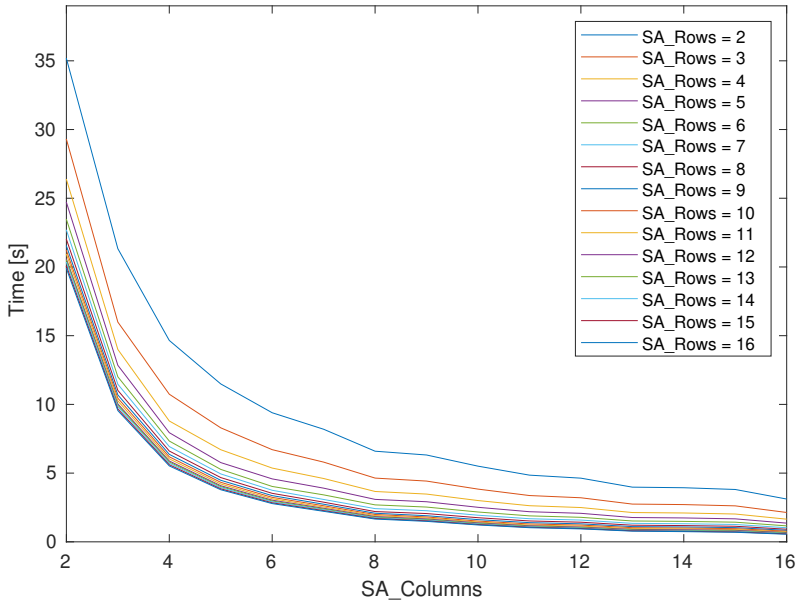


Figure C.2: ResNet18 total execution time at 100 MHz clock frequency with constant `SA_Rows` and variable `SA_Columns` in Accelerator v1.

Table C.1: ResNet18 optimal SA shape for different convolutional layers given 220 DSPs

M x K x N	**Optimal SA shape**	**Required tiles**	**Time [ms]**
64 x 147 x 16384	10 x 22	33525	36.20
64 x 576 x 4096	10 x 22	32538	35.14
64 x 576 x 4096	10 x 22	32538	35.14
64 x 576 x 4096	10 x 22	32538	35.14
64 x 576 x 4096	10 x 22	32538	35.14
128 x 576 x 1024	10 x 22	16356	17.66
128 x 1152 x 1024	10 x 22	32712	35.32
128 x 1152 x 1024	10 x 22	32712	35.32
128 x 1152 x 1024	10 x 22	32712	35.32
256 x 1152 x 256	10 x 22	16704	18.04
256 x 2304 x 256	10 x 22	33264	35.92
256 x 2304 x 256	10 x 22	33264	35.92
256 x 2304 x 256	10 x 22	33264	35.92
512 x 2304 x 64	10 x 22	16632	17.96
512 x 4608 x 64	10 x 22	33192	35.84
512 x 4608 x 64	10 x 22	33192	35.84
512 x 4608 x 64	10 x 22	33192	35.84
Total			551.74

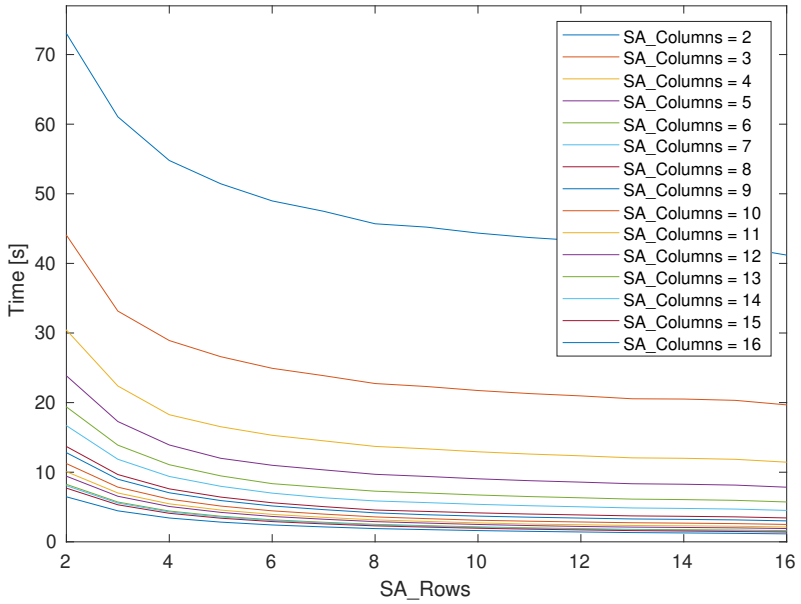


Figure C.3: ResNet50 total execution time at 100 MHz clock frequency with constant `SA_Columns` and variable `SA_Rows` in Accelerator v1.

