Multidataflow Systolic Array

A THESIS

Submitted by

Rohan Kaulgekar

(EE15B097)

under the guidance of

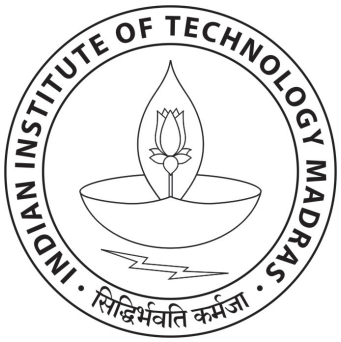
Dr. Pratyush Kumar

for the award of

B.Tech & M.Tech Degree

in

ELECTRICAL ENGINEERING



INDIAN INSTITUTE OF TECHNOLOGY MADRAS

2020

Thesis Certificate

This is to certify that the thesis titled with **Multidataflow Systolic Array** submitted by **Rohan Kaulgekar** to Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology & Master of Technology**, is a bona fide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Pratyush Kumar

Project Guide

Assistant Professor

Computer Science and Engineering
Indian Institute of Technology Madras

Place: Chennai

Date: 22 June, 2020

Acknowledgement

I express my sincere gratitude to my project guide Dr. Pratyush Kumar. for his constant support during the entire period of my Dual Degree Project. His friendly guidance kept me motivated at all times.

I also express my gratitude towards my teammates: Vinod Ganesan, Gokulan Ravi, Mohan Prasath G R and Neel Gala for countless discussions which helped me in solving challenging problems.

Abstract

Deep learning architectures such as Deep Neural Networks have been applied to various fields in the modern world. Convolutional Neural Networks(CNNs) are the dominant DNNs of choice. It is observed that systolic arrays, due to its data reuse capabilities, count as one of the most dominant architectures to perform the convolution operations effectively. This work proposes one such systolic architecture. The project explains the design and operation of each Processing Element(PE) inside the systolic array, and then moves towards analyzing and testing the array as a whole. The project finally analyzes the hardware and mode-switching cost of systolic array.

Contents

List of Figures

Abbreviations

PE: Processing Element

MDF: Multidataflow

SA: Systolic Array

WS: Weight Stationary

OS: Output Stationary

nRow: Number of rows in systolic array

nCol: Number of columns in systolic array

bWidth: Bitwidth of input and weight data elements

twbWidth: Bitwidth of accumulator data elements

rg_coord: 8-bit register storing the Y-coordinate value of PE

rg_stationary: twbWidth-wide register storing the stationary value inside PE

we_data: Input data element inside a PE coming from west FIFO

ns_data: Weight/accumulator data element inside a PE coming from north FIFO

NSData: Data structure storing ns_data, 8-bit counter value and enumerated mode value which travels along north-south direction

MAC: Multiply-and-ACcumulate

TxRx: Custom Bluespec package for transmitting/recieving data to/from a FIFO

Chapter 1

Introduction

1.1 Introduction

Deep learning architectures such as Deep Neural Networks(DNN) have been applied to fields like computer vision, speech recognition, NLP and audio recognition programs, where they have produced results comparable to and even in some cases surpassing human expert performance. A DNN is an Artificial Neural Network with multiple layers between the input and output layers. The DNN finds the correct mathematical manipulation to turn the input into the output, whether it be a linear relationship or a non-linear relationship.

DNNs are typically feedforward networks in which data flows from the input layer to the output layer without looping back. At first, the DNN creates a map of virtual neurons and assigns random numerical values, or "weights", to connections between them. The weights and inputs are multiplied and return an output between 0 and 1. Once such a network is ready, the user feeds in a set of inputs whose outputs are already known. This phase is called training phase, where the outputs computed by the network is compared against actual output. If the network did not accurately

recognize a particular pattern, an algorithm would adjust the weights to steer the network in the correct direction. That way the algorithm can make certain parameters more influential, until it determines the correct mathematical manipulation to fully process the data.

One of the main issues with the DNNs is that it must consider many training parameters, such as the size (number of layers and number of units per layer), the learning rate(the rate of change of weights), and initial weights. Sweeping through the parameter space for optimal parameters is a challenge due to the cost in time and computational resources. Hence, it is paramount that we accelerate this operation to get the maximum performance possible out of a DNN by using suitable hardware accelerators.

The primitive unit of computation is the convolution operation which amounts for more than 90% of the execution time for modern DNNs. In mathematical terms, convolution is basically an element-wise matrix multiplication of an input data matrix and a filter matrix to generate an output feature map. Each output entry is generated by one matrix multiply operation, and the feature matrix is strided across the complete input matrix to generate the output map.

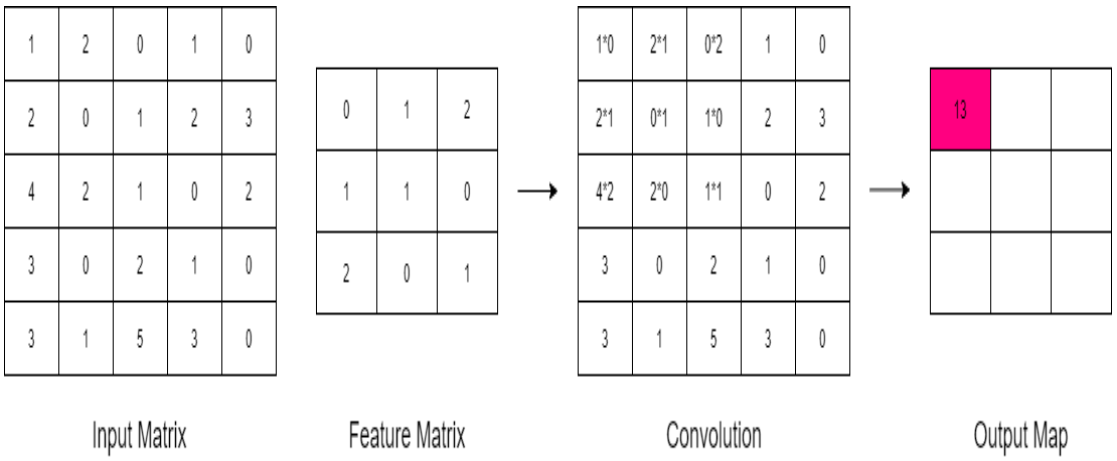


Figure 1.1: Convolution Operation

It can be clearly seen that one output element requires each input element only once. However, adjacent elements reuse these data elements. Conventional CPUs in this case will not perform well as they perform one operation at a time. Thus for bigger dimensions of matrices, the CPU will run out of free cache and will have to fetch the same data elements again. Hence we need a better hardware to tackle this problem.

1.2 Systolic Operation

A systolic array is made of multiple small PEs, capable of performing a MAC operation, connected in a grid-like structure. Elements from one of the maps(which is generally the feature map or weights) are kept stationary inside the PE. Elements from the other map are pushed across in west to east direction. Elements from output feature map(which are generally 0) are then pushed in

north to south direction. During each cycle, the PE multiplies the incoming input element with the weight value stored inside its register and adds it to the partial output element. The updated elements are then sent in east and south directions and gets ready for performing next MAC operation on the next incoming data set. The final output elements are extracted at the end of each column.

1.3 Objective

The objective of this work is to use the idea of systolic dataflow and create and test a systolic array based on it, with support for output-stationary and weight-stationary dataflows. We will then verify the operation of the array by building custom testbenches around it, and analyze the hardware costs for the same. With this, the final objective will be able to analyze the cost of reconfigurability.

1.4 Scope of the thesis

The thesis organization is as follows:

- The thesis first gives a brief overview of the DNN Accelerator, where we plan to use the multidataflow systolic array
- It then explains two dataflows in systolic architecture, namely weight stationary and output-stationary dataflows
- It then proposes a PE design which can be used for both the

dataflows, followed by connecting these elements to form a 2D array

- We then look at how the data needs to be sent across the buffers to carry out convolution operation
- We then discuss the simulation results of the array which was run under a custom testbench and follow it up by comparing the hardware requirements for each of the dataflow
- Finally, the thesis analyses and comments on the current design, its strengths and potential improvements

The main point to note here is that multidataflow array is basically a Matrix Multiply Unit inside the DNN accelerator. All the other blocks like memory modules, buffers, frontend and host-interface modules are not discussed and are out of the scope of this thesis.

