

ABSTRACT

BAKER, LEE B. Design of a 3DIC system to aid in the acceleration of embedded systems that employ multiple instances of disparate artificial neural networks. (Under the direction of Paul Franzon.)

This work demonstrates how a customized Three-dimensional dynamic random-access memory (3D-DRAM) with a very wide databus can be combined with application-specific layers to produce a system meeting the requirements of embedded systems employing multiple instances of disparate Artificial Neural Networks (ANNs). This work avoids any dependencies on static random-access memory (SRAM) that might limit the size of supported ANNs. This work demonstrates the required utilization of the very wide dynamic random-access memory (DRAM) databus when employing instructions and data structures that facilitate operating directly out of the 3D-DRAM. By allowing system functions to operate asynchronously this work is able to absorb the latencies associated with DRAM and provide the bandwidth required to support multiple useful-sized disparate ANNs. By demonstrating effective use of the very wide bus of a customized 3D-DRAM, this system demonstrates a 3X power improvement and 6X area improvement over similar ANN systems.

© Copyright 2018 by Lee B. Baker

All Rights Reserved

Design of a 3DIC system to aid in the acceleration of embedded systems that
employ multiple instances of disparate artificial neural networks

by
Lee B. Baker

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Electrical Engineering

Raleigh, North Carolina

2018

APPROVED BY:

Winser Alexander

Gregory Byrd

Richard Warr

Paul Franzon
Chair of Advisory Committee

DEDICATION

To my wife Mandy, my children Adam, Rachel and Paul and my parents Joan and Barry.

BIOGRAPHY

The author was born in the United Kingdom. After high school he took a job in a local electronic engineering firm under a vocational program. While working on the manufacturing floor and seeing the white coated "engineers" being called down from upstairs to solve the "big" problems, he decided he wanted to wear one of those white coats. The journey to the "white coat" took him to Brighton Polytechnic, now Brighton University and a First Class Honors Degree in Electrical Engineering. After working in the UK for a couple of years, he moved to the United States. The journey included a family with a daughter and two sons. The education continued with a Masters in Engineering from Villanova University and a Masters in Business Administration from North Carolina State University.

With the family now being somewhat independent, he decided to make a career change that would hopefully include teaching.

That career change included enrolling in the Electrical Engineering PhD program at North Carolina State University. This stage of the education journey has resulted in this dissertation.

Remember:

"do not stand still."

"do not let your past dictate your future."

ACKNOWLEDGEMENTS

At a personal level, I would like to thank my wife Mandy and my children Adam, Rachel and Paul for their encouragement.

I would like to thank my advisor, Paul Franzon for his help in making this possible.

I would also like to thank Steve Lipa, Jong Beom Park and Josh Schabel for their feedback on this dissertation.

I would also like to thank my fellow students, especially Jong Beom, Josh, Sumon and Weifu for their healthy discussions and, being an older student, referring to me as Lee and not Sir or Mr. Baker.

This work was funded in part by DARPA and AFRL under FA8650-15-1-7518 and DARPA and ONR under N00014-17-1-3013, as part of the CHIPS program.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	ix
Chapter 1 Introduction	1
1.1 Overview	1
1.2 Abbreviations	3
1.3 Artificial Neural Networks	7
1.3.1 ANN Overview	8
1.3.2 ANN Layers	13
1.3.2.1 Deep Neural Networks	14
1.3.2.2 Feature Layers	14
1.3.3 ANN Processing	19
1.4 Motivation	23
1.4.1 The Problem	23
1.4.1.1 Why not SRAM?	23
1.4.2 Alternatives	24
1.4.2.1 Graphics processing units (GPUs)	24
1.4.2.2 Application-Specific Integrated Circuits (ASICs)/Application-Specific Instruction-set Processors (ASIPs)	25
1.5 Three-Dimensional Integrated Circuits (3DICs)	26
1.5.1 Pros and Cons	26
1.5.2 Construction	29
1.5.3 Design Guidelines	31
1.6 DRAM Overview and Customizations	33
1.6.1 Accessing DRAM and SRAM	34
1.6.1.1 Access Locality/Reuse and SRAM as a Cache	35
1.6.2 DRAM Customizations	37
1.6.2.1 Customization One: Very-Wide Bus	37
1.6.2.2 Customization Two: Write Mask	38
1.7 The Solution	40
1.8 Novelty	41
1.9 Summary	41
Chapter 2 State-of-the-art	43
Chapter 3 System Overview	46
3.1 Sub-System Column	48
3.2 Processing a group of Artificial Neurons (ANes)	49
3.3 Processing a single ANe	50
3.4 Sub-System Column (SSC) Blocks	50
3.4.1 Customized DRAM : Dis-Integrated 3D DRAM (DiRAM4)	50