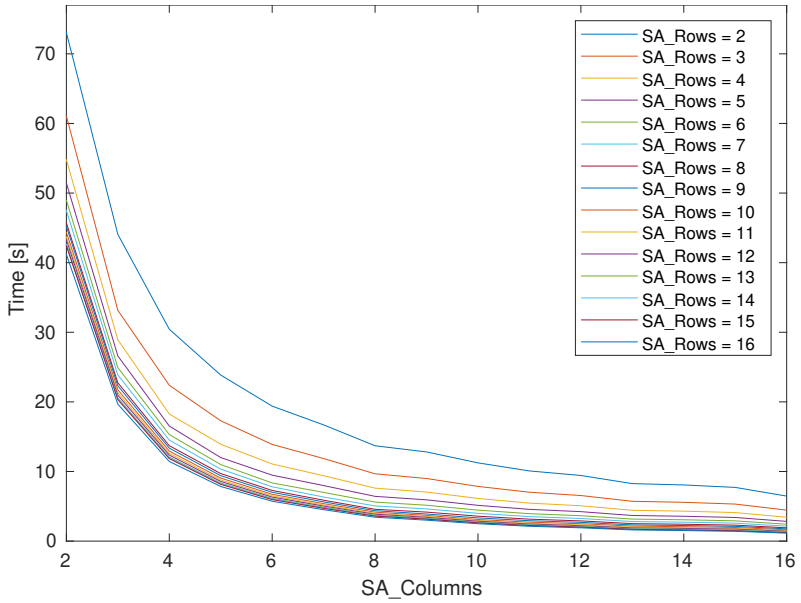


Figure C.4: ResNet50 total execution time at 100 MHz clock frequency with constant `SA_Rows` and variable `SA_Columns` in Accelerator v1.

Table C.2: ResNet50 optimal SA shape for different convolutional layers given 220 DSPs

M x K x N	**Optimal SA shape**	**Required tiles**	**Time [ms]**
64 x 147 x 16384	10 x 22	33525	36.20

Continued on next page

Table C.2: ResNet50 optimal SA shape for different convolutional layers given 220 DSPs (Continued)

64 x 64 x 4096	10 x 22	3927	4.24
64 x 576 x 4096	10 x 22	32538	35.14
256 x 64 x 4096	11 x 20	15990	16.30
64 x 256 x 4096	10 x 22	14586	15.75
64 x 576 x 4096	10 x 22	32538	35.14
256 x 64 x 4096	11 x 20	15990	16.30
64 x 256 x 4096	10 x 22	14586	15.75
64 x 576 x 4096	10 x 22	32538	35.14
256 x 64 x 4096	11 x 20	15990	16.30
128 x 256 x 4096	10 x 22	29172	31.50
128 x 1152 x 1024	10 x 22	32712	35.32
512 x 128 x 1024	10 x 22	14664	15.83
128 x 512 x 1024	10 x 22	14664	15.83
128 x 1152 x 1024	10 x 22	32712	35.32
512 x 128 x 1024	10 x 22	14664	15.83
128 x 512 x 1024	10 x 22	14664	15.83
128 x 1152 x 1024	10 x 22	32712	35.32
512 x 128 x 1024	10 x 22	14664	15.83
128 x 512 x 1024	10 x 22	14664	15.83
128 x 1152 x 1024	10 x 22	32712	35.32
512 x 128 x 1024	10 x 22	14664	15.83
256 x 512 x 1024	10 x 22	29328	31.67
256 x 2304 x 256	10 x 22	33264	35.92
1024 x 256 x 256	10 x 22	14664	15.83
256 x 1024 x 256	10 x 22	14832	16.01
256 x 2304 x 256	10 x 22	33264	35.92
1024 x 256 x 256	10 x 22	14664	15.83
256 x 1024 x 256	10 x 22	14832	16.01
256 x 2304 x 256	10 x 22	33264	35.92
1024 x 256 x 256	10 x 22	14664	15.83
256 x 1024 x 256	10 x 22	14832	16.01
256 x 2304 x 256	10 x 22	33264	35.92
1024 x 256 x 256	10 x 22	14664	15.83
256 x 1024 x 256	10 x 22	14832	16.01