Chapter 2

Literature Review

2.1 Introduction

Many recent hardware-based state-of-the-art deep learning accelerators use systolic arrays for efficient implementations of convolutional neural networks (CNNs). They leverage properties of systolic arrays such as parallel processing under the dataflow architecture, regular layout of processing elements, efficient inter-processor communication, and minimized I/O by being able to reuse the same data fetched from the memory many times.

2.2 Gemmini microarchitecture [1]

The microarchitecture of the systolic array is illustrated in Figure 2.1. The basic element of the systolic array is a fully combinational processing element (PE), which performs MACs. The PEs can support weight-stationary and output-stationary dataflows. The PEs can also support different bitwidths for their inputs, outputs, and internal buffer.

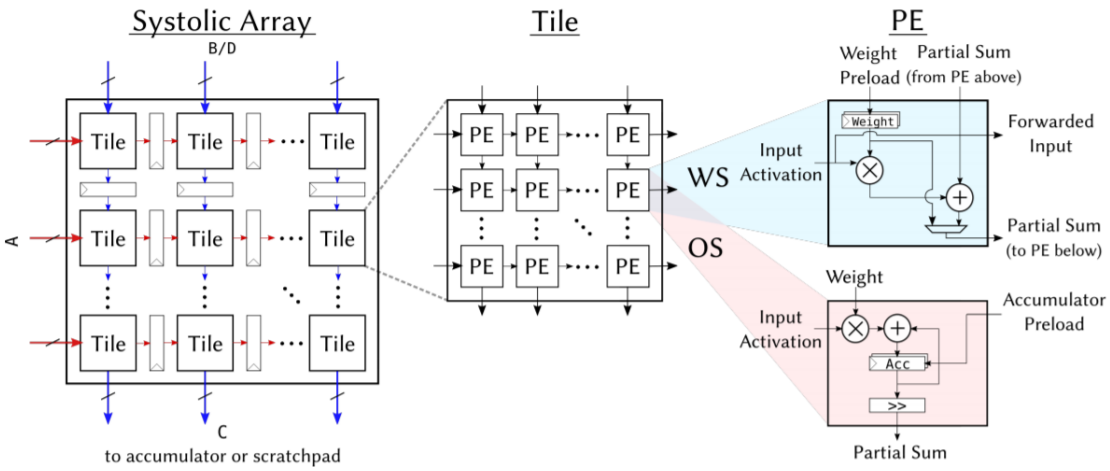


Figure 2.1: Gemmini Microarchitecture, Figure 2 in [1]

The authors concluded that weight-stationary dataflow consumed less power than the output-stationary baseline, as it did not require accumulators in the PEs of the systolic mesh. The combined dataflow consumed more power and occupied larger area compared to both, the output-stationary and weight-stationary dataflows as it needed extra control logic to get the desired results.

2.3 Google TPUs [2]

Tensor Processing Unit(TPU) is a custom ASIC developed by Google Inc. for inference, when they realised that speech recognition DNNs would requirethem to double their datacenters to meet the computation demands, and would be very expensive to satisfy with conventional CPUs. The TPU was designed to be a coprocessor on the PCIe I/O bus, allowing it to plug into existing servers just as a GPU does.

The data is stored in a Unified Buffer(UB) inside the TPU, which interacts with the host interface. The main computational part of TPU is the Matrix Multiply Unit(MMU), which uses weight-stationary systolic execution to save energy by reducing reads and writes from the buffers. Its inputs are the weight FIFO and the UB, and the output is stored inside accumulators. Custom CISC instruction set was developed, out of which the key ones are used to send the data in and out from the UB to host memory. The transfer of weights from UB to Weight FIFO and from Accumulators back to UB is done using similar instructions. Once the peripherals to MMU are ready, a *Convolve* command is given to MMU to begin the MAC operations.

The authors employed a roofline model to compare the performances of their TPU, an NVIDIA K80 GPU and an Intel Haswell CPU. They found out that on Neural Networks such as LSTMs, CNNs and MLPs, the ridge point at 1350 operations per byte of weight memory fetched TPU. This was significantly higher as compared to 13 and 9 operations per byte in the case of Haswell CPU and K80 GPU respectively. Power-wise, the TPU server has 17 to 34 times better total-performance/Watt than Haswell, and 14 to 16 times the performance/Watt of the K80 server.

This was so because inference apps usually emphasize response-time over throughput since they are often user-facing. The time-varying optimizations of CPUs (caches, out-of-order

execution, multithreading, multiprocessing, prefetching) that help average throughput more than guaranteed latency was key reason for CPU's poor performance. Because of the latency limits, the K80 GPU was underutilized for inference, and is just a little faster than a Haswell CPU. Thus, TPUs were the servers-of-choice for inference apps.

2.4 Bit-serial Systolic Arrays [3]

Kung et al. used bit-serial systolic cells . It is a weight-stationary dataflow array where instead of buses, they use 1-bit serial input wires for both, input and accumulator.

Figure 2.2 shows the proposed bit-serial MAC design which is used across all systolic array implementation for 8-bit input Xi and 8-bit filter weight W. The white logic elements implement the bit-serial multiplication between the input Xi and the absolute value of the filter weight. The blue logic elements negate the product based on the sign of the filter weight. The pink full adder performs bit-serial addition between the product and the input accumulation Yi.

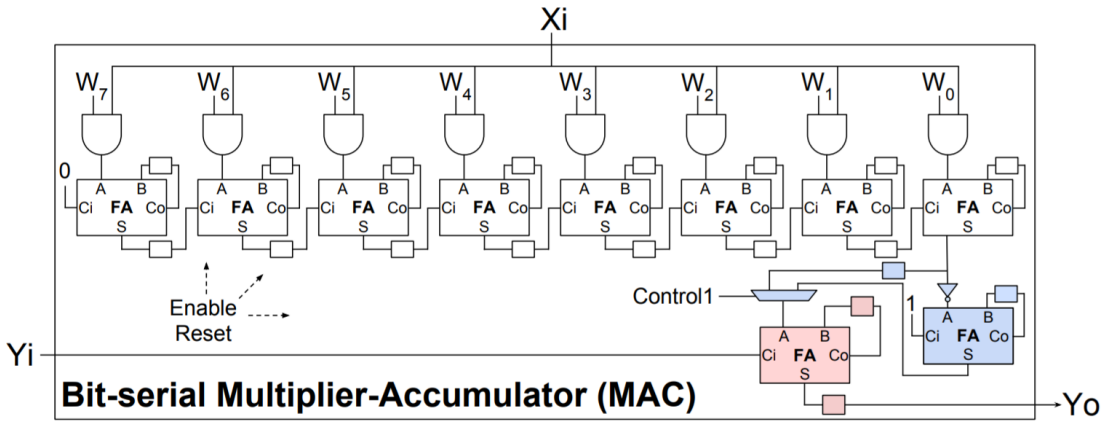


Figure 2.2: Bit-serial PE, Figure 7 in [3]

The balanced version of the array considers 8-bit input and 8-bit accumulator values. Thus, maintaining one clock delay across adjacent inputs/accumulators across columns/rows is sufficient. However in case of unbalanced widths, a delay of M-N clock cycles was required between two subsequent inputs, where M is accumulator width and N is input width. To hide this latency, when M=32 and N=8, they filled in these gaps for each cell by processing four independent input data streams simultaneously in an interleaved manner, while expanding the processing power and accumulation data path by 4x. Figure 2.3 shows the corresponding cell arrays.

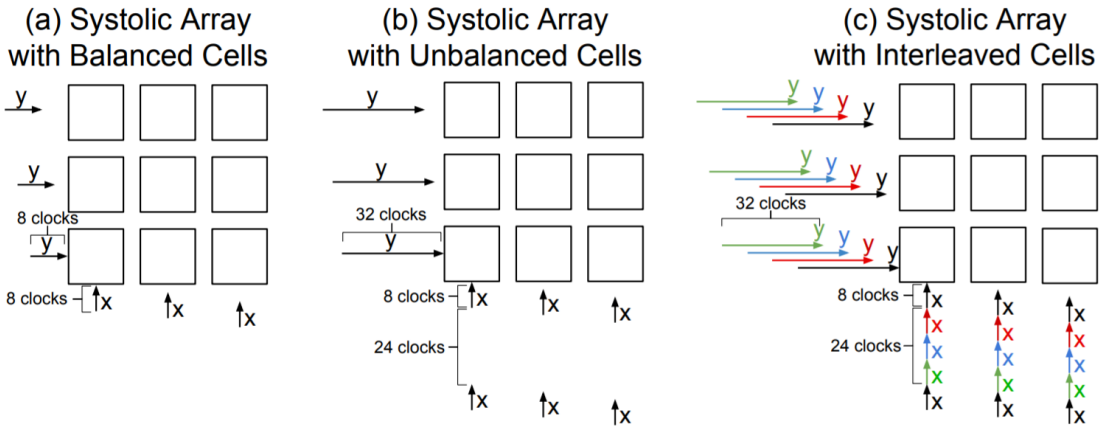


Figure 2.3: Bit-serial Array, Figure 9 in [3]

Since the main objectives of the paper were to introduce a column combining algorithm and joint optimization methodology, it is not very clear that how much of an impact was generated by bit-serial arrays. Similarly, the paper didn't explain the algorithm which the authors used for pushing in weights.