3.4.2	Layer Interconnect	52
3.4.3	Stack Bus	52
3.4.3.1	Common Bus Signaling	53
3.4.4	DRAM Bus	54
3.4.5	Manager Layer	54
3.4.6	Processing Engine Layer	58
3.4.7	Inter-Manager Communication	59
3.4.7.1	Network-on-Chip (NoC) Future Work	62
3.5	Summary	63
Chapter 4	System Operations	65
4.1	Instructions	66
4.1.1	Instruction Types	67
4.1.2	Compute Instruction	67
4.1.2.1	Accessing of Pre-synaptic ANe states and connection weights	69
4.1.2.2	Storage Descriptor	72
4.1.2.3	Writing ANe state results to memory	72
4.1.3	Configuration Instruction	73
4.1.3.1	Configuration Data Instruction	75
4.1.3.2	Configuration Sync Instruction	75
4.1.4	Multiple Instruction Functions	75
4.2	Host Instructions	77
Chapter 5	Detailed System Description	81
5.1	Manager	81
5.1.1	System controller	81
5.1.1.1	Initial Boot	82
5.1.2	Instruction Decoder	84
5.1.2.1	Compute Instructions	84
5.1.2.2	Configuration Instructions	86
5.1.2.2.1	Sync Send	86
5.1.2.2.2	Sync Wait	87
5.1.2.2.3	Sync Pause	87
5.1.2.2.4	Sync Flush	88
5.1.2.2.5	Instruction download	88
5.1.2.2.6	Sync group table download	89
5.1.2.2.7	Memory download	89
5.1.2.2.8	Memory upload	90
5.1.3	Main memory controller (MMC)	90
5.1.4	Memory read controller (MRC)	91
5.1.5	Return data processor (RDP)	93
5.1.6	Memory write controller (MWC)	94
5.2	Processing Engine	95
5.2.1	Configuration	95

5.2.2	PE Controller	97
5.2.3	Streaming Operations	98
5.2.4	SIMD	98
5.2.5	Special function units (SFUs)	99
5.2.6	DMA Controller and Local memory	100
5.2.7	Upstream controller	100
Chapter 6	Results	105
6.1	Power and Area Scaling estimates	106
6.2	Physical Placement	107
6.3	Synthesis	108
6.4	Logic Verification	108
6.5	Power Estimates	110
6.6	Summary	110
6.6.1	Recent state-of-the-art comparison	112
Chapter 7	Conclusions and Future Work	115
7.1	Conclusions	115
7.2	Future Work	116
BIBLIOGRAPHY		118
APPENDICES		122
Appendix A	Softmax Implementation	123
Appendix B	Max pooling Implementation	126
Appendix C	DiRAM4 Characteristics	128

LIST OF TABLES

Table 1.2	Estimated system bandwidth and storage design requirements	21
Table 1.1	Baseline ANN layer configuration [27]	22
Table 1.3	TSV design characteristics	32
Table 3.1	NoC header cycle fields	61
Table 3.2	NoC option/data cycle fields	61
Table 4.1	Option tuple functions	69
Table 5.1	stOp pointer memory fields	97
Table 5.2	Streaming operation fields	101
Table 5.3	SIMD wrapper special function codes	102
Table 6.1	Example design area and power	106
Table 6.2	68 nm to 28 nm scaling numbers	106
Table 6.3	Area contribution	107
Table 6.4	Fanin bandwidth tests	109
Table 6.5	Baseline ANN expected bandwidth	109
Table 6.6	Power Estimates	110
Table 6.7	System features implemented	111
Table C.1	DiRAM4 characteristics [14]	129

