Memory Management in a Systolic Array Based Accelerator for Deep Learning

A Project Report

submitted by

MOHAN PRASATH G R

(EE15B044)

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2020

THESIS CERTIFICATE

This is to certify that the thesis titled **Memory Management in a Systolic Array Based Accelerator for Deep Learning**, submitted by **Mohan Prasath G R**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by him under our 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.

Prof. Pratyush Kumar
Research Guide
Assistant Professor
Dept. of CSE
IIT Madras, 600036

Prof. Janakiraman
Co-Guide
Assistant Professor
Dept. of Electrical Engineering
IIT Madras, 600036

Place: Chennai

Date: 2nd June 2020

ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Pratyush Kumar for guiding me and giving me an opportunity to work in this project.
I would also like to thank Gokulan and Vinod, who were part of this project, for helping me throughout this project.

ABSTRACT

KEYWORDS: systolic array ; deep learning; hardware acceleration.

Deep neural networks (DNNs) are widely used for various artificial intelligence applications. While they tend to perform well for various tasks, it also comes with the cost of high energy consumption and latency. In order to get the best out of DNNs it is important to use a better hardware which can deliver good performance with reasonable energy consumption. With the end of Moore's law, general purpose architectures are not able to deliver high performance with energy efficiency. This raised a need to build domain-specific architectures. Thus we look to build a hardware accelerator for DNN inference in edge devices which will interface with the Shakti C-Class processor. This work mainly discusses how data movement is done from the main memory to the accelerator and how data storage is done inside the accelerator. The main contributions of this work are design and implementation of the load module of the accelerator, implementing double buffering and design trade-off analysis of few design choices in the buffer module.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

DNN	Deep Neural Network
GPU	Graphics Processing Unit
FPGA	Field Programmable Gate Array
PE	Processing Elements
TPU	Tensor Processing Unit
ISA	Instruction Set Architecture
GEMM	GEneral Matrix Multiply
AXI	Advanced eXtensible Interface

NOTATION

N	Batch size of 3D feature map
M	Number of filters
C	feature map or filter channels
H/W	feature map height/width
R/S	filters height/width

CHAPTER 1

INTRODUCTION

Deep neural networks have gained popularity due to their success in various applications such as computer vision, speech and audio recognition, etc. However they require a lot of computations which results in high latency and energy consumption. This leaves us with a need to use hardware with high compute capabilities with limited energy consumption. With Moore's law coming to an end, we are not going to get the performance gains from CPUs as before and also with the end of Dennard scaling, peak power density doesn't remain constant anymore. GPUs became popular for running deep learning workloads. While GPUs give good compute capability it comes at a cost of high power consumption. FPGAs then became a popular choice for CNN based deep learning workloads. While FPGAs gave better performance per watt due to low power consumption, they were still less efficient than the application specific hardwares. Thus domain-specific hardware for deep learning became a need to get high performance and energy efficiency.

Convolution is an important operation and contributes to the majority of the compute time in DNN inference. Thus accelerating convolutions can give us a high overall improvement in performance. Systolic array (2) is an architecture that can compute convolutions / matrix multiplications efficiently. This is why a lot of existing DNN accelerators are based on systolic arrays.

Some early hardware accelerators for neural networks were from NeuFlow (3) (2011) and DianNao (4) (2014). NeuFlow contains a grid of processing tiles where each tile can be programmed to do a task in runtime. DianNao came up with an accelerator for large scale DNNs with special emphasis on the impact of memory.

Eyeriss (5) (2017) proposed a dataflow called Row stationary(RS) dataflow on a spatial architecture of 14x12 PE array. It also implements compression to improve energy efficiency. Google came up with a systolic array based accelerator called the Tensor Processing Unit (6) (2017) to accelerate their neural network workloads. In CNNs, the intermediate feature maps and the weights can be sparse i.e, consists of a lot of zeros in them. Accelerators like SCNN (7) (2017), EIE (8) (2016) look to exploit the sparsity and improve efficiency . Gemmini (9) (2019) is a systolic array based accelerator with configurable parameters. The idea is to generate custom systolic arrays for different deep learning workloads.

Thus we look to build an open source DNN hardware accelerator which will act as a co-processor interfacing with Shakti C-Class processor (1).

The rest of the thesis is organized as follows:

- Literature survey on two popular systolic array based accelerators - TPU and Gemmini in Chapter 2.
- The overall micro-architecture of the accelerator and the ISA will be briefed in Chapter 3.

The Contributions of this work are as follows:

- The design and implementation of the load module in different versions of the accelerator and doing the preliminary verification for the same. Implementation of double buffering in the initial version of the design. More on this will be discussed in Chapter 4
- Design trade-off analysis on the on-chip memory(buffer module) of the accelerator, discussing how can the port contention in buffers reduced. Trade-off analysis on using RISC or the CISC based ISA (Chapter 5)

CHAPTER 2

BACKGROUND

A Deep neural network is a neural network with multiple layers between the input layer and output layer. A common version of DNN is Convolution neural network(CNN) which consists of multiple convolution layers, pooling layers and fully connected layers. Data coming from each layer is termed as a feature map (fmap) which is sent as an input to the next layer. The convolution layer performs convolution operation on its inputs and sends the outputs to the next layer. Convolution layer is the most computationally intensive part of CNN. Pooling layer compresses the feature maps before sending it to the next layer. Two common pooling methods are max pooling and average pooling. Fully connected layers generally form the last layers of the CNNs and are memory bandwidth limited, rather than compute resource limited. The convolution layers and fully connected layers are generally followed by non-linear activation functions such as ReLU, sigmoid, etc.

2.1 Convolution layer

In 2D convolution, an output pixel is generated by performing element wise multiplication of 2D weight matrix and a part of input fmap, with the same dimension as weight matrix, and accumulating the values. The output fmap is generated by sliding the weight matrix over the 2D input fmap and performing this operation. In case of 3D input feature maps (C channels of H x W 2D fmaps), weights are of the form C x R x S where C is the number of channels, each channel of input convolves with corresponding channel of weight to generate C 2D output maps. These C channels are accumulated to get a single 2D output fmap (E x F). Multiple 3D filters(M) can be convolved with the input maps to generate 3D output fmap of the dimensions MxExF. The 3D inputs can often be sent as batches (N x C x H x W) to convolve with weights (M x C x R x S) and the outputs are of the dimensions N x M x E x F. This operation is a compute bound as it involves high reuse of data.

2.2 Fully connected layer

The input to this layer is either from the final convolution layer or pooling layer. The fmaps are unrolled to form a 1D vector and are passed to the fully connected layer. The 1D input is element wise multiplied with a 1D weight vector and accumulated to form an output element. Multiple weight vectors are multiplied with the same input to generate the output values. This layer is memory bandwidth limited but still offers some data reuse.

2.3 Systolic arrays

A systolic array is a network of interconnected processing elements(PE) that can accelerate compute bound operations like convolutions. Systolic array accelerates operations by reducing main memory accesses and reusing the data as much as possible. In CNNs there is scope for lots of data reuse thus we can reduce the main memory accesses in turn reducing overall latency and energy consumption. Convolution as such is not suitable for systolic arrays therefore it is converted into matrix multiplication operation and then computed by the systolic array. We use a 2D grid systolic array where each PE performs a multiply and accumulate (MAC) operation. There are different ways to reuse data in convolution and based on the reuse, the dataflow of the systolic arrays will be chosen. The data flows for convolution in 2D systolic arrays are input stationary, weight stationary and output stationary.