Chapter 3

DNN Accelerator

3.1 Overview

The accelerator itself can be divided into three components, namely the compute module, the control and memory subsystem and the interface module. The compute module consists of the multi-dimensional array of Processing Elements (PEs), built as a systolic array. In addition, the module also contains a Tensor ALU, which performs SIMD operations on inputs.

The control and memory subsystem can be split into frontend module and backend module, handling instructions and data respectively. The frontend consists of instruction queues, to which instructions are fetched from the interface module, and finally dispatched to the compute module. The backend contains buffers for storing input and output feature maps and filters. The frontend and compute module communicate using a set of micro-ops, which store the hyper-parameters of operation to be executed on the grid.

Lastly, the interface module manages the communication with the core (to configure registers) and memory (to send/receive data). The interface module connects the accelerator sub-system to DRAM and other associated storage through AXI interface. In

addition, it also connects the base control processor with the accelerator sub-system. The module sends/receives data to/from buffers in the memory subsystem.

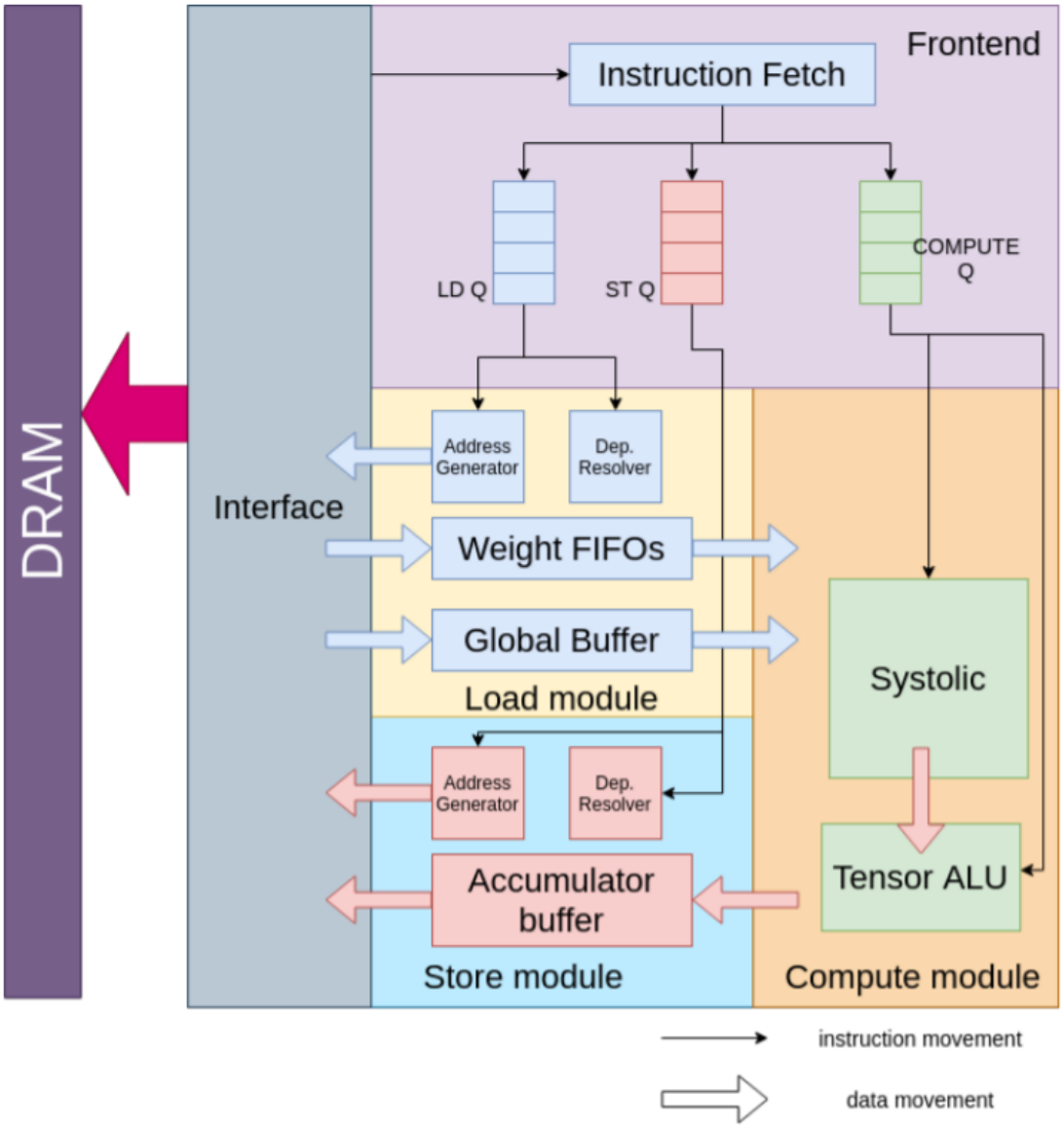


Figure 3.1: DNN Accelerator

3.2 Compute Grid

Processing Element (PE) is the smallest unit of logic, which is replicated to a multi-dimensional grid. Each PE contains registers

to store one value each of input, weight and output, and performs one Multiply-and-ACcumulate operation each cycle using those values to generate one output value. Depending on the way data flows through the compute module, later explained in the Design Section, one data-structure out of inputs, weights and the outputs are kept stationary in the PE and the remaining are streamed across the rows and columns of the systolic grid.
The above mentioned PE is replicated along two dimensions, and interconnected along both the dimensions with immediate PEs using FIFOs to form a systolic array. Three different multi-dimensional vectors (input maps, output maps, weight filters) are involved in the convolution operation. One of them is first populated into the systolic grid, with each PE holding one or few values throughout the entire computation. This work proposes a PE design which can work correctly in two of the dataflows, namely weight-stationary and output-stationary.

3.3 Control and Memory subsystem

The control and memory subsystem of the accelerator can broadly be split into two parts - the frontend for fetching and executing the accelerator instructions and buffers for storing input and output maps. The frontend consists of task queues to which tasks are streamed into by the core, and configuration registers which store

the runtime parameters of the current operation.

The frontend consists of instruction queues, one queue for each type of instruction - LOAD, STORE, GEMM and ALU operation, with each instruction being 128-bit wide. The frontend has an instruction fetch module, which fetches instructions from the main memory by interacting with the interface module. In addition, there is a load and a store unit that takes in the instruction from the command queue, contains address generation units that generates the relevant set of addresses to fetch from the main-memory and a dependency resolution module which resolves the dependency across instructions coming from the command queue.

The buffers of relevance are Global buffer and Accumulator buffer, which store input maps and output maps respectively. Both buffers are banked structures, and hence support multiple accesses simultaneously. Global buffer is used to store input feature maps. To maximize utilization, in each cycle, one value should be sent into each row. Accumulator buffer is used to store output feature maps. The minimum number of banks in the accumulator buffer is same as the number of columns present in the systolic grid. Each column produces one output value per cycle, and hence, equal number of banks are needed to store them.

3.4 Mapping DNNs to Systolic Array

The stationary values are first unrolled and fed to each columns where they are kept stationary in the PEs. The inputs are streamed across the rows. The output partial-sums/weights are streamed across the columns in WS/OS dataflows respectively. In WS dataflow, the outputs along the column are simply pushed inside Accumulator buffer. However for OS dataflow, an extra step for draining the outputs is used and the outputs are pushed into buffers in this final stage.

Chapter 4

Dataflows in Systolic Architecture

4.1 Introduction

This chapter explains the two dataflows used for the convolution operation, namely the output-stationary and weight stationary dataflows.