Continued on next page

Table C.2: ResNet50 optimal SA shape for different convolutional layers given 220 DSPs (Continued)

256 x 2304 x 256	10 x 22	33264	35.92
1024 x 256 x 256	10 x 22	14664	15.83
256 x 1024 x 256	10 x 22	14832	16.01
256 x 2304 x 256	10 x 22	33264	35.92
1024 x 256 x 256	10 x 22	14664	15.83
512 x 1024 x 256	10 x 22	29664	32.03
512 x 4608 x 64	10 x 22	33192	35.84
2048 x 512 x 64	10 x 22	14664	15.83
512 x 2048 x 64	10 x 22	14760	15.94
512 x 4608 x 64	10 x 22	33192	35.84
2048 x 512 x 64	10 x 22	14664	15.83
512 x 2048 x 64	10 x 22	14760	15.94
512 x 4608 x 64	10 x 22	33192	35.84
2048 x 512 x 64	10 x 22	14664	15.83
Total			1151.30

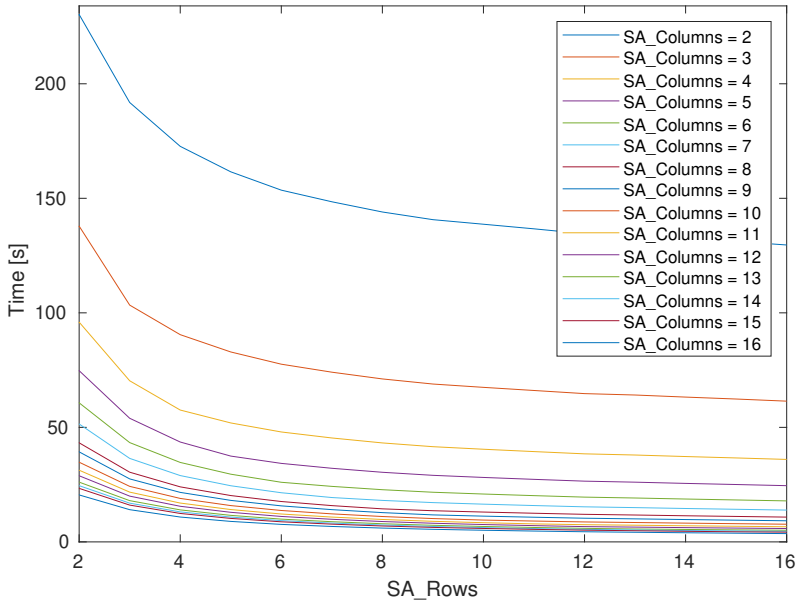


Figure C.5: VGG16 total execution time at 100 MHz clock frequency with constant `SA_Columns` and variable `SA_Rows` in Accelerator v1.

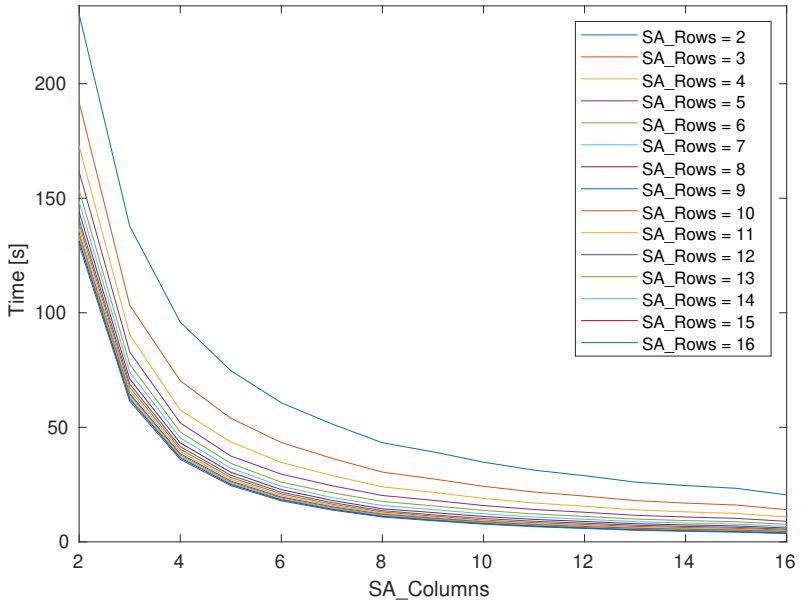


Figure C.6: VGG16 total execution time at 100 MHz clock frequency with constant `SA_Rows` and variable `SA_Columns` in Accelerator v1.