In our work we implement the weight stationary dataflow. In this setup, each PE stores a weight value, multiplies it with an input received from its left neighbour and adds it to the output of the PE present above and sends the result below. The inputs are being sent to PE present at the right. By this we try to make maximum reuse of the data and try to reduce the memory access as much as possible. In this dataflow only the inputs and outputs move whereas the weights once loaded remain in the PE, thus named as the weight stationary dataflow. The I/O movement only happens at the sides of the 2D array.

CHAPTER 3

LITERATURE REVIEW

In this chapter, we will discuss about two popular systolic-array based accelerators. One is the Gemmini accelerator from the University of California, Berkeley and other accelerator is the Tensor processing Unit (TPU) from Google, Inc. We will compare and analyse both the works based on their system architecture and the design choices behind it.

3.1 Tensor Processing Unit

In 2013, when Google (6) realized that people use voice search for 3 minutes a day, using speech recognition DNNs, would require their datacenters to double to meet the computation demands, which would be very expensive to satisfy with the conventional CPUs. They realized the need for a custom ASIC that would accelerate the inference workloads. The goal of the project was to improve the cost-efficiency by 10x over GPUs.

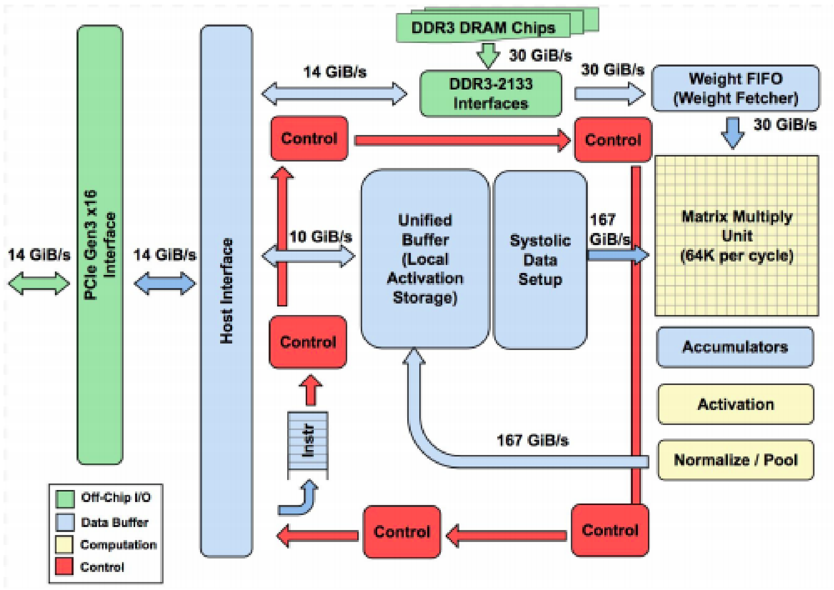


Figure 3.1: A system overview of the TPU from (6)

3.1.1 System Architecture:

Figure 3.1 shows the system overview of the TPU. The Matrix Multiply unit is the heart of the accelerator. It is a 256x256 sized systolic array that has PEs or MACs that can perform 8-bit multiply-and-adds on signed or unsigned integers. The reason for choosing 256x256 being, scaling further up (512*512) reduced the performance instead of increasing it.

TPU has a 24 MiB unified buffer that can store the inputs and intermediate results. One reason for having a unified buffer is that the intermediate results can be fed back to the systolic array as inputs. A programmable DMA controller is present and will act as a load and store modules to transfer data to and from host CPU memory to Unified buffer. For the weights, TPU has a on-chip FIFO that reads from a 8GiB off-chip DRAM. The weight FIFO can store upto four tiles of weights(4*256*256).

TPU has 4MiB of 32-bit Accumulators which will store the 16-bit products from the systolic array. The 4MiB represents 4096, 256-element, 32-bit accumulators. 256-element accumulators because 256 columns of the matrix unit send 256 values each cycle. The reason behind 4096 entries being, based on the roofline model the number of operations needed to reach peak performance was 1350, rounding off to 2048 entries, and the accumulators are duplicated so that compiler could use it for double buffering.

The TPU has support for 8-bit and 16-bit weights and activations. The array uses a weight stationary dataflow where it holds one 64KiB tile (256*256) of weights and another tile for double-buffering. TPU has hardware for performing non-linear activation functions and pooling.

3.1.2 Accelerator interfacing with the processor:

The TPU was designed to be a co-processor on the PCIe I/O bus allowing it to plug into the existing servers just like how GPU does. To simplify the hardware design and debugging, the host processor sends the instructions to the accelerator rather than the TPU fetching from the memory itself.

As instructions are sent over the relatively slow PCIe bus, TPU instructions follow the CISC tradition, including a repeat field. The average clock cycles per instruction (CPI) of these CISC instructions is typically 10 to 20.
Key TPU instructions are:

- **Read_Host_Memory** reads data from the CPU host memory into the Unified Buffer (UB).
- **Read_Weights** reads weights from Weight Memory into the Weight FIFO as input to the Matrix Unit.
- **MatrixMultiply/Convolve** causes the Matrix Unit to perform a matrix multiply or a convolution from the Unified Buffer into the Accumulators.
- **Activate** performs the nonlinear function of the artificial neuron, with options for ReLU, Sigmoid, and so on. Its inputs are the Accumulators, and its output is the Unified Buffer. It can also perform the pooling operations needed for convolutions using the dedicated hardware on the die, as it is connected to nonlinear function logic.
- **Write_Host_Memory** writes data from the Unified Buffer into the CPU host memory.

3.1.3 Software:

The portion of the application run on the TPU is typically written in TensorFlow and is compiled into an API that can run on GPUs or TPUs. The TPU stack is split into a User Space Driver and a Kernel Driver. The Kernel Driver is lightweight and handles only memory management and interrupts. It is designed for long-term stability. The User Space driver changes frequently. It sets up and controls TPU execution, reformats data into TPU order, translates API calls into TPU instructions, and turns them into an application binary. The User Space driver compiles a model the first time it is evaluated, caching the program image and writing the weight image into the TPU's weight memory; the second and following evaluations run at full speed.

3.2 Gemmini

Gemmini (9) was an attempt to build a systolic array generator rather than a fixed accelerator itself. The motive being different use-cases for DNN, from edge devices to cloud, can't be catered by a single type of systolic array accelerator. Thus Gemmini is a systolic array generator that enables systematic evaluations of deep learning architectures. The goal is to build a systolic array accelerator where we can tune important parameters like dataflow, precision, on-chip memory capacity and banking strategy and do a Design space exploration to figure out the right architectural parameters for common NN workloads.