4.2 Output-Stationary Dataflow

4.2.1 Data movement

In output-stationary dataflow, initial accumulator values(which will be 0 if there are no initial values) are stored inside the PE. The weight values flow in north-south direction, whereas the input values flow in west-east direction. In each cycle of Multiply phase, the PE multiplies the incoming weight value with the incoming input value and adds it to the accumulator value stored inside its stationary register. The weight and input values are then pushed in respective directions without any change. Once all the multiplications are done, the array pushes the final accumulator values inside the AccumFIFOs at the rate of one element per column per cycle during the drain phase.

4.2.2 Operational view of PE

At the start of each cycle, PE dequeues NSData and looks at the associated mode value. It then either dequeues we_data(if mode=Os_mac) or not. In the latter case, it also compares the associated counter value with the coordinate value stored inside rg_coord. If there is a match, ns_data is written into rg_stationary and the previous value of rg_stationary is pushed along with counter and mode values in the south direction. In case of no match, NSData is pushed in the south direction without making any changes.

If the mode value is Os_mac, we_data is dequeued as well. The PE then multiplies this value with the ns_data and adds the result to partial accumulator value stored inside rg_stationary. NSData and we_data are then pushed in south and east directions respectively without any change.

4.3 Weight-Stationary Dataflow

4.3.1 Data movement

In weight-stationary dataflow, weight values are stored inside the PE. The accumulator values flow in north-south direction, whereas the input values flow in west-east direction. In each cycle of Multiply phase, the PE multiplies the weight value stored inside its

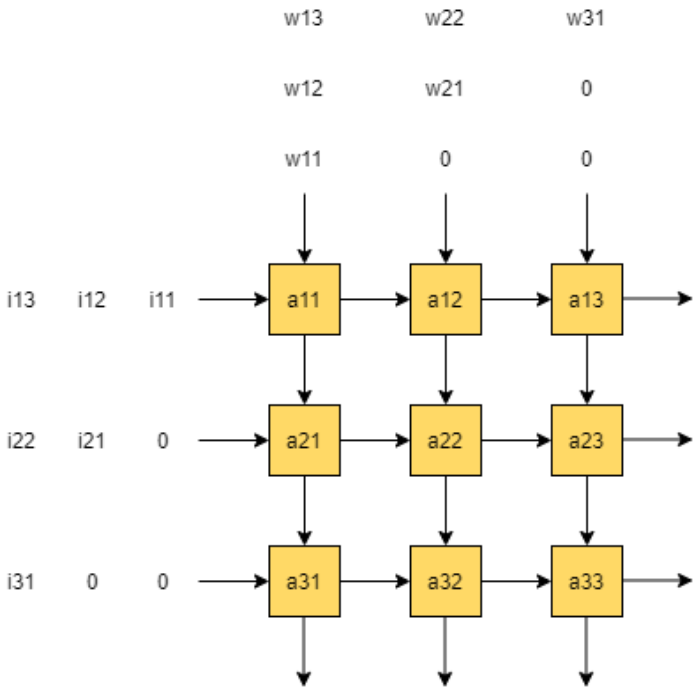


Figure 4.1: Output-stationary dataflow

stationary register with the incoming input value and adds it to the accumulator value coming from north direction. Once the accumulator value gets updated by all the PEs in the column, it is ready and already present at the end of the column. We can then connect an AccumFIFO and push the final accumulator value into it.

4.3.2 Operational view of PE

At the start of each cycle, PE dequeues NSData and looks at the associated mode value. It then either dequeues we_data(if mode=Ws_mac) or not. In the latter case, it also compares the associated counter value with the coordinate value stored inside rg_coord. If there is a match, ns_data is written into rg_stationary

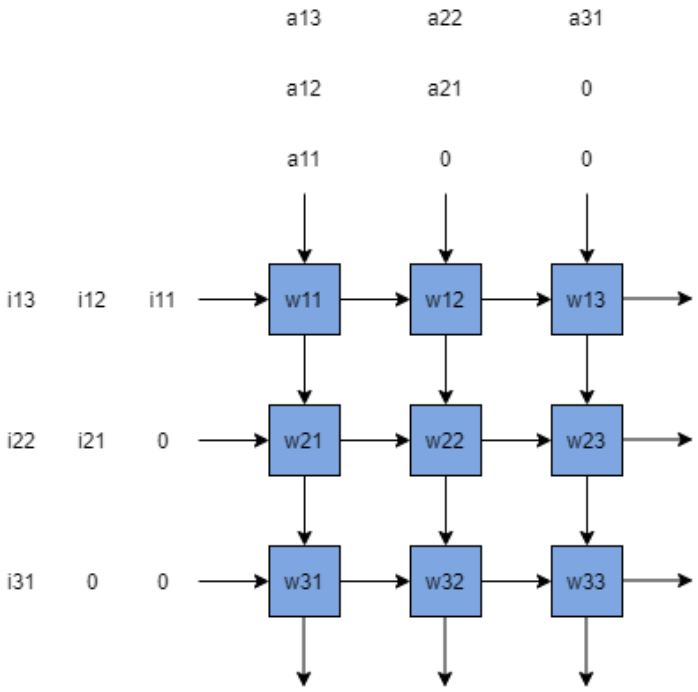


Figure 4.2: Weight-stationary dataflow

and the previous value of rg_stationary is pushed along with counter and mode values in the south direction. In case of no match, NSData is pushed in the south direction without making any changes.

If the mode value is Ws_mac, we_data is dequeued as well. The PE then multiplies this value with the rg_stationary value and adds the result to partial accumulator value coming from north direction. The updated NSData and we_data are then pushed in south and east directions respectively.

Chapter 5

Design of Multidataflow Processing Element

5.1 Introduction to mode-wise approach

A multidataflow PE should operate in both the dataflows. Because of this, there will be a lot of control flow and thus we come up with such a mode-wise approach for tackling this issue. The reason is that a complete operation in any of the phases can be divided into various modes. And it was observed that the setup operation in both dataflows are functionally same. Even the drain operation for output-stationary dataflow is similar, it is just that we need an extra label to let the acccumulator FIFOs know that this is a fresh copy of output which needs to be stored in a particular location. No such observations were made while comparing the MAC phases, this is so because the second multiplicand and the result destinations itself differ.

With these points in mind, this chapter introduces a multidataflow PE and its construction. We then discuss how the PE operates in each of the modes. Chapter 6 then explains how do we construct a 2D array using MDF-PEs. Chapter 7 discusses the order in which the top module needs to push in the corresponding data elements to get desired outputs.

5.2 Multidataflow PE

Figure 5.1 gives a broader control and data level view of MDF-PE. It has receiver interfaces in north and west direction and is able to receive data elements of type NSData and we_data respectively in each cycle. The PE then has similar transmitter interfaces in south and east direction, which are able to send data elements of type NSData and we_data respectively.

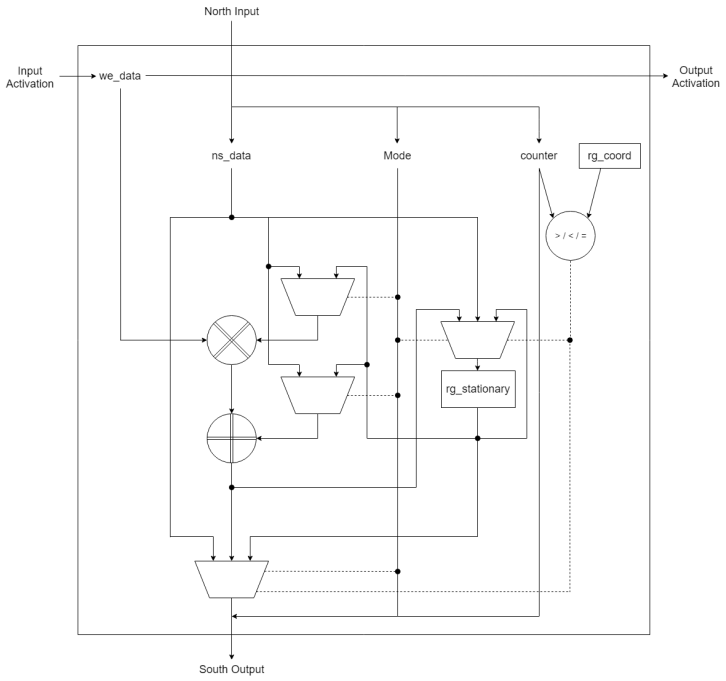


Figure 5.1: Multidataflow PE

5.3 PE operation in different Modes

We currently are using 4 different modes, namely: Setup, Ws_mac, Os_mac and Os_drain.