Table C.3: VGG16 optimal SA shape for different convolutional layers given 220 DSPs

M x K x N	**Optimal SA shape**	**Required tiles**	**Time [ms]**
64 x 27 x 50176	9 x 22	20529	21.76
64 x 576 x 50176	10 x 22	396894	428.64
128 x 576 x 12544	10 x 22	198708	214.60
128 x 1152 x 12544	10 x 22	397416	429.20
256 x 1152 x 3136	10 x 22	199056	214.98
256 x 2304 x 3136	10 x 22	396396	428.10
256 x 2304 x 3136	10 x 22	396396	428.10
512 x 2304 x 784	10 x 22	199584	215.55
512 x 4608 x 784	10 x 22	398304	430.16
512 x 4608 x 784	10 x 22	398304	430.16
512 x 4608 x 196	10 x 22	99576	107.54
512 x 4608 x 196	10 x 22	99576	107.54
512 x 4608 x 196	10 x 22	99576	107.54
Total			3563.93

C.2 Accelerator v2

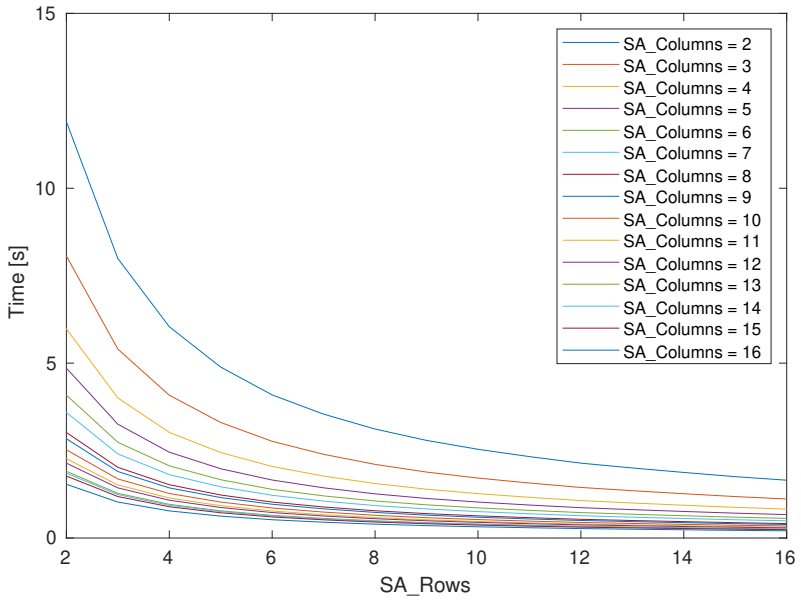


Figure C.7: ResNet18 total execution time at 100 MHz clock frequency with constant `SA_Columns` and variable `SA_Rows` in Accelerator v2.

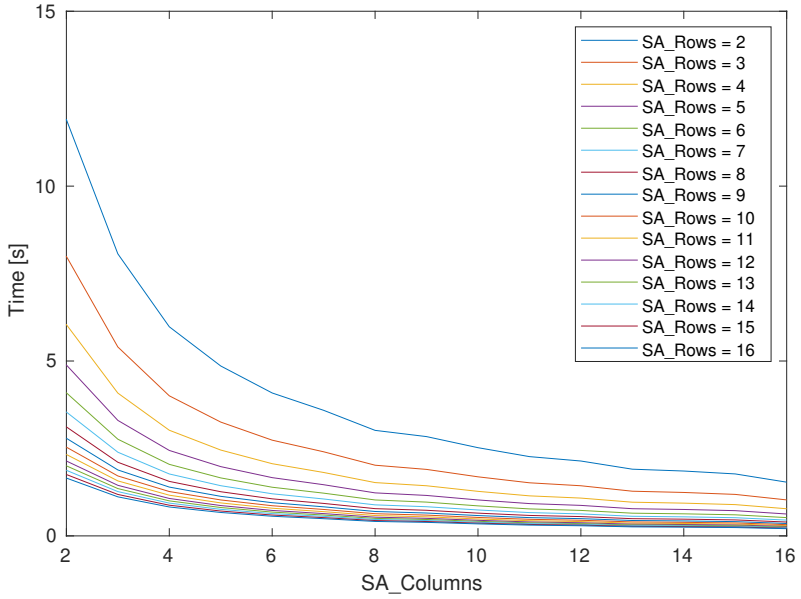


Figure C.8: ResNet18 total execution time at 100 MHz clock frequency with constant `SA_Rows` and variable `SA_Columns` in Accelerator v2.