LIST OF FIGURES

Figure 1.1	Artist's impression of a mammalian neuron [44]	8
Figure 1.2	Artificial Neural Network	10
Figure 1.3	Example Rated-Based Model Activation functions	10
Figure 1.4	Example Spiking Activation Function Model	11
Figure 1.5	Layered Artificial Neural Networks	12
Figure 1.6	Classification using ANNs layers [6]	13
Figure 1.7	Deep network showing feature layers	14
Figure 1.8	Locally Connected Layer to Layer Connection Types	16
Figure 1.9	Fully Connected Layer to Layer Connection Types	17
Figure 1.10	Features and locally-connected filters (kernels)	18
Figure 1.11	Single Layer constructed from 3D layers of Features	19
Figure 1.12	Baseline DNN showing layer order [27]	22
Figure 1.13	3DIC Stack of Die	27
Figure 1.14	Die Stack profile [20] with Through-Silicon Vias (TSVs)	30
Figure 1.15	Typical Memory Block Diagram [22]	33
Figure 1.16	RAM Storage Cell Types	34
Figure 1.17	Typical DRAM Block Diagram [12]	38
Figure 1.18	Exposing more of the DRAM page	38
Figure 3.1	3DIC System Stack	47
Figure 3.2	Sub-System Column (SSC)	48
Figure 3.3	Multiplexing DRAM data to execution lanes	51
Figure 3.4	DRAM Physical interface layout showing area for SSC	52
Figure 3.5	Stack Bus signaling	55
Figure 3.6	Read and Write request to DiRAM4 [14]	56
Figure 3.7	Worst case PO/PC sequence	56
Figure 3.8	DRAM Read Path Buffering	57
Figure 3.9	Four descriptor instruction (4-tuple)	57
Figure 3.10	Processing Engine (PE) ANe calculation	59
Figure 3.11	NoC manager connectivity	60
Figure 3.12	NoC packet format	61
Figure 3.13	System Flow Diagram	63
Figure 4.1	Typical compute instruction (4-tuple)	68
Figure 4.2	Operation descriptor (5-tuple example)	69
Figure 4.3	Compute Instruction details	70
Figure 4.4	ROI Storage	71
Figure 4.5	Storage Descriptor	73
Figure 4.6	Configuration tuple	74
Figure 4.7	Configuration instruction types	79
Figure 4.8	Host unsolicited download first NoC packet	80
Figure 4.9	Host transfer NoC data only packet	80

Figure 5.1	Sub-System Column (SSC) Flow Diagram	82
Figure 5.2	Manager block diagram	83
Figure 5.3	System controller	84
Figure 5.4	Host boot code download NoC packet	85
Figure 5.5	MRC block diagram	91
Figure 5.6	Result data processor	93
Figure 5.7	MWC block diagram	94
Figure 5.8	PE block diagram	95
Figure 5.9	Downstream OOB data transactions	96
Figure 5.10	Downstream OOB simulation waveform	96
Figure 5.11	PE controller block diagram	97
Figure 5.12	Streaming operations block diagram	101
Figure 5.13	SIMD wrapper block diagram	102
Figure 5.14	Upstream data transactions	103
Figure 5.15	Sub-System Column (SSC) Block Diagram	104
Figure 6.1	Manager and PE Die layouts	114
Figure A.1	Classifier layer	124
Figure A.2	Classifier additional implementation layers	124
Figure A.3	Classifier layer stOp/SIMD implementation	125
Figure A.4	Classifier layer stOp/SIMD sequence timing	125
Figure B.1	Classifier additional implementation layers	127
Figure B.2	Pooling implementation	127

CHAPTER

1

INTRODUCTION

1.1 Overview

Machine Learning in the form of Deep Neural Networks (DNNs) has gained traction over the last few years, in applications such as image recognition and speech recognition. DNNs are constructed from a basic building block, the Artificial Neuron (ANe). With popular DNNs, the Artificial Neural Network (ANN) is often formed from tens of layers with each layer containing many ANes. In most cases, these layers are processed in a feed-forward manner with one layer being the inputs to the next layer. Therefore, useful DNNs often require hundreds of thousands of ANes and within the network, each ANe can have hundreds, even thousands of feeder or pre-synaptic ANes.

There have been implementations that use different number formats from double precision floating point to eight bit integers, but in all cases these useful ANNs have significant memory requirements to store the connection weights (parameters), therefore requiring high-capacity dynamic

random-access memory (DRAM) to store the ANe parameters.