5.3.1 Setup mode

This mode is used by MDF-SA to set initial values in rg_stationary registers of the PEs. In WS dataflow, these are the weight data elements whereas in OS dataflow, these are the inital biases for the output data elements.

A PE operates in this mode if the south FIFO is notFull and the north FIFO is notEmpty, with the first mode value being Setup. In this mode, the PE compares the rg_coord value with the counter value. If both of them match, ns_data is copied into rg_stationary and the rg_stationary element is pushed through south TXe interface, along with counter and mode values. If there is no match, the incoming NSData is directly pushed through south TXe interface.

Pseudo code for Setup mode

```
if(mode==Setup){
   NSData_in=deq.from.north();
   if(counter==rg_coord){
      rg_stationary=NSData_in.ns_data;
      NSData_out={rg_stationary,NSData_in.mode,NSData_in.counter};
   }else{
      NSData_out=NSData_in;
   }
   enq.to.south(NSData_out);
```

5.3.2 Ws_mac mode

This mode is used by MDF-SA to perform a MAC operation in WS dataflow. Incoming ns_data value is the partial accumulator value, and the value stored in rg_stationary is the weight data element. A PE operates in this mode if the south and east FIFOs are notFull, and the west and north FIFOs are notEmpty, with the first mode value being Ws_mac. In this mode, the PE multiplies rg_stationary value with input we_data and adds it to the input ns_data. The updated ns_data is then pushed through south TXe interface, along with counter and mode values. , The we_data is pushed through east TXe interface

Pseudo code for Ws_mac mode

```
if(mode==Ws_mac){
    NSData_in=deq.from.north();
    WEData=deq.from.west();
    new_accum=WEData*rg_stationary + NSData_in.ns_data;
    NSData_out={new_accum,NSData_in.mode,NSData_in.counter};
    enq.to.south(NSData_out);
    enq.to.east(WEData);
}
```

5.3.3 Os_mac mode

This mode is used by MDF-SA to perform a MAC operation in OS dataflow. Incoming ns_data value is the weight value, and the value stored in rg_stationary is the partial accumulator value.
A PE operates in this mode if the south and east FIFOs are notFull, and the west and north FIFOs are notEmpty, with the first mode value being Os_mac. In this mode, the PE multiplies input ns_data value with input we_data and adds it to the input rg_stationary. The NSData is then pushed through south TXe interface, and we_data is pushed through east TXe interface.

Pseudo code for Os_mac mode

```
.....................................................................

   if(mode==Os_mac){
      NSData_in=deq.from.north();
      WEData=deq.from.west();
      rg_stationary=WEData*NSData_in.ns_data + rg_stationary;
      enq.to.south(NSData_in);
      enq.to.east(WEData);
   }

.....................................................................
```

5.3.4 Os_drain mode

This mode is used by MDF-SA to drain the accumulator values from rg_stationary registers of the PEs in OS dataflow.
A PE operates in this mode if the south FIFO is notFull and the north FIFO is notEmpty, with the first mode value being Os_drain. In this mode, the PE compares the rg_coord value with the counter value. If both of them match, ns_data is copied into rg_stationary and the rg_stationary element is pushed through south TXe interface, along with counter and mode values. If there is no match, the incoming NSData is directly pushed through south TXe interface.

Pseudo code for Os_drain mode

```
if(mode==Os_drain){
   NSData_in=deq.from.north();
   if(counter==rg_coord){
      rg_stationary=NSData_in.ns_data;
      NSData_out={rg_stationary,NSData_in.mode,NSData_in.counter};
   }else{
      NSData_out=NSData_in;
   }
   enq.to.south(NSData_out);
```

Chapter 6

Design of Multidataflow Array

6.1 Introduction

The MDF-SA consists of multidataflow PEs connected in a grid like structure by a top module using TxRx custom package. The inputs and outputs to the array were pushed in and extracted using the Get-Put interfaces.

The we_data is the input data element of the systolic array and travels along West-East direction. The ns_data is the weight/output data element in OS/WS dataflows respectively. This ns_data is clubbed along with an 8-bit counter value and an enumerated mode value to form an NSData data structure. NSData travels along North-South direction.

Figure 5.1 shows the basic multidataflow PE. These PEs are connected in a grid-like structure to form a systolic array, as shown in figure 6.1.

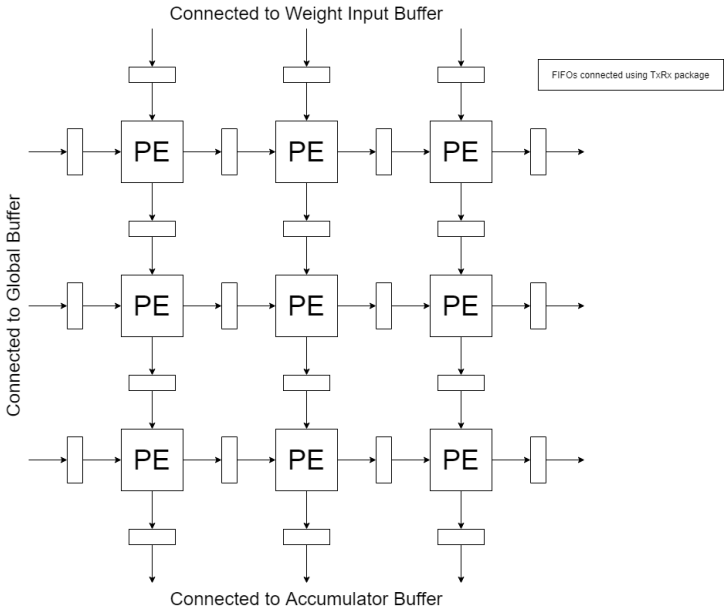


Figure 6.1: Multidataflow Systolic Array

6.2 Top Module

A top module is used to instantiate the MDF-SA, along with defining nRow, nCol, bWidth and twbWidth parameters. The intra-array connections between FIFOs and PEs through TxRx interfaces are done here. The input and output GetPut interfaces for the array are exposed to corresponding buffers so as to receive inputs and send out the outputs.

6.3 Interfacing

6.3.1 Array Connections using TxRx custom package

The PEs were connected in a top file using custom TxRx package developed by Shakti team, IIT Madras.
For every connection along North-South or East-West direction, we instantiate a FIFO. Inside the PE to be connected to the input side of this FIFO, a transmitter(TX) interface is created. Inside the PE to be connected to the output side of this FIFO, a receiver(RX) interface is created. These interfaces are connected to the FIFO inside the top module.

6.3.2 Connections to Buffers

At the edges of the array, we declare another set of input-output FIFOs. FIFOs along West and North edges of the array have their input sides connected to input buffers using GetPut interfaces, whereas their output sides are connected to PEs in first column and row respectively using TX interface. FIFOs along East and South edges of the array have their output sides connected to output buffers using GetPut interfaces, whereas their input sides are connected to PEs in last column and row respectively using RX interface.

Chapter 7

Operation of Multidataflow Array

7.1 Introduction

The MDF-SA should operate in OS and WS dataflows. This chaptern describes how the inputs should be pushed in along the rows and columns to ensure correct operation in either dataflows. We will also look at how switches between these modes can be carried out so that no data is lost.
Let us assume we have instantiated an MxN array. Similarly, let us assume we want to perform X MAC operations in each phase.

7.2 WS Operation

7.2.1 Setting up weights for first phase

Before the start of MAC operations, we want to set up weight values correctly in rg_stationary registers of each PE. Every PE has an inbuilt coordinator value stored in its rg_coord register. This value stores the row number of PE, and takes value from the set {1,...M}. To set the weights correctly, the north input FIFO needs to push NSData = {Weight_i, i, Setup} $\forall$ i $\in \{1, ...M\}$. The order in

which weights are being pushed in do not matter as long as they are accompanied by the corresponding counter values. At the south output FIFO, the mode value 'Setup' can be used to infer that the accompanying ns_data is not convolution output and should be discarded.