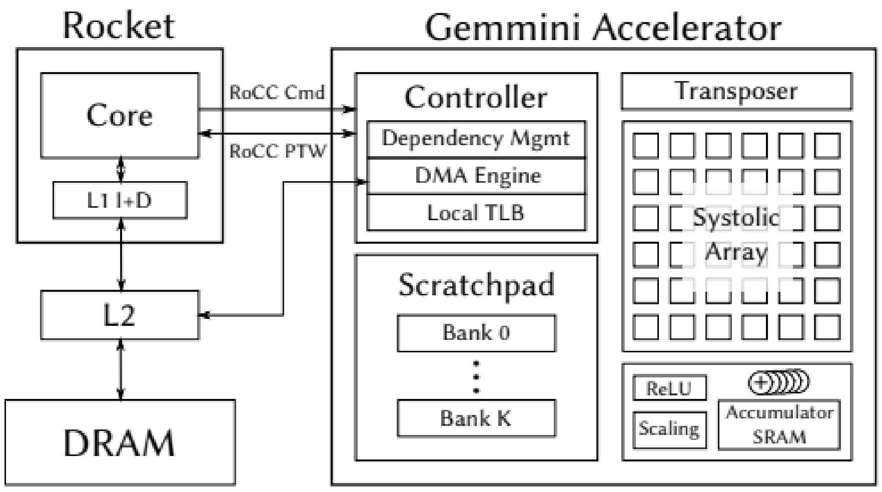


Figure 3.2: A system overview of the Gemmini systolic array generator from (9)

3.2.1 System Architecture:

Figure 3.2 shows the system overview of the Gemmini accelerator. The PEs in the systolic array are arranged in a combinational grid to form a tile, and the tiles are arranged in a pipelined grid to form the systolic array itself. This is done so that the pipeline depth of the array can be varied by changing the size of the tile. The PEs are double-buffered so that the weights can be loaded for next compute operation.

Gemmini supports two data-flows, weight stationary and output stationary, with a choice of either having one of them or both. The PEs also support different bit-widths for input, outputs and internal buffer.

Gemmini has a banked scratchpad made of SRAMs that store the inputs, weights and output fmaps(feature maps). A DMA controller is present to take care of data transfer to and from the host processor to the scratchpad. A Dependency

Mgmt unit is present to take care of dependencies between the instructions. A higher bit-witdh accumulator buffer made of SRAMs is present to store the result of the matrix multiplication. Adders are present in inputs of accumulators to perform partial sum accumulations.

Additional hardware is present to support non-linear activation functions such as ReLU, ReLU6, hardware to scale the outputs to lower bit-width to feed into the next layer. Peripheral circuits are present to perform transpose of the weight matrix which is needed for output stationary data-flow.

3.2.2 Accelerator interfacing with the processor:

The accelerator is integrated with the Rocket System-on-chip generator, which can be configured to interface with either the Rocket in-order core or the BOOM out-of-order core. The accelerator communicated with the host processor through the Rocket Co-processor (RoCC) interface, which enables the host to send the accelerator a stream of custom instructions. The Gemmini ISA provides three instructions:

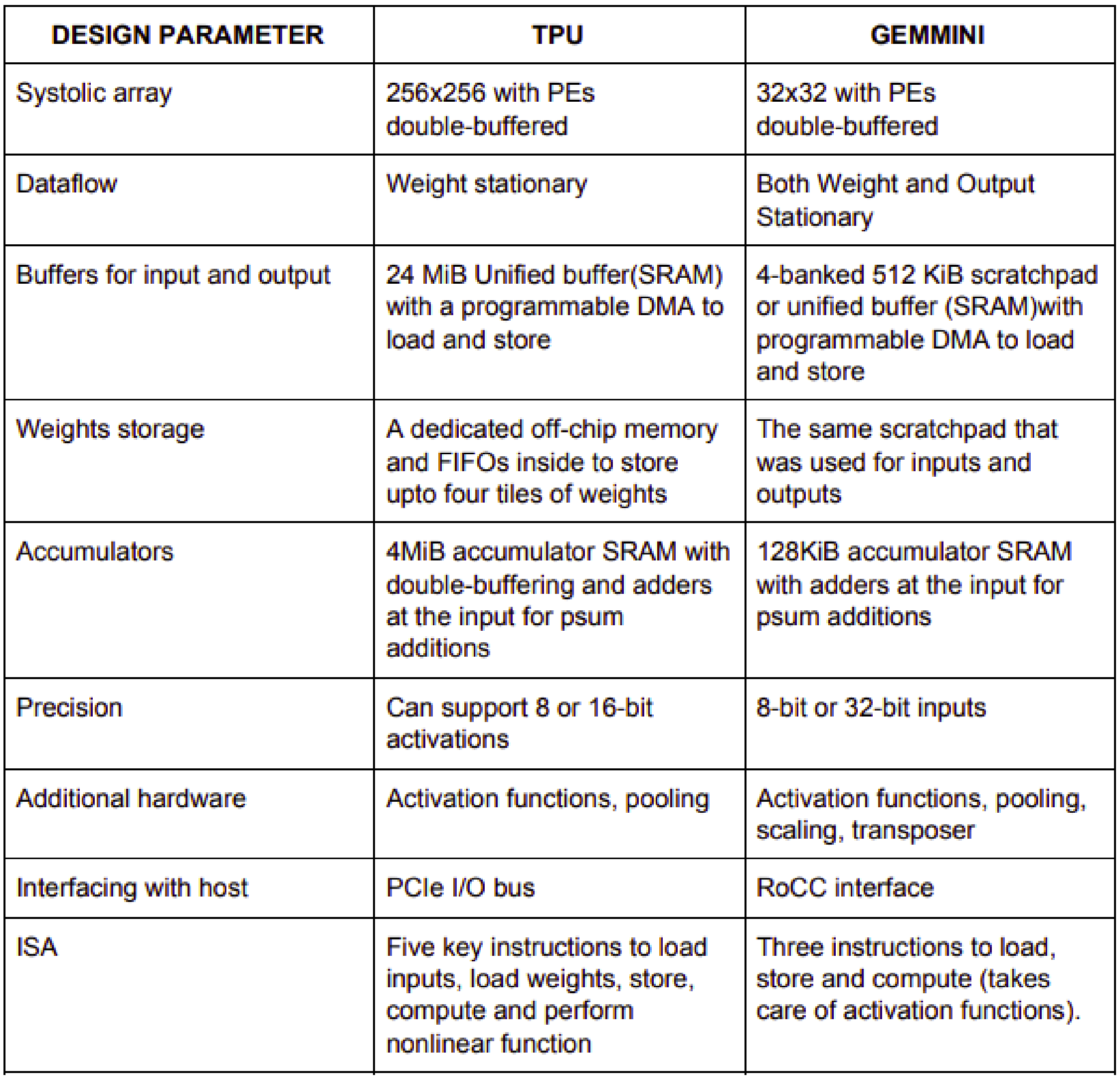
- **mvin** instruction to load the data from the main memory to the accelerator's scratchpad.
- **mvout** instruction stores the data from the scratchpad or the accumulator to the main memory.
- **compute** instruction configures the parameters like activation, dataflow and starts with the systolic operation and stores the output in the accumulator or the scratchpad.

3.2.3 Software:

To make it easier for the programmers to use Gemmini accelerator, a software library has been provided that implements hand-tuned, tiled GEMM functions such as matrix multiplication of any size, multi layer perceptron (MLP), CNNs, non-linear activation and quantizations. Tiling is performed along the size of the systolic array and the accelerator scratchpad. These tiling parameters are generated by the Chisel generator and are included as a header file in software libraries.