There have been many successful attempts to accelerate ANNs, but in most cases the focus is on a subset of the DNN known as the Convolutional Neural network (Convolutional Neural Network (CNN)). CNNs assume a significant amount of reuse of the weights connecting ANes and thus they can take advantage of local memory (SRAM).

Much of the ASIC and ASIP ANN research has focused on taking advantage of the performance and ease of use of SRAM. These implementations can be shown to be effective with specific ANN architectures, such as CNNs where the ANN parameters can be stored in SRAM in a cache-like architecture avoiding constant accessing of the slower DRAM. In addition, to achieve a high performance, these rely on processing a batch of inputs, such as processing a batch of images or voice recordings using the same ANN.

The work in this paper considers embedded applications that require the processing of a disparate set of useful sized ANNs. The work assumes that the application system is utilizing ANNs for the processing of various sub-systems, such as navigation, engine monitoring etc. This work also does not assume the ANN is specifically a CNN but a DNN where there may not be opportunities to store and reuse portions of the ANN in SRAM. A further assumption is that the target embedded devices does not include opportunities to perform batch processing. Under these circumstance, when these implementations need to constantly load ANN parameters directly from main memory, the performance is constrained to the DRAM interface bandwidth and the performance of SRAM-based ASIC/ASIP implementations are severely degraded to the point of being unacceptable.

This work uses the DRAM as the primary processing storage and employs minimal SRAM for the processing of the ANe. In addition, the work considers Three-Dimensional (3D) integrated circuit technology and a custom 3D-DRAM. By employing Three-Dimensional Integrated Circuit (3DIC) technology, this work takes advantage of the reduced energy and area and increased connectivity and bandwidth to allow the DRAM to be employed efficiently without the need for local SRAM. This work demonstrates that a 3DIC system based on a customized 3D-DRAM could be used in embedded applications requiring at or near real-time performance for systems running multiple ANNs.

It should be noted that this work does not design a custom 3D-DRAM but answers the question “if such a device were available, can we employ it within a useful ANN system.”

An overview of ANN technology is given in Section 1.3. The motivation for this work is given in Section 1.4. An overview of 3DIC technology is given in Section 1.5 and the pros and cons of DRAM and SRAM along with some proposed DRAM customizations are given in Section 1.6. Some state-of-the-art implementations are reviewed in Chapter 2. An overview of the proposed system is described in Chapter 3 with more details in Chapter 5. An overview of the instruction architecture is given in Chapter 4. Simulation results are shown in Chapter 6. The conclusion and further work are discussed in Chapter 7.

1.2 Abbreviations

Acronyms

3D Three-Dimensional

3D-DRAM Three-dimensional dynamic random-access memory

3DIC Three-Dimensional Integrated Circuit

ANe Artificial Neuron

ANN Artificial Neural Network

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Instruction-set Processor

binary16 half-precision floating-point

binary32 single-precision floating-point

BOOTP Bootstrap Protocol

CNN Convolutional Neural Network

DDR Double Data Rate

DiRAM4 Dis-Integrated 3D DRAM

DMA direct memory access

DNN Deep Neural Network

DRAM dynamic random-access memory

DRC design rule check

EDRAM embedded dynamic random-access memory

EOD end of descriptor

EOM end of message

ESD Electrostatic discharge

FIFO first-in first-out queue

FLOP floating point operation

FLOPS floating point operations per second

FP floating-point

FSM finite-state machine

GPU graphics processing unit

HBM High Bandwidth Memory

HMC Hybrid Memory Cube

IC Integrated Circuit

IO input/output

IP Intellectual property

ISCA International Symposium on Computer Architecture

JTAG Joint Test Action Group

KGD Known Good Die

LSTM Long Short-term memory

MAC multiply-accumulate

MMC main memory controller

MOD middle of descriptor

MOM middle of message

MRC memory read controller

MSB Most Significant Bit

MTU maximum transmission unit

MWC memory write controller

NoC Network-on-Chip

NOP no operation

OOB Out of Band

PC Program Counter

PE Processing Engine

QDR Quad data rate

RDP return data processor

ReLU Rectified Linear Unit

ROI region-of-interest

RTL register-transfer level

SDP storage data processor

SDRAM synchronous dynamic random-access memory

SFU special function unit

SIMD Single-Instruction Multiple-Data

SoC System-on-Chip

SOD start of descriptor

SOM start of message

SRAM static random-access memory

SSC Sub-System Column

StOp Streaming Operation block

TDP total design power

TPU tensor processing unit

TSV Through-Silicon Via

1.3 Artificial Neural Networks

Recently, there has been much interest in the use of artificial neural networks in systems that employ tasks such as image recognition [27], text recognition [38] and game playing [33]. In particular, in the field of image recognition these artificial neural network models have demonstrated superior performance over other state-of-the-art technology [27]. These artificial neural networks will continue to be applied to numerous other areas such as voice recognition, text recognition, face recognition and autonomous control.