7.2.2 Performing MAC operations

During MAC phase, the north input FIFO needs to push NSData = {InitialAccum$_i$, i, Ws_mac} $\forall$ i $\in \{1, ...X\}$. In MAC phase, each PE waits for WEData to be present at the west input FIFO before starting the operation. Thus as long as relative order is maintained inside each input FIFO, we can guarantee correct operation. At the south output FIFO of the array, the counter value 'i' and mode value 'Ws_mac' can be used to store the output of convolution at the corresponding location inside the buffer.

7.2.3 Setting up weights for next phase

The operation described in section 7.2.1 can be repeated to push in next set of inputs. The entire procedure can then be repeated for the subsequent set of operations in MAC phase.

7.3 WS to OS Switch

7.3.1 Setting up initial accumulator values

The following procedure needs to be performed if the user has to either start operations directly in OS dataflow or switch from WS to OS dataflow. The initial value in rg_stationary of each PE will be either '0' or W_i from the previous operation set. For operating in OS dataflow, the user should set up initial biases in rg_stationary of each PE. To do so, the north input FIFO needs to push NSData = {InitialAccum$_i$, i, Setup} $\forall$ i $\in \{1, ...M\}$. The order in which the accumulator values are pushed do not matter as long as they are accompanied by the corresponding counter value. At the south output FIFO, the mode value 'Setup' can be used to infer that the accompanying ns_data is not convolution output and should be discarded.

Cycle	P1			P2			P3			Column Output	
	ns_data(counter)	mode	rg_stationary	ns_data(counter)	mode	rg_stationary	ns_data(counter)	mode	rg_stationary	ns_data(counter)	mode
1	**W1(1)**	**Setup**	**0**	-	-	0	-	-	0	-	-
2	W2(2)	Setup	W1	0(1)	Setup	0	-	-	0	-	-
3	W3(3)	Setup	W1	**W2(2)**	**Setup**	**0**	0(1)	Setup	0	-	-
4	A[1](1)	Ws_mac	W1	W3(3)	Setup	W2	0(2)	Setup	0	0(1)	Setup
5	A[2](2)	Ws_mac	W1	A[1]'(1)	Ws_mac	W2	**W3(3)**	**Setup**	**0**	0(2)	Setup
6	A[3](3)	Ws_mac	W1	A[2]'(2)	Ws_mac	W2	A[1]'(1)	Ws_mac	W3	0(3)	Setup
7	A[4](4)	Ws_mac	W1	A[3]'(3)	Ws_mac	W2	A[2]'(2)	Ws_mac	W3	**A[1]"(1)**	**Ws_mac**
MAC Operations											
X+3	A[X](X)	Ws_mac	W1	A[X-1]'(X-1)	Ws_mac	W2	A[X-2]'(X-2)	Ws_mac	W3	**A[X-3]"(X-3)**	**Ws_mac**
X+4	**Y1(1)**	**Setup**	**W1**	A[X]'(X)	Ws_mac	W2	A[X-1]'(X-1)	Ws_mac	W3	**A[X-2]"(X-2)**	**Ws_mac**
X+5	Y2(2)	Setup	Y1	Y1(1)	Setup	W2	A[X]'(X)	Ws_mac	W3	**A[X-1]"(X-1)**	**Ws_mac**
X+6	Y3(3)	Setup	Y1	**Y2(2)**	**Setup**	**W2**	Y1(1)	Setup	W3	**A[X]"(X)**	**Ws_mac**

Figure 7.1: Weight Stationary operation: *Wi*s are the weights, *A[i]*s are initial accumulator value, *A[i]'*s are partial output accumulator values and *A[i]"*s are final output accumulator values. *Yi*s are stationary values for the next set of operations, which will either be weights/accumulator values for WS/OS dataflow

7.4 OS Operation

7.4.1 Performing MAC operations

During MAC phase, the north input FIFO needs to push NSData = $\{\text{Weight}_i, \text{i}, \text{Os\_mac}\}\ \forall\ \text{i} \in \{1, ...X\}$. In MAC phase, each PE waits for we_data to be present at the west input FIFO before starting the operation. Thus as long as relative order is maintained inside each input FIFO, we can guarantee correct operation. At the south output FIFO of the array, the mode value 'Os_mac' can be used to infer that the accompanying ns_data is not convolution output and should be discarded.

7.4.2 Draining out the outputs

During the drain phase, the north input FIFO needs to push NSData = {Y_i, i, Os_mac} $\forall$ i $\in \{1, ...M\}$. Y_i in this regard refers to the stationary value which the user wants to use for the next MAC operation phase. The order in which NSData is pushed in does not matter, as long as it is accompanied by the corresponding counter value. At the south output FIFO of the array, the counter value 'i' and mode value 'Os_drain' can be used to store the output of convolution at the corresponding location inside the buffer.

7.5 OS to WS Switch

For carrying out this switch operation, the north input FIFO needs to push NSData = {$Weight_i$, i, Os_drain} $\forall$ i $\in \{1, ...M\}$ after the final Os_mac phase. $Weight_i$ in this regard refers to the weight value which the user wants to use for the next WS dataflow operation. At the south output FIFO of the array, the counter value 'i' and mode value 'Os_drain' can be used to store the output of convolution at the correct location inside the buffer. Once all the weights for next WS operation are pushed in and outputs from the previous OS operation are drained out, procedure mentioned in section 7.2.2 should be carried out to perform the MAC operations for current WS phase.

Cycle	P1			P2			P3			Column Output	
	ns_data(counter)	mode	rg_stationary	ns_data(counter)	mode	rg_stationary	ns_data(counter)	mode	rg_stationary	ns_data(counter)	mode
1	**A1(1)**	**Setup**	**0**	-	-	0	-	-	0	-	-
2	A2(2)	Setup	A1	0(1)	Setup	0	-	-	0	-	-
3	A3(3)	Setup	A1	**A2(2)**	**Setup**	**0**	0(1)	Setup	0	-	-
4	W[1](1)	Os_mac	A1	A3(3)	Setup	A2	0(2)	Setup	0	0(1)	Setup
5	W[2](2)	Os_mac	A1'	W[1](1)	Os_mac	A2	**A3(3)**	**Setup**	**0**	0(2)	Setup
6	W[3](3)	Os_mac	A1'	W[2](2)	Os_mac	A2'	W[1](1)	Os_mac	A3	0(3)	Setup
7	W[4](4)	Os_mac	A1'	W[3](3)	Os_mac	A2'	W[2](2)	Os_mac	A3'	W[1](1)	Os_mac
MAC Operations											
X+3	W[X](X)	Os_mac	A1'	W[X-1](X-1)	Os_mac	A2'	W[X-2](X-2)	Os_mac	A3'	W[X-3](X-3)	Os_mac
X+4	**Y1(1)**	**Os_drain**	**A1"**	W[X](X)	Os_mac	A2'	W[X-1](X-1)	Os_mac	A3'	W[X-2](X-2)	Os_mac
X+5	Y2(2)	Os_drain	Y1	A1"(1)	Os_drain	A2"	W[X](X)	Os_mac	A3'	W[X-1](X-1)	Os_mac
X+6	Y3(3)	Os_drain	Y1	**Y2(2)**	**Os_drain**	**A2"**	A1"(1)	Os_drain	A3"	W[X](X)	Os_mac
X+7	x	x	x	Y3(3)	Os_drain	Y2	A2"(2)	Os_drain	A3"	**A1"(1)**	**Os_drain**
X+8	x	x	x	x	x	x	**Y3(3)**	**Os_drain**	**A3"**	**A2"(2)**	**Os_drain**
X+9	x	x	x	x	x	x	x	x	x	**A3"(3)**	**Os_drain**

Figure 7.2: Output Stationary operation: *W[i]*s are the weights, *Ai*s are initial accumulator value, *Ai'*s are partial output accumulator values and *Ai"*s are final output accumulator values. *Yi*s are stationary values for the next set of operations, which will either be weights/accumulator values for WS/OS dataflow

Chapter 8

Results and Discussions

8.1 Introduction

In this section, we will look at simulation and hardware synthesis results of the MDF array in both, OS and WS dataflows. We will then comment on the pros and cons of the current design and finally look at which design suits well for what type of workloads.

8.2 Simulation Results

This section shows the simulation results of the array. **NOTE:** The rg_coord value of the PE is the row value of PE + 1.This means for PE[1][1], rg_coord=2.

8.2.1 Weight Stationary

A 3x3 SA was initialised with twbWidth=32 bits and bWidth=8bits. The operation involved setting up weights for the first phase, performing 5 MAC operations for each weight element, followed by setting up new set of weights for next operation.