3.3 Summary

In this section, we will summarise the design choices made by both the accelerators. Since Gemmini doesn't have fixed numbers for the parameters, we will choose a design point they chose for their physical design for this comparison.



DESIGN PARAMETER	TPU	GEMMINI
Systolic array	256x256 with PEs double-buffered	32x32 with PEs double-buffered
Dataflow	Weight stationary	Both Weight and Output Stationary
Buffers for input and output	24 MiB Unified buffer(SRAM) with a programmable DMA to load and store	4-banked 512 KiB scratchpad or unified buffer (SRAM)with programmable DMA to load and store
Weights storage	A dedicated off-chip memory and FIFOs inside to store upto four tiles of weights	The same scratchpad that was used for inputs and outputs
Accumulators	4MiB accumulator SRAM with double-buffering and adders at the input for psum additions	128KiB accumulator SRAM with adders at the input for psum additions
Precision	Can support 8 or 16-bit activations	8-bit or 32-bit inputs
Additional hardware	Activation functions, pooling	Activation functions, pooling, scaling, transposer
Interfacing with host	PCIe I/O bus	RoCC interface
ISA	Five key instructions to load inputs, load weights, store, compute and perform nonlinear function	Three instructions to load, store and compute (takes care of activation functions).

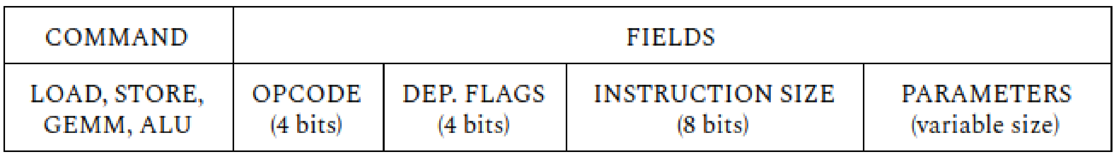
Figure 3.3: Comparing TPU and Gemmini based on few design parameters

CHAPTER 4

MICRO ARCHITECTURE

In this chapter we will look at the overview of the instruction set and the micro-architecture of our accelerator.

4.1 Overview of Instruction set



COMMAND	FIELDS			
LOAD, STORE, GEMM, ALU	OPCODE (4 bits)	DEP. FLAGS (4 bits)	INSTRUCTION SIZE (8 bits)	PARAMETERS (variable size)

Figure 4.1: Instruction encoding

We have four CISC instructions to execute tasks in our accelerator. The general instruction encoding is shown in Figure 4.1. The opcode is used by the decode logic to send the instructions to respective queues. The dependency flags are set by the compiler to let the hardware know the dependencies between the modules. The instruction size denotes the size of the parameters for that particular instruction. The parameters will vary from 16-18 bytes depending on the instruction. The four instructions in the ISA are as follows:

- **LOAD**: Loads data from DRAM to the on-chip buffer (SRAM).
- **GEMM**: Reads input maps and weights from the buffer, performs im2col on the inputs, computes on the systolic array and stores the outputs in the output buffer.
- **ALU**: Performs nonlinear activation functions and pooling operation
- **STORE**: Stores the outputs from output buffer to DRAM

4.2 System architecture

The accelerator has five modules: frontend module, load module, compute module (systolic-array), tensor ALU and store module. The on-chip buffers are present in the buffer module. More about the buffer module will be discussed in Chapter 4. Figure 4.2 gives the system overview of the accelerator. This accelerator will act as co-processor and interface with the Shakti C-Class processor. We will briefly look at each module in the next sections.

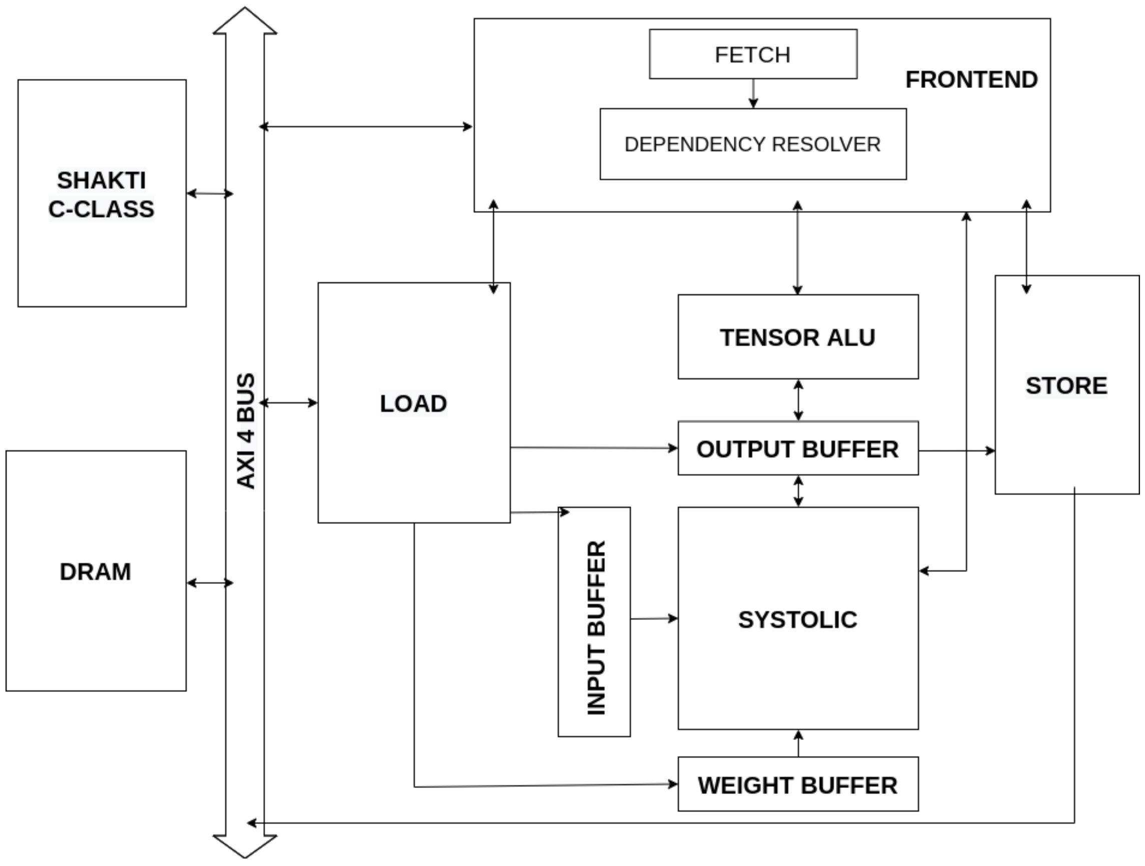


Figure 4.2: System overview of the accelerator

4.3 Frontend

4.3.1 Fetch and Decode:

The fetch module contains an AXI4 slave interface using which the core programs the program counter (PC) and set bit. The fetch module sends axi requests to read the accelerator instructions from the memory. Based on the instruction size another request is sent to fetch the parameters for the corresponding instruction. The decode logic then sends the instruction to the appropriate queue based on the opcode.