Artificial neural networks (ANN) take their inspiration from neuron behavior observed in the mammalian brain, although implementations are simplifications of what actually exists in the brain. The mammalian neuron is a cell that receives input and generates output in the form of electrical and chemical processes. The neuron has a cell body (or soma), a group of dendrites, which provide the inputs from other cells, a cell body, an axon that generates the output signals, and the axon terminals that are the outputs of the cell. The connection from a cell's output, or axon terminal, to another cells input, or dendrite, is known as a synapse. The connection in the synapse is a chemical process stimulated by electrical impulses. The neuron can be seen in Figure 1.1.

The connection from one cell to another has both an associated delay and strength. The strength of the connection can be influenced by the size of the pre-synaptic neuron spike or by the pre-synaptic neuron generating a series of spikes rather than a single spike.

It is known that mammalian neurons generate “spikes” in response to inputs that for humans include sight, touch, sound etc.. This spiking behavior is often referred to as the neuron being activated. When these neurons are activated, their spikes propagate to other neurons. Under certain conditions, the combination of the various inputs to a neuron causes it to activate. A particular neuron may have many hundreds, perhaps thousands of other neurons connected to its “input.” These input neurons are referred to as pre-synaptic neurons. These pre-synaptic neurons may provide input to many neurons that are referred to as post-synaptic neurons. A particular neuron can get activated by a particular arrival pattern of pre-synaptic neuron spikes or simply by the intensity of the pre-synaptic

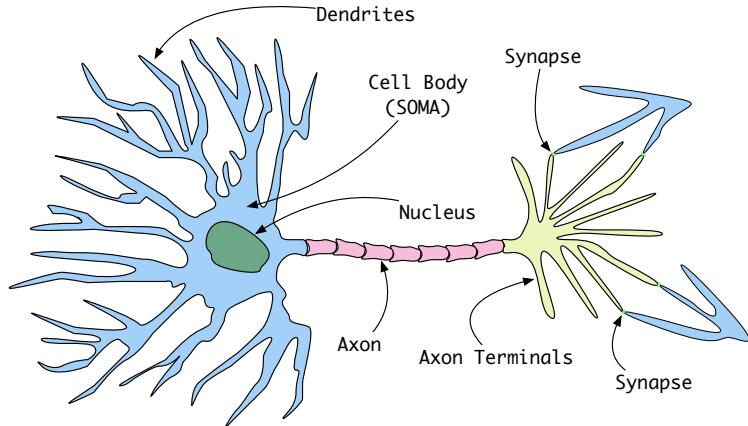


Figure 1.1 Artist's impression of a mammalian neuron [44]

spikes.

The spiking behavior of a neuron also varies and many spiking profiles have been observed, including single spikes, groups of spikes and repetitive spiking. It is believed that information is carried in the delay and strength of the connections and how pre-synaptic neurons combine to cause a neuron to activate. In simple terms, if a neuron is activated by its pre-synaptic neurons, then the activation of the neuron means a pattern has been detected which will influence a reaction. In mammalian terms, that might be the detection of a threat from both smell and sight neurons and the reaction is to control muscles resulting in flight.

The various chemical and electrical processes that result in the generation and propagation of these neuron spikes is beyond the scope of this dissertation, but how neurons and networks of neurons are artificially emulated is what we will discuss next.

1.3.1 ANN Overview

When modeling these neurons in artificial neural networks, the neuron models either generate actual spikes similar to actual neurons or produce a value that is proportional to the rate at which spikes occur. These ANNs can be categorized as rate-based coded or spike time coded neurons [4][6].

When used in networks of neurons, both model types employ a connection weight between the

pre and post-synaptic neuron. However, the spiking neuron network also introduces a time delay associated with the connection.