Table C.4: ResNet18 optimal SA shape for different convolutional layers given 220 DSPs

M x K x N	Optimal SA shape	Required tiles	Time [ms]
64 x 147 x 16384	10 x 22	45	15.24
64 x 576 x 4096	10 x 22	174	14.44
64 x 576 x 4096	10 x 22	174	14.44
64 x 576 x 4096	10 x 22	174	14.44
64 x 576 x 4096	10 x 22	174	14.44
128 x 576 x 1024	10 x 22	348	7.33
128 x 1152 x 1024	10 x 22	696	14.60
128 x 1152 x 1024	10 x 22	696	14.60
128 x 1152 x 1024	10 x 22	696	14.60
256 x 1152 x 256	11 x 20	1365	7.59
256 x 2304 x 256	11 x 20	2730	15.15
256 x 2304 x 256	11 x 20	2730	15.15
256 x 2304 x 256	11 x 20	2730	15.15
512 x 2304 x 64	11 x 20	5460	9.30
512 x 4608 x 64	11 x 20	10894	18.54
512 x 4608 x 64	11 x 20	10894	18.54
512 x 4608 x 64	11 x 20	10894	18.54
Total			242.17

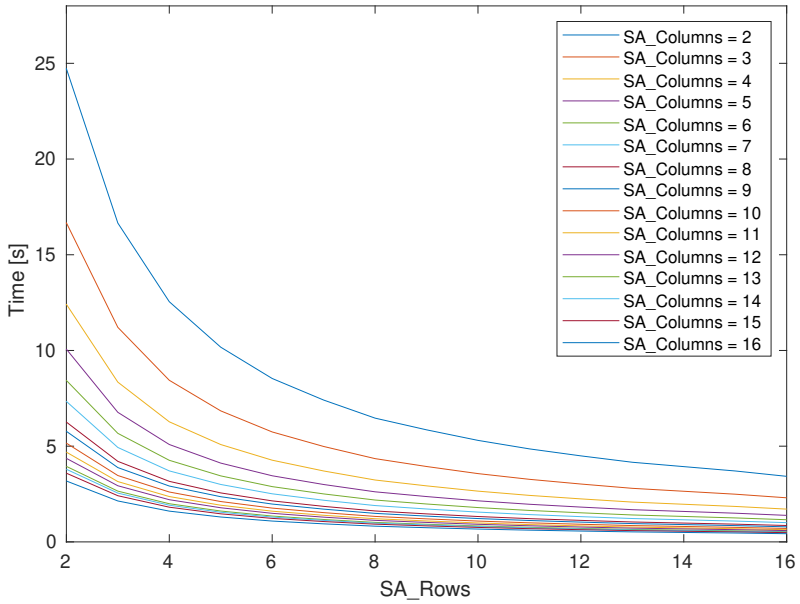


Figure C.9: ResNet50 total execution time at 100 MHz clock frequency with constant `SA_Columns` and variable `SA_Rows` in Accelerator v2.