4.3.2 Dependency resolver:

This module takes care of dependencies between instructions. The module has queues for each instruction to which the decode logic pushes the instruction into. The instruction has four flags, push_prev, push_next, pop_prev, pop_next. Each module has a previous module and a next module as shown in Table 4.1.

Module	Previous	Next
Load	-	GEMM
Store	ALU	-
GEMM	Load	ALU
ALU	GEMM	Store

Table 4.1: Dependencies for each module

When a push flag is set in the instruction, the corresponding module pushes a token once the execution is complete, to another module, depending on whether push_next or push_prev is set. When a pop_prev flag is set for a module, its previous module will have the push_next flag set, so once the previous module pushes the token, the dependency resolver module pops that token and sends the instruction to the current module. Each module has a FIFO through which the dependency resolver sends the instruction(once dependencies are resolved).

4.4 Load

This module loads the feature maps and weights from the DRAM to the on-chip memory. This module along with store is what is represented as DMA engine(in Gemmini) or DMA controller(in TPU). It has an AXI4 master interface using which read requests are sent to the DRAM. The module contains an address generator logic which calculates the read and the write address for every read request based on the parameters like DRAM base address, SRAM base address, strides and sizes in different dimensions. The instruction also has a reset bit which when set, loads an immediate value instead of loading from the DRAM. Based on the SRAM base address the module decides to which buffer the data should be loaded. This module will be covered in detail in the next chapter.

4.5 GEMM/Compute

This module contains the systolic array and performs the convolution operation. The instruction contains the base address of inputs, weights and outputs. The weights are first pre-loaded into the array by reading it from the weight buffer, then the inputs are fed systolically by reading from the input buffer. For the partial sum accumulation, the partial sums can be pre-loaded into the PEs if needed or zeros will be filled, this will be decided based on a flag in the instruction. The outputs are written into the output buffer, which can be either used by the systolic array for next GEMM or read by the ALU to perform activation function.

4.6 ALU

The ALU module performs operations other than convolution, mainly non-linear activation functions such as ReLU and pooling operations. ALU units are replicated to create a vector ALU that can operate on multiple data. The module fetches its inputs from the output buffer based on the address, sizes and strides and writes the output after computing back to the output buffer.

4.7 Store

This module stores the outputs from the output buffer to the DRAM. Similar to the load module, this module contains an AXI4 master interface through which the data is sent to the DRAM. The module takes parameters such as DRAM address, SRAM address, sizes and strides to store the 3D slice to the DRAM.

CHAPTER 5

LOAD AND BUFFER MODULE

In this chapter we will see about two key modules of the micro-architecture. One is the load module and other is the buffer module. Under load module we will discuss in detail what happens when the module receives the load instruction and how the module loads a 3D slice from the DRAM and stores it in the SRAM. In the Buffer module, we will see about the buffers present for the data storage in the accelerator and the need for double buffering.

5.1 Load Module

5.1.1 Overview:

The task of the load module is to load the input feature maps, weights and outputs from the DRAM into the on-chip memory. This module works similar to DMA engine(Gemmini) or the DMA controller(TPU) by accessing the main memory directly with minimal intervention from the processor. The 4D feature maps are stored in the NHWC format in the DRAM and the weights are stored in the RSCM format. The feature maps and weights are stored in this format by the software. The feature maps and the weights are loaded by reading one 3D slice per load instruction. The NHWC format (RSCM for weights) is retained while storing it in the on-chip memory. The 3D slice that will be loaded can be discontinuous in DRAM, but it will be stored continuously in SRAM. There is an option to store immediate values instead of loading the values from DRAM. The parameters needed for the LOAD instruction are:

- DRAM base address/ immediate constant value
- SRAM base address
- Z_SIZE - Number of channels in the 3D slice (number of filters in case of weights)

- Z_STRIDE - Offset from start of one column to access the next column
- Y_SIZE - Number of columns to access to complete one 2D slice (number of channels in case of weights)
- Y_STRIDE - Offset from start of 2D slice to the next 2d slice
- X_SIZE - Number of rows to access to complete one 3D slice (number of pixels in unrolled RxS, in case of weights)
- RESET - If this bit is set, do not generate any memory request. Every location in the SRAM to which the 3D slice will be written to is simply written with the immediate constant value.