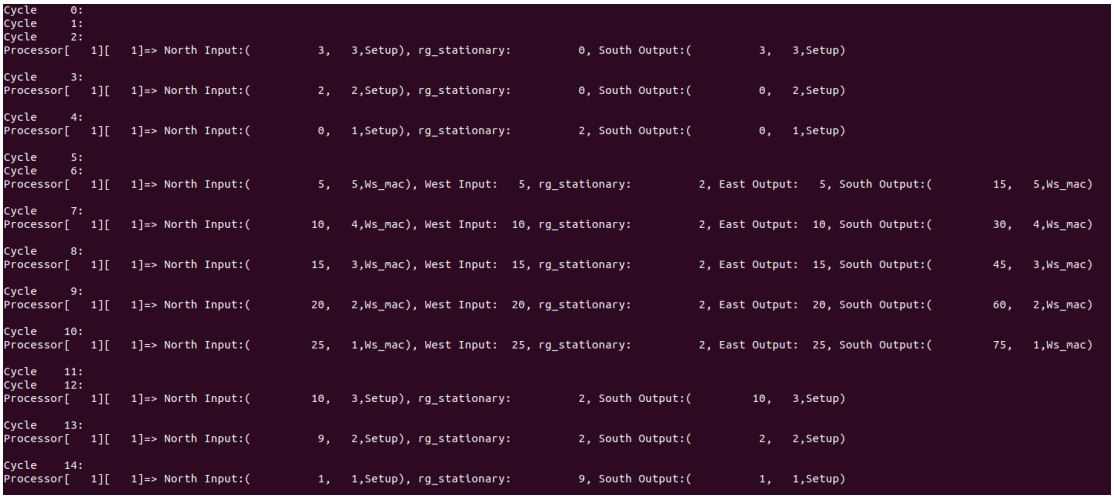


Figure 8.1: WS PE view: The result shows how operations are being carried out in PE[1][1]

Refer figure 8.1. Initially, there is a 2-cycle delay. That is because the top module(in this case the testbench) takes one cycle to fill the first element in the column FIFO of the SA. Because of the way the testbench was constructed, we are encountering one cycle delay everytime a mode switch happens. These delays can be removed if the data elements are ready and pushed into column FIFOs well in time.

In the next cycle, PE[0][1] processes on the first data element. Thus, we see that PE in 2nd row begins its operation in 3rd cycle. In cycle 2 and cycle 4, since the PE encounter Setup mode but the counter values don't match, the rg_stationary value remains same for the following cycle. The NSData element is pushed southwards without any change. However in cycle 3, Setup mode is encountered and counter values match. Hence the rg_stationary copies the ns_data value(which is reflected in the next cycle) and

the initial value of rg_stationary is pushed instead.

During the MAC phase(cycles 6 - 10)

$$South\ Output = North\ Input + West\ Input * rg\_stationary$$

The behaviour of the PE in cycles 12-14 is similar to that in cycles 2-4. This is expected as both of them are actually the setup phases. The only difference between the sets is the value of the ns_data elements(weights in this case).

```
Col:   0=> NSData sent:(          3,   3,Setup)
Col:   0=> NSData sent:(          2,   2,Setup)
Col:   0=> NSData sent:(          1,   1,Setup)
Col:   0=> NSData sent:(          0,   5,Ws_mac)
Col:   0=> NSData sent:(          0,   4,Ws_mac)
Col:   0=> NSData sent:(          0,   3,Ws_mac)
Col:   0=> NSData sent:(          0,   2,Ws_mac)
Col:   0=> NSData sent:(          0,   1,Ws_mac)
Col:   0=> NSData output received:(         30,1)
Col:   0=> NSData output received:(         60,1)
Col:   0=> NSData sent:(         10,   3,Setup)
Col:   0=> NSData output received:(         90,1)
Col:   0=> NSData sent:(          9,   2,Setup)
Col:   0=> NSData output received:(        120,1)
Col:   0=> NSData sent:(          8,   1,Setup)
Col:   0=> NSData output received:(        150,1)
```

Figure 8.2: WS Column view: The results show how the inputs should be pushed inside column FIFOs and see if we get correct results at the end of the array

8.2.2 Output Stationary

A 3x3 SA was initialised with twbWidth=32 bits and bWidth=8bits. The operation involved setting up initial

accumulator values for the first phase, performing 5 MAC operations for each output element, followed by setting up new set of accumulator elements for next operation.

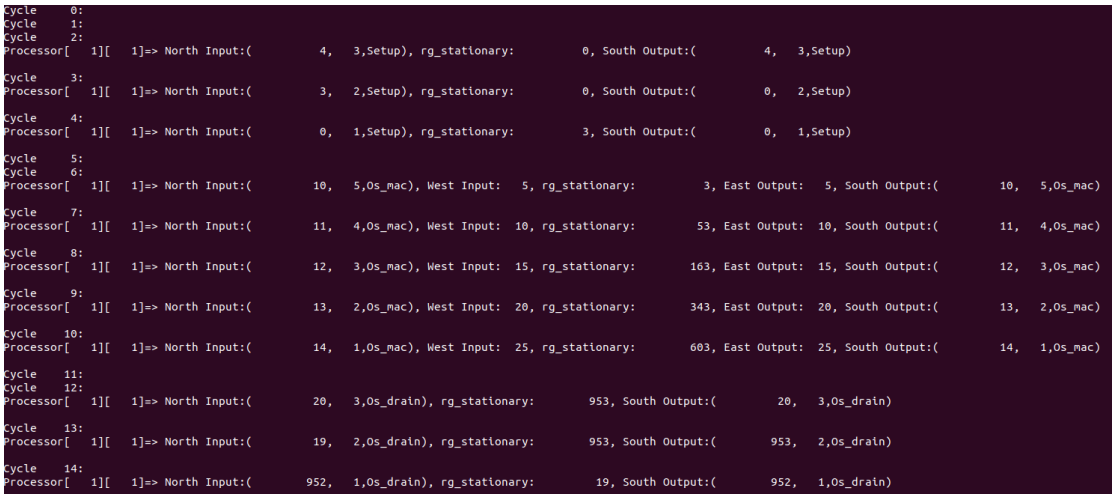


Figure 8.3: OS PE view: The result shows how operations are being carried out in PE[1][1]

Refer figure 8.3. The delays are similar to the ones seen in weight stationary dataflow. We have a 2-cycle delay for PEs in second row, followed by one cycle delay for each mode change because of testbench structure. The setup phase of OS dataflow(cycles 2-4) is exactly the same as presented in WS dataflow. During the MAC phase(cycles 6 - 10)

$$rg\_stationary = North\ Input + West\ Input * rg\_stationary$$

NSData is pushed without modifying by the PE as they contain weight elements which should not be modified. The change in rg_stationary is reflected in next cycle, as register writes have one cycle write time.

The drain phase(cycles 12-14) is operationally similar to the Setup

phase. However we need to differentiate between the two because ns_data elements popped along with Os_drain should be pushed inside the accumulator buffer and not the weight buffer. This is demonstrated in figure 8.4, where the column FIFO prints the output value when the ns_data is popped along with Os_drain mode.

```
Col:   0=> NSData sent:(          4,   3,Setup)
Col:   0=> NSData sent:(          3,   2,Setup)
Col:   0=> NSData sent:(          2,   1,Setup)
Col:   0=> NSData sent:(         10,   5,Os_mac)
Col:   0=> NSData sent:(         11,   4,Os_mac)
Col:   0=> NSData sent:(         12,   3,Os_mac)
Col:   0=> NSData sent:(         13,   2,Os_mac)
Col:   0=> NSData sent:(         14,   1,Os_mac)
Col:   0=> NSData sent:(         20,   3,Os_drain)
Col:   0=> NSData sent:(         19,   2,Os_drain)
Col:   0=> NSData sent:(         18,   1,Os_drain)
Col:   0=> NSData output received:(        954,   3)
Col:   0=> NSData output received:(        953,   2)
Col:   0=> NSData output received:(        952,   1)
```

Figure 8.4: WS Column view: The results show how the inputs should be pushed inside column FIFOs and see if we get correct results at the end of the array

8.3 Discussions on the design

Thus after looking at the results, we can see that we have an operational Systolic Array with Multidataflow support. However, just like any other design, the current hardware design has its own pros and cons.