The spiking neuron model [37] is characterized by:

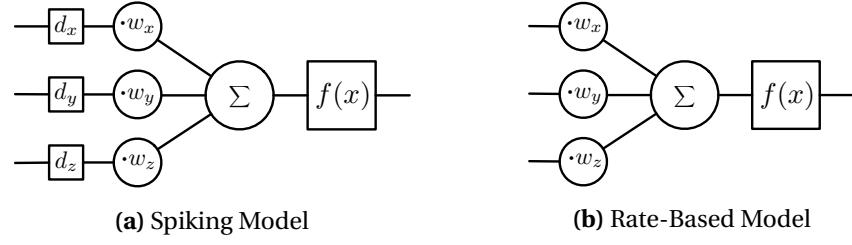
- Connections have a strength and a delay
- The weighted inputs from all pre-synaptic neurons are accumulated
- The accumulated inputs drive an activation function
 - The activation function $f(x)$ is based on differential equations
 - Varying levels of complexity from Leaky integrate and fire [5] to Izhikevich [21] (see Fig. 1.4a)

The Rate-based neuron model [6] is characterized by:

- Connections between neurons have only a strength
- The weighted inputs from all pre-synaptic neurons are accumulated
- The accumulated inputs drive a differentiable non-linear activation function $f(x)$
 - Such as Sigmoid [37] or Rectified Linear Unit (ReLU) [32] (see Fig. 1.3):

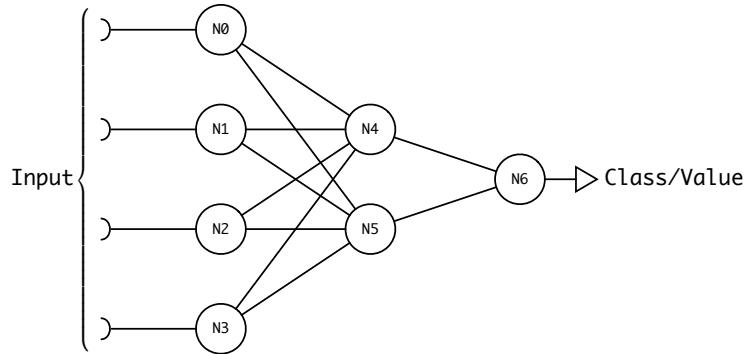
The artificial neurons are connected in networks, typically with layers of sub-networks which are in effect separated by the non-linear activation function, to emulate complex behavior. Examples of both rate-based and spiking artificial neural networks can be seen in Fig. 1.5a and Fig. 1.5b respectively. Typically neural networks process in a feed-forward fashion. Considering Fig. 1.2c, this means the input arrives on the left, the inputs propagate to neurons N0 through N3. When N0 through N3 are processed, their values propagate forward to neurons N4 and N5 etc.. Sometimes ANNs also include recursion where for example neurons N0 through N4 are not only influenced by the input, but also by themselves. Many ANNs operate only in feed-forward fashion but some popular ANNs, such as Long Short-term memory (LSTM) [19], employ recursion.

Another popular type of ANN, the DNN [1][18] has gained traction over the last few years. DNNs get good press in applications such as image recognition and speech recognition. DNNs are often formed from tens of layers of ANEs with each layer containing many ANEs. DNNs are also processed



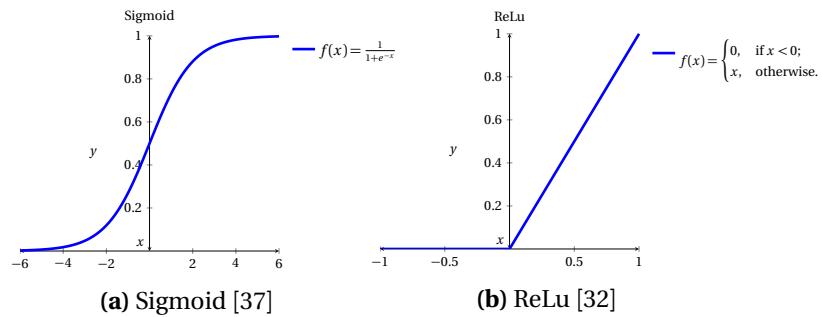
(a) Spiking Model

(b) Rate-Based Model



(c) Network of Artificial Neurons

Figure 1.2 Artificial Neurons and Network [6][34]