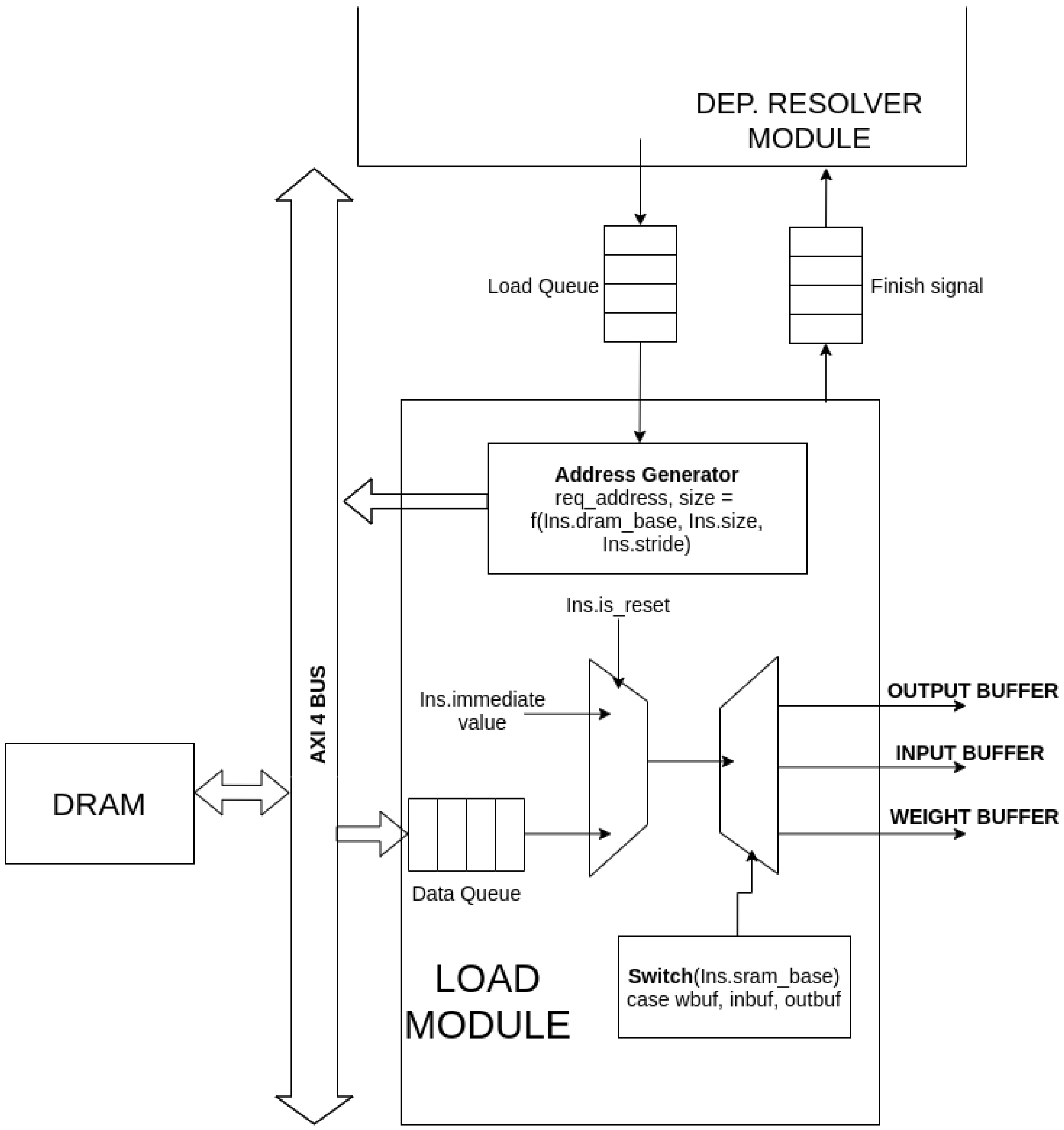


Figure 5.1: Block diagram of the load module

Figure 5.1 shows the block diagram of the load module. The dependency resolver checks for dependencies for a particular load instruction, once resolved, pushes the instruction into the load queue. The load module then pops the instruction from the queue and sends it to the address generator. The address generator block contains an AXI4 master interface using which read requests to the DRAM are sent. This AXI4 master acts similar to a DMA controller where it can send read requests to the main memory with minimum intervention from the processor core. This logic generates the required read address and size of data to be loaded using the DRAM base address, different sizes and strides in the instruction and sends an axi request. We will learn more about this logic later in this section.

The reset bit in the instruction is used to decide whether we have to load from the DRAM or just write the immediate value which is represented as a multiplexer in the block diagram. Once the data is received by the load module, the module sends the data, index and bank to the corresponding buffer with a valid bit. The buffer to which the load should be done is decided based on the SRAM base address present in the instruction. This is represented as a de-mux in the block diagram. Once the load instruction is completed, the module sends a finish signal to the dependency resolver module. The load module was coded in Bluespec HDL and basic verification was done using a testbench. Code below shows the interface of the load module written in Bluespec HDL.

Listing 5.1: Interface of Load module in Bluespec HDL

```

interface Ifc_load_Module#(numeric type addr_width, numeric type
    data_width);
      interface AXI4_Master_IFC#(addr_width, data_width,0) master;
      //to get parameters from the dependency module
      interface Put#(Bit#(128)) subifc_get_loadparams;
      interface Get#(Bool) subifc_send_loadfinish;
      //methods to send write requests to buffers
      method Vector#(TDiv#(data_width,`INWIDTH),Tuple4#(Bool,Bit#(`
    IBUF_INDEX),Bit#(`IBUF_Bankbits),Bit#(`INWIDTH)))) ibuf_wr_data;
      method Vector#(TDiv#(data_width,`INWIDTH), Tuple4#(Bool,Bit#(`
    WBUF_INDEX),Bit#(`WBUF_Bankbits), Bit#(`INWIDTH)))) wbuf_wr_data;
      method Vector#(TDiv#(data_width,`OUTWIDTH),Tuple4#(Bool,Bit#(`
    OBUF_INDEX),Bit#(`OBUF_Bankbits), Bit#(`OUTWIDTH)))) obuf_wr_data;
```

```

endinterface
```

5.1.2 Loading a 3D slice - example:

We will now see through an example, how a 3D slice of a feature map is loaded from the DRAM, what exactly does the sizes and strides in the instruction parameters map to.

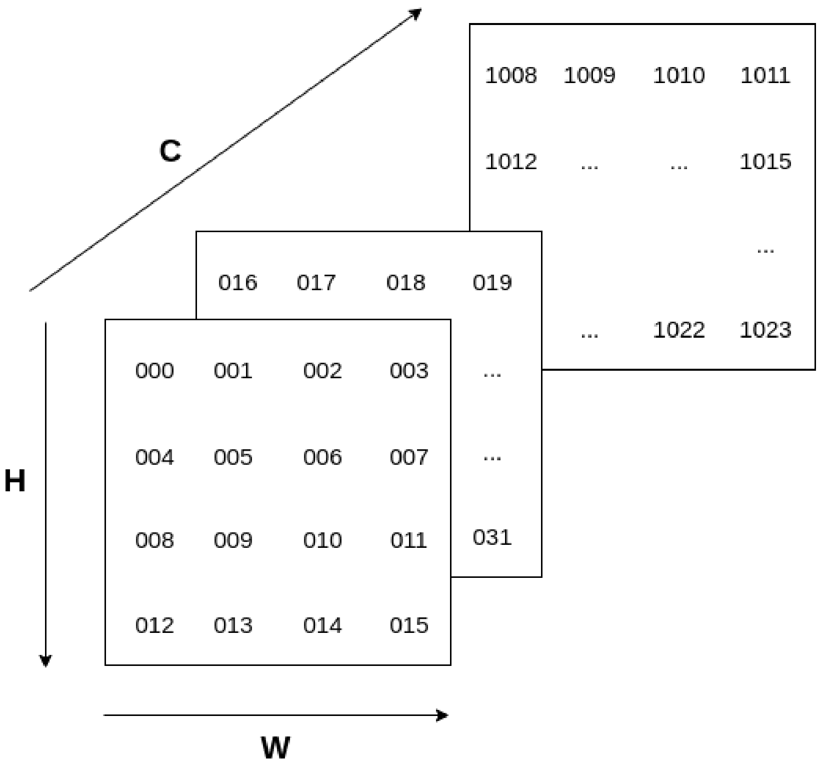


Figure 5.2: An example 1x64x4x4 feature map

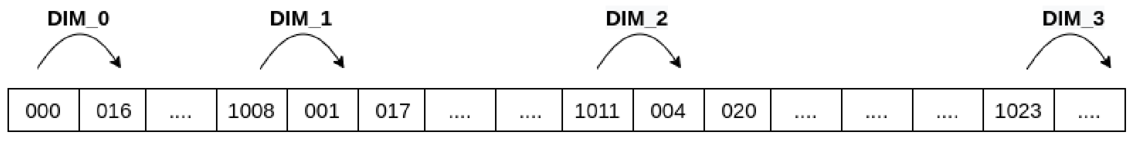


Figure 5.3: Layout in the DRAM in NHWC format

Let us consider a feature map example to be loaded as in Figure 5.2 with the dimensions of 1 x 64 x 4 x 4. This feature map will be stored in the NHWC format in the DRAM as shown in Figure 5.3. DIM_0 represents the innermost dimension channels(C), DIM_1 represents width(W), DIM_2 represents height(H) and DIM_3 represents batch(N).