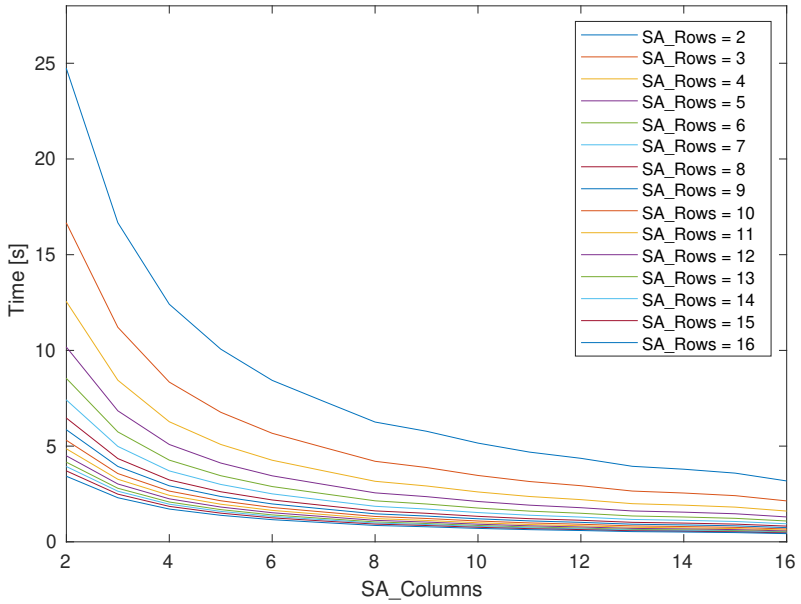


Figure C.10: ResNet50 total execution time at 100 MHz clock frequency with constant `SA_Rows` and variable `SA_Columns` in Accelerator v2.

Table C.5: ResNet50 optimal SA shape for different convolutional layers given 220 DSPs

M x K x N	**Optimal SA shape**	**Required tiles**	**Time [ms]**
64 x 147 x 16384	10 x 22	45	15.24
64 x 64 x 4096	13 x 16	20	1.81
64 x 576 x 4096	10 x 22	174	14.44
256 x 64 x 4096	11 x 20	78	6.94
64 x 256 x 4096	10 x 22	78	6.54
64 x 576 x 4096	10 x 22	174	14.44
256 x 64 x 4096	11 x 20	78	6.94
64 x 256 x 4096	10 x 22	78	6.54
64 x 576 x 4096	10 x 22	174	14.44
256 x 64 x 4096	11 x 20	78	6.94
128 x 256 x 4096	10 x 22	156	13.08
128 x 1152 x 1024	10 x 22	696	14.60
512 x 128 x 1024	10 x 22	312	6.76
128 x 512 x 1024	10 x 22	312	6.58
128 x 1152 x 1024	10 x 22	696	14.60
512 x 128 x 1024	10 x 22	312	6.76
128 x 512 x 1024	10 x 22	312	6.58

Continued on next page

Table C.5: ResNet50 optimal SA shape for different convolutional layers given 220 DSPs (Continued)

128 x 1152 x 1024	10 x 22	696	14.60
512 x 128 x 1024	10 x 22	312	6.76
128 x 512 x 1024	10 x 22	312	6.58
128 x 1152 x 1024	10 x 22	696	14.60
512 x 128 x 1024	10 x 22	312	6.76
256 x 512 x 1024	11 x 20	611	12.90
256 x 2304 x 256	11 x 20	2730	15.15
1024 x 256 x 256	10 x 22	1222	6.89
256 x 1024 x 256	11 x 20	1222	6.80
256 x 2304 x 256	11 x 20	2730	15.15
1024 x 256 x 256	10 x 22	1222	6.89
256 x 1024 x 256	11 x 20	1222	6.80
256 x 2304 x 256	11 x 20	2730	15.15
1024 x 256 x 256	10 x 22	1222	6.89
256 x 1024 x 256	11 x 20	1222	6.80
256 x 2304 x 256	11 x 20	2730	15.15
1024 x 256 x 256	10 x 22	1222	6.89
256 x 1024 x 256	11 x 20	1222	6.80
256 x 2304 x 256	11 x 20	2730	15.15
1024 x 256 x 256	10 x 22	1222	6.89
256 x 1024 x 256	11 x 20	1222	6.80
256 x 2304 x 256	11 x 20	2730	15.15
1024 x 256 x 256	10 x 22	1222	6.89
512 x 1024 x 256	11 x 20	2444	13.60
512 x 4608 x 64	11 x 20	10894	18.54
2048 x 512 x 64	11 x 20	4841	8.31
512 x 2048 x 64	11 x 20	4862	8.28
512 x 4608 x 64	11 x 20	10894	18.54
2048 x 512 x 64	11 x 20	4841	8.31
512 x 2048 x 64	11 x 20	4862	8.28
512 x 4608 x 64	11 x 20	10894	18.54
2048 x 512 x 64	11 x 20	4841	8.31
Total			502.57