Pros:

- **Switching**: It is easier to switch between different dataflows

and also between different modes of the same dataflow. Since every data element is accompanied by a corresponding mode value, we just need to push in the NSData elements in the desired order.

- **No global signals**: Using mode-wise approach also serve another purpose. We do not need to have global signals to switch the array from one operational mode to other.
- **Row-column independency**: The logic of the PE does not allow the we_data to dequeue if the PE does not have corresponding ns_data in any of the MAC modes. Similarly, the MAC data element doesn't get dequeued/pushed forward before we_data element is available. Hence to get the desired operation, the user just needs to make sure that relative order of pushing inputs along each row and column is correct and not worry about the cycle times. So as can be seen in figures 4.2 and 4.1, we do not need to push those '0's (or wait for those many cycles).

Cons:

- **Hardware costs**: Because of this mode-approach, we now have to have an extra register to store this value for each PE. One can come up with alternate designs(like usage of some global register) to counter this, but then it would impact the latency of the array.
- **Complexity at top module**: The top module responsible for pushing and popping the inputs and outputs now has to arrange for this mode value as well. It might be possible to remove this complexity if we use a design similar to the one mentioned in the previous point.

Potential Areas of improvement:

- **Getting rid of mode approach**: It might be possible to get rid of mode approach and save the corresponding hardware registers if we use a global register for array itself. This register can then be used to send interrupts to each PE.

However, the new design then has to have provisions for interrupts and high delays will be incurred.

- **Using a different connection interface**: The current design uses TxRx interfaces which are connected to the FIFOs. The FIFO behaves like a flip-flop and enqueue/dequeue signal behaves like the enable signals. It might be possible to get rid of these FIFOs by using some alternate connection interfaces.
- **Multi-data Support**: In the current design, ns_data(twbWidth bits wide) and we_data(bWidth bits wide) are expected to have only one data element. By investing in some extra hardware, we can extend the support for multiple data elements. Basically, we can use bit partitioning to send n data elements along the column(twbWidth/n bits wide) and along the row(bWidth/n bits wide) in each cycle.

8.4 Synthesis Results

Synthesis for the three dataflows(WS, Os and MDF) was carried out in Vivado 2019.2 on Spartan SP701 evaluation platform. A 3x3 SA was initialised with twbWidth=32 bits and bWidth=8 bits.

Following results show hardware requirements for one PE.

```
Hierarchical RTL Component report
Module mkintMulWS
Detailed RTL Component Info :
+---Adders :
       2 Input     32 Bit       Adders := 1
+---Registers :
                   32 Bit    Registers := 1
                    8 Bit    Registers := 1
+---Multipliers :
                    32x32  Multipliers := 1
+---Muxes :
       2 Input     41 Bit        Muxes := 1
       2 Input     32 Bit        Muxes := 1
```

Figure 8.5: HW: Weight Stationary PE

```
Hierarchical RTL Component report
Module mkintMulWS
Detailed RTL Component Info :
+---Adders :
       2 Input     32 Bit       Adders := 1
+---Registers :
                   32 Bit    Registers := 1
                    8 Bit    Registers := 1
+---Multipliers :
                   32x32  Multipliers := 1
+---Muxes :
       2 Input     42 Bit        Muxes := 1
       2 Input     32 Bit        Muxes := 2
```

Figure 8.6: HW: Output Stationary PE

```
Hierarchical RTL Component report
Module mkintMulWS
Detailed RTL Component Info :
+---Adders :
       2 Input     32 Bit       Adders := 2
+---Registers :
                   32 Bit    Registers := 1
                    8 Bit    Registers := 1
+---Multipliers :
                   32x32  Multipliers := 2
+---Muxes :
       4 Input     42 Bit        Muxes := 1
       2 Input     32 Bit        Muxes := 2
```

Figure 8.7: HW: Multidataflow PE

Refer figures 8.5, 8.6 and 8.7. As expected, WS requires the least amount of hardware. This is so because we need only two modes, namely Setup and Ws_mac for this dataflow. OS PE, on the other hand, requires an extra 32-bit MUX as compared to WS. The reason behind this might be that in OS dataflow, rg_stationary register must be written by ns_data during the Setup mode and an extra Os_drain mode(which is not present in WS dataflow). Finally as Multidataflow itself has a lot of control flow, we see the highest hardware requirements for this PE.

```
Hierarchical RTL Component report
Module FIFO2
Detailed RTL Component Info :
+---Registers :
                   41 Bit    Registers := 2
                    1 Bit    Registers := 2
+---Muxes :
       2 Input      1 Bit        Muxes := 3
Module FIFO2__parameterized0
Detailed RTL Component Info :
+---Registers :
                    8 Bit    Registers := 2
                    1 Bit    Registers := 2
+---Muxes :
       2 Input      1 Bit        Muxes := 3
```

Figure 8.8: HW: Weight Stationary FIFO Connectable

```
Hierarchical RTL Component report
Module FIFO2
Detailed RTL Component Info :
+---Registers :
                   42 Bit    Registers := 2
                    1 Bit    Registers := 2
+---Muxes :
       2 Input      1 Bit        Muxes := 3
Module FIFO2__parameterized0
Detailed RTL Component Info :
+---Registers :
                    8 Bit    Registers := 2
                    1 Bit    Registers := 2
+---Muxes :
       2 Input      1 Bit        Muxes := 3
```

Figure 8.9: HW: Output Stationary FIFO Connectable

```
Hierarchical RTL Component report
Module FIFO2
Detailed RTL Component Info :
+---Registers :
                   42 Bit    Registers := 2
                    1 Bit    Registers := 2
+---Muxes :
       2 Input      1 Bit        Muxes := 3
Module FIFO2__parameterized0
Detailed RTL Component Info :
+---Registers :
                    8 Bit    Registers := 2
                    1 Bit    Registers := 2
+---Muxes :
       2 Input      1 Bit        Muxes := 3
```

Figure 8.10: HW: Multidataflow FIFO Connectable

Refer figures 8.8, 8.9 and 8.10. Each PE uses one 8-bit FIFO

connectable module in West-East direction and an x-bit FIFO connectable in North-South direction. 'x' takes the value of 41 in WS as we have 32-bits for ns_data, 8-bits for counter and 1-bit for mode(2 modes for WS). On the other hand, 'x' takes the value of 42 in OS and MDF as we have 32-bits for ns_data, 8-bits for counter and 2-bits for mode(3 modes for OS and 4 modes for MDF). Since the number of FIFOs used for each PE in all the modes is same, we do not see any other difference between the 3 results.

8.5 Discussions on the dataflow

Since a real NN will require a fully functional DNN accelerator, we did not offload an actual convolution operation on the MDF-SA. Hence we can qualitatively discuss which dataflow performs well in certain circumstances and how can we use the MDF-SA to speed the underlying MAC operations up.

Output stationary dataflow can be used if we want to minimizes Read/Write energy consumption for partial accumulators. Since the output is available only at the end of the MAC phase, high latency is involved. Thus OS dataflow should be used if latency is not of prime concern and the weight maps are too large to fit inside a limited size SA.

Weight stationary dataflow can be used if we want to minimizes Read/Write energy consumption for filter weights. Since the

output has a latency equal to the number of rows in the SA, the latency is low. Thus WS dataflow should be used if latency is of prime concern and if we want to generate/evaluate huge number of outputs which use the same weight, but number of *Weight * Input* terms are low.

A MDF-SA would be a good choice if the user wants to map multiple types of workloads on the same hardware, at the cost of some extra hardware per conventional single dataflow PE.

Chapter 9

Summary

The objective of the project was to design a Multidataflow SA design which can work in OS and WS dataflows and can be mapped on the same hardware. It involved using a mode-wise approach as different phases in the aforementioned dataflows require different types of operation to be performed inside the PE. The final design used 4 modes:

- **Setup:** For setting up initial accumulator/weight values
- **Ws_mac:** For performing MAC operation in WS dataflow
- **Os_mac:** For performing MAC operation in OS dataflow
- **Os_drain:** For draining out the final accumulator values after Os_mac phase

We were able to simulate the design in both OS and WS dataflows and found out that the operations were performed successfully. We were also able to extract and analyse the synthesis results for each PE block used inside the array.

Bibliography

[1] Hasan Genc et al. *Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures*. 2019. arXiv: 1911.09925 [cs.DC].

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

[3] H. T. Kung, Bradley McDanel, and Sai Qian Zhang. *Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization*. 2018. arXiv: 1811.04770 [cs.LG].