(a) Sigmoid [37]

(b) ReLu [32]

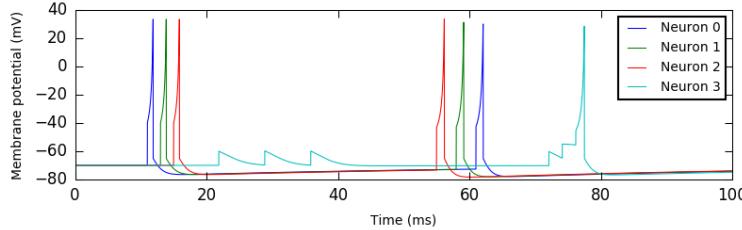
Figure 1.3 Example Rated-Based Model Activation functions

$$v' = 0.04v^2 + 5v + 140 - u - I$$

$$u' = a(bv - u)$$

$$\text{if } v \geq 30 \text{ mV, then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases}$$

(a) Izhikevich Model[21]



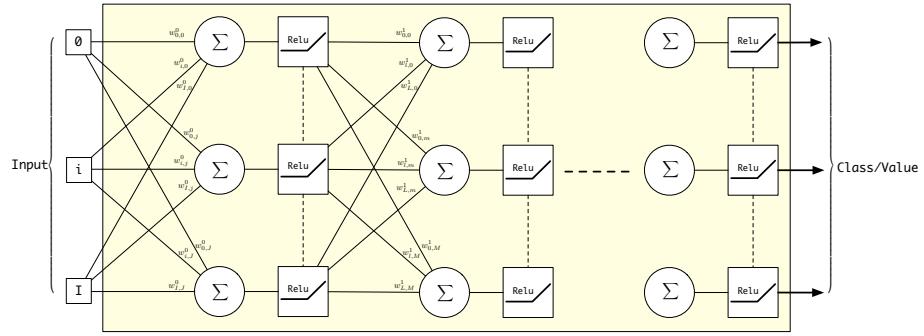
(b) Izhikevich Model Simulation [7][21]

Figure 1.4 Example Spiking Activation Function Model

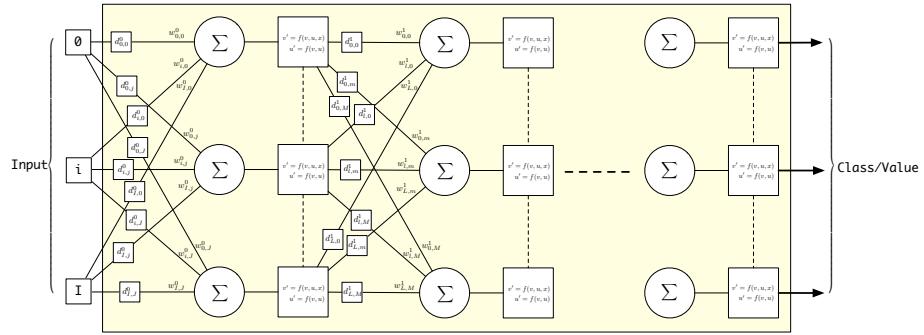
in a feed-forward manner with one layer being the inputs to the next layer. As mentioned in [27], these useful DNNs often require hundreds of thousands of ANes and within the network, each ANe can have hundreds, even thousands of feeder or pre-synaptic ANes. There have been implementations that use different number formats from double precision floating point to eight bit integers, but in all cases, these useful ANNs require a significant amount of memory to store the connection weights (parameters).

Although the spiking neural network more closely models the behavior of real neurons, over the last 20 years there have been breakthroughs in the “teaching” of rate-based models, especially with the introduction of the back-propagation algorithm [43] and stochastic gradient descent [49] which are used to “learn” the connection weights of a DNN. Along with the abundance of data now available in the form of voice, images etc. to “teach” these networks using back-propagation, most of the effective applications of ANNs have employed these rate-based models.

This work does not address the training or “teaching” of these rate-based ANNs. The training is mostly performed offline. This work is addressing inference using ANNs where the ANNs are used for



(a) Rate-based Model Artificial Neural Network (with ReLu activation function)



(b) Spiking-based Model Artificial Neural Network