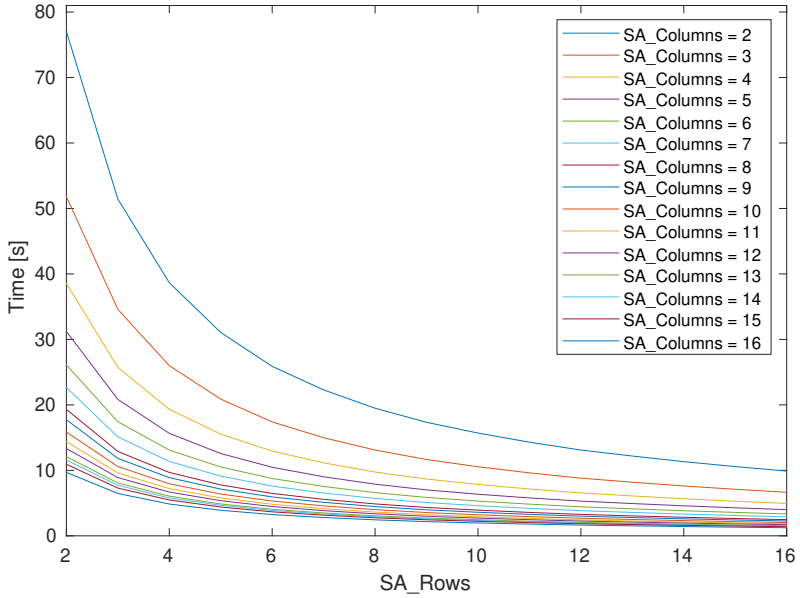


Figure C.11: VGG16 total execution time at 100 MHz clock frequency with constant `SA_Columns` and variable `SA_Rows` in Accelerator v2.

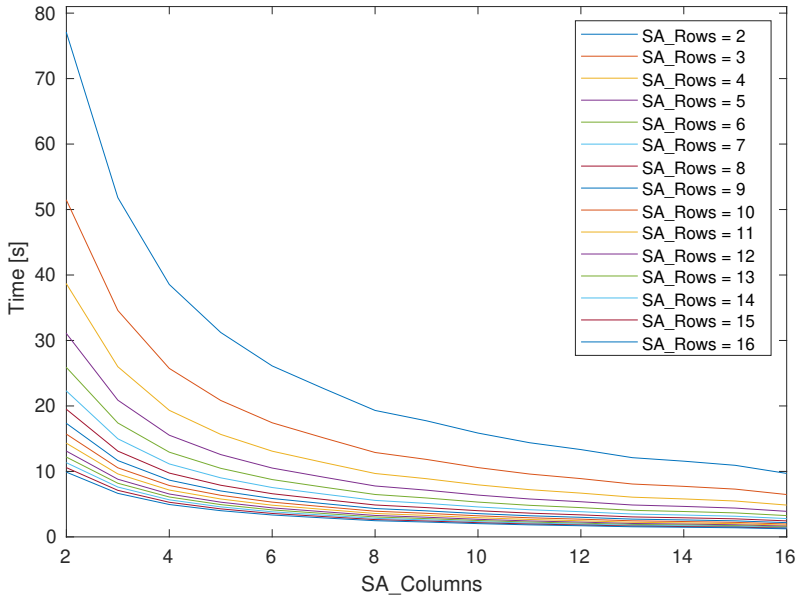


Figure C.12: VGG16 total execution time at 100 MHz clock frequency with constant `SA_Rows` and variable `SA_Columns` in Accelerator v2.

Table C.6: VGG16 optimal SA shape for different convolutional layers given 220 DSPs

M x K x N	**Optimal SA shape**	**Required tiles**	**Time [ms]**
64 x 27 x 50176	9 x 22	9	10.49
64 x 576 x 50176	10 x 22	174	176.14
128 x 576 x 12544	10 x 22	348	88.18
128 x 1152 x 12544	22 x 10	689	174.91
256 x 1152 x 3136	11 x 20	1365	86.58
256 x 2304 x 3136	11 x 20	2730	172.77
256 x 2304 x 3136	11 x 20	2730	172.77
512 x 2304 x 784	11 x 20	5460	88.11
512 x 4608 x 784	11 x 20	10894	175.59
512 x 4608 x 784	11 x 20	10894	175.59
512 x 4608 x 196	11 x 20	10894	47.33
512 x 4608 x 196	11 x 20	10894	47.33
512 x 4608 x 196	11 x 20	10894	47.33
Total			1463.18