Let us load a 3D slice of dimensions 1 x 2 x 2 x 32 from the original feature map. The load parameters for this slice would be:

- Z_size - no. of channels which will be **32** for this example.
- Z_stride - Offset from one column to another which will be the number of channels of the actual feature map i.e., **64**, independent of the 3D slice to be loaded
- Y_size - no. of columns to complete one 2D slice – **2** for this example
- Y_stride - Offset from start of one 2D slice to next which will be equal to W*C of the actual feature map i.e., 4*64 = **256**, independent of the 3D slice to be loaded
- X_size - no.of 2D slices to be loaded – **2** for this example

Since the module loads a 3D slice from the actual 3D fmap, the data will not be continuous in the DRAM. So the AXI4 master cannot load the 3D slice from a single read request. A number of read requests are being sent by the AXI master to load the 3D slice completely. We will load Z_SIZE number of values per request since they are continuous in the memory. The address generator logic calculates the DRAM address, SRAM address for every read request. Each set of Z_SIZE elements are stored in the SRAM in a continuous manner. The pseudo-code below gives the logic for the address generation module.

```

for i = 0 to X_SIZE: // Nested loop unrolled in time
  for j = 0 to Y_SIZE:
    dram_addr = dram_base + j * Z_STRIDE + i * Y_STRIDE
    sram_addr = sram_base + j * Z_SIZE + i * Z_SIZE * Y_SIZE
    //Z_SIZE loaded per axi request
    load(dram_addr, Z_SIZE, sram_addr)

```

Listing 5.2: Psuedo-code for address generation logic

5.2 Buffer module

This module contains all the on-chip memory made of SRAM required for data storage inside the accelerator. The current design of our accelerator has three buffers, input buffer, weight buffer and the output buffer. The number of banks for the buffer is decided based on the systolic array dimensions. The systolic array reads number of inputs per cycle equal to its number of rows, so the number of banks of the input buffer should be at least equal to the number of rows of the array. Similarly for weights and output buffers, systolic reads weights equal to its number of columns and write partial outputs equal to number of its columns into the output buffer. This means the minimum number of banks needed in weights and output buffers will be equal to the number of columns in the systolic array. A buffer for storing the parameters was added initially but is removed now(discussed in section 6.2). The sizes of each buffer will be determined based on the compute capacity of the accelerator and memory bandwidth available. Detailed design space exploration for the buffers has been done in the next chapter, section 6.1.

5.2.1 Double buffering:

This is one of the important optimizations that can be done to improve throughput. In case of load and compute instruction, the load module loads the input buffer while the systolic array reads from it. Loading from main memory can take several cycles and the systolic has to wait till the load has been completed, this would affect the overall latency of the inference. One way to improve this is to try to overlap these tasks so that they can run in parallel. This is done by replicating the buffers so that each module can use one buffer at a time and swap in the next cycle of instruction (also popularly known as ping pong buffer). When buffers have high contention on its ports, it forces us to execute tasks serially thus increasing the latency and reducing the utilization of hardware. In section 6.1 we will see how double buffering, in particular for the output buffer, can help execute tasks in parallel. Double buffering can either be implemented in hardware or by the compiler after exposing the address space, For the initial version of the design, double buffering was implemented in hardware (coded in Bluespec HDL).

CHAPTER 6

DESIGN SPACE EXPLORATION

6.1 Data storage in the accelerator

As seen in section 5.2, the buffer module consists of three buffers each for input, weight and output. Deciding the amount of buffers and the size is crucial for an accelerator. Having limited on-chip memory can create high contention on the buffers thus forcing us to serialize operations and see reduction in performance whereas having excess memory can lead to increased area and cost. In this section we will analyse two design choices on the buffers.

6.1.1 Having separate buffers for input, weight and output:

Buffer	Load	Store	GEMM	ALU
Input Buffer	Write	-	Read	-
Output Buffer	Write	Read	Read and Write	Read and Write
Weight Buffer	Write	-	Read	-

Table 6.1: Overview of Buffers and how different modules use them

For the input buffer, the load module writes input fmaps into it and the compute module reads the data from it. Having a buffer with simultaneous read/write can ensure loading input fmaps and compute operations executing in parallel, thus giving us the benefits of double buffering.

Similarly for the weights buffer, the load module writes weights into it and the compute module reads from the buffer. Parallel execution can be ensured with simultaneous read/write buffer.

From the Table 6.1 it is clear that the output buffer has the most contention in its ports since all four modules (load, alu, compute, store) try to access the buffer. The tasks leading to these contentions are as follows:

- The load module writes the partial sums into the buffer – write.
- The compute module reads the partial sums from the buffer and stores it in the PEs before starting the systolic operation – read.
- The compute module then writes the partial results into the buffer every cycle till the systolic operation is completed – write.
- The ALU then reads the outputs from the buffer, performs operations like activation functions, pooling (if required) and then writes it back to the buffer – read and write.
- The store module then reads from the output buffer and stores it into DRAM – read.

Since we have separate queues for all these instructions we can execute all these operations in parallel if they do not have any dependencies. But since the output buffer can do only one read and write at the same time, there is contention in the ports of the output buffer forcing us to execute these instructions sequentially.

This is where having multiple buffers (double or triple buffering) can help. To solve the contention due to the ALU and systolic array reading the partial sums, we can go with double buffering. Along with the output buffer we can have another accumulator buffer/scratchpad made of SRAM which will be used by the compute module to cycle the partial sums by writing and reading it again.

One more additional optimization is double buffering the accumulator buffer into Tensor ALU scratchpad and Systolic scratchpad. With this setup, the compute module will write the partial sums into this Systolic scratchpad which will be read back by the compute module again. The ALU will read the output from the Tensor ALU scratchpad, perform activation functions and pooling if required and write it to the unified buffer from which the store operation will write it to the DRAM. This can ensure the compute operation and ALU executing in parallel thus giving the maximum parallelism.

Having three buffers for outputs, one each for compute, ALU and store gives us the ability to perform three reads and three writes at the same time, using which we can achieve parallel execution.

6.1.2 Shared Buffer:

From the previous design choice we had to have five buffers. Another design choice is to see if we can have a unified or global buffer with the number of banks being thrice the number of rows/columns of the systolic array. One idea is to combine input, weights and output buffers as a single unified buffer and have two scratchpads each for Tensor ALU and Systolic array.

Two key advantages with this setup are as follows:

In separate buffers, the output fmaps once computed is stored in the output buffer which is then stored back to the DRAM by the store instruction. When the next convolution layer starts, the output fmap of the previous layer is loaded back again into the input buffer and is read by the systolic array. With the shared buffer, we can have the output fmaps of the current layer in the shared buffer itself which can be directly fed as inputs for the next layer. This saves us on the latency and energy consumption of additional load and store operation. However the whole output fmap of a layer cannot be stored in the buffer due to restrictions in the size of the buffer, so this can be done for a part of fmap and rest being stored in DRAM and loaded back as in the previous setup.

Another advantage is when we consider a compute fold, there will be very less number of load input instructions when compared to the number of compute instructions. Due to this, with a separate buffer for inputs, the write port of the input buffer is under utilized. Whereas in the shared buffer case this will lead to overall reduced contention in the ports.

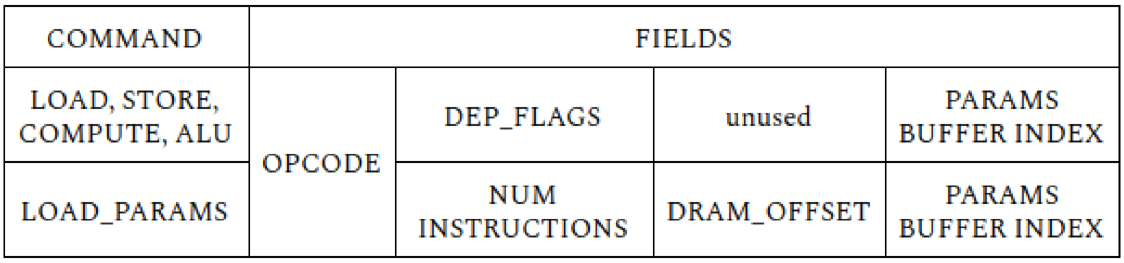
Recalling the design choices on on-chip memory of Gemmini and TPU from the literature survey in Chapter 2: TPU has an unified buffer for inputs and outputs, a dedicated off-chip memory and FIFOs inside the accelerator for weights. TPU also has a dedicated accumulator SRAM which are double buffered. Gemmini has a unified buffer for inputs, weights and outputs and a single accumulator SRAM.

6.2 Parameters for the instruction

As we have seen in Chapter 3, every instruction or module will need a set of parameters to execute their tasks. The size of these parameters vary from 16-18 bytes depending on the instruction. One design choice is to go with the CISC type instruction with parameters within the instruction itself and other is having a RISC instruction without parameters. In this section we will look at the trade-off analysis between these two choices.

6.2.1 RISC:

The idea here is to have instructions with small size (around 8 bytes) without parameters in it. For loading the parameters we will have a separate instruction called LOAD_PARAMS. The instruction encoding for this setup is shown in Figure 6.1.



COMMAND	FIELDS			
LOAD, STORE, COMPUTE, ALU	OPCODE	DEP_FLAGS	unused	PARAMS BUFFER INDEX
LOAD_PARAMS		NUM INSTRUCTIONS	DRAM_OFFSET	PARAMS BUFFER INDEX

Figure 6.1: Instruction encoding in RISC setup

With a single load_params instruction, parameters for next 12-16 instructions will be loaded. A dedicated buffer called param_buffer is added to store the parameters. The param_buffer is a 4-banked 1KiB SRAM. This load_params instruction will be treated like just another load instruction and will be sent to the load module by the dependency resolver module. The load module then loads the parameters from the DRAM into the parameter buffer.

As seen in Chapter 3, there are FIFOs present from the dependency resolver module to every other module to send the instruction or the parameters. In this setup, the dependency resolver module sends only the address of the param_buffer to the modules. For the load module, it sends the whole instruction to perform load_params operation. The other modules will fetch the parameters from this buffer, using the param_buffer index received from the FIFO, before executing their tasks.

6.2.2 CISC:

The other option is the CISC instruction type which is implemented in the current version of the design and has been discussed in Chapter 2. The instruction encoding is as shown in Figure 4.1. We now have all the parameters as part of the instruction itself. The dependency resolver module sends the parameters through FIFOs to every module.

6.2.3 RISC vs CISC:

In CISC, the fetch module has to fetch 16-20 bytes for every instruction as compared to 8 bytes for the RISC model. This reduces the instruction fetch rate for the CISC setup. In RISC, having a load_param instruction increases the workload for the load module. The module loads parameters of size around 256 bytes per load_param instruction.

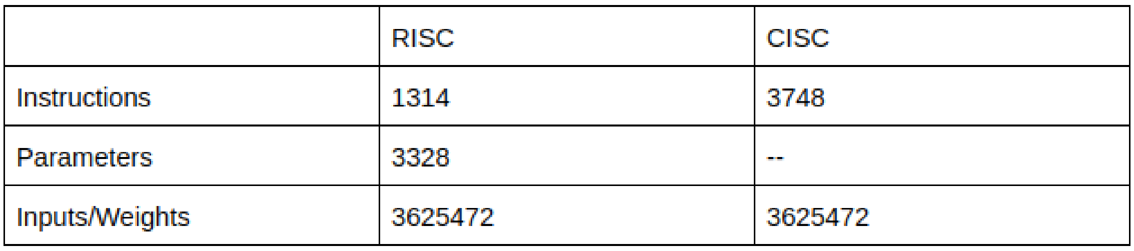
In terms of on-chip memory requirements, In RISC setup we will need a param_buffer of size 1KiB which will not be required in CISC. In terms of FIFOs present between the dependency resolver module and the other modules, CISC setup will need FIFOs of bigger bit width, reason being, we have to send parameters which will need 16-18 bytes of bit width for each instruction. In RISC setup, only param_buffer index is sent to the modules(except for load module), which means required bit width is less than 4 bytes per FIFO entry. The total size of FIFOs can be determined only after we know the number of entries in each FIFO which is not decided yet.

To see the difference of number of bytes loaded from the memory in both cases, we will evaluate them for computing a sample convolution layer. Consider the following parameters for the systolic array and the convolution layer:

- Array dimensions: 64 x 64
- Input dimensions: 1 x 512 x 13 x 13

- Weight dimensions: 384 x 512 x 3 x 3
- Output dimensions: 1 x 384 x 13 x 13
- Input/weights width: 1 byte
- Stride: 1
- Padding: 1
- Parameters size: 16 (It will vary little for each instruction but assumed to be constant(16) for now)
- Instruction size for RISC: 6 bytes
- Instruction size for CISC: 18 bytes

The instruction trace for this layer had 206 instructions. A Load_param instruction for every 16 instructions is added in the RISC setup. The comparison for number of bytes loaded from memory in both the setup is as follows:



	RISC	CISC
Instructions	1314	3748
Parameters	3328	--
Inputs/Weights	3625472	3625472

Figure 6.2: Comparing RISC and CISC on Number of bytes loaded from memory

In RISC setup we had to load **934** bytes more than in CISC for this particular convolution layer.

CHAPTER 7

CONCLUSION

To summarise the contributions of this work, the load module was designed and implemented in different versions of the design. Preliminary verification of the load module was done using testbench. Future work here will be to formally verify and perform the synthesis to estimate the hardware requirements. Double buffering was implemented in the initial version of the design for the input buffer. All implementations were done using Bluespec HDL. In the buffer module, currently there are three buffers. We have done theoretical trade-off analysis for the on-chip memory in the accelerator in Chapter 5. Future work would be to add scratchpads for Tensor ALU and Compute modules and implement double buffering either in hardware or by the compiler. Also simulations can be performed using simulators to estimate the optimal choice for the on-chip memory. Another trade-off analysis was made between RISC and CISC type ISA to decide on how the parameters for the instructions should be loaded and stored in the accelerator.

REFERENCES

[1] **Gala, N., A. Menon, R. Bodduna, G. S. Madhusudan, and V. Kamakoti**, "Shakti processors: An open-source hardware initiative," 2017.

[2] **Kung**, "Why systolic architectures?", 1982.

[3] **Clement Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun**, "NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision", 2011.

[4] **Chen, T., Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam,** "Diannao: A small-footprint high-throughput accelerator for ubiquitous machinelearning.", 2014.

[5] **Chen, Y., T. Krishna, J. S. Emer, and V. Sze,** "Eyeriss: An energy efficient reconfigurable accelerator for deep convolutional neural networks", 2017.

[6] **Jouppi, N. P., C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon**, " In-datacenter performance analysis of a tensor processing unit", 2017.

[7] **Parashar, A., M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally**, "SCNN: An accelerator for compressed-sparse convolutional neural networks," 2017

[8] **Han, S., X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally**, "EIE: Efficient inference engine on compressed deep neural network," 2016.

[9] **Genc, H., A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, Y. S. Shao, B. Nikolic, I. Stoica, and K. Asanovic**, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures", 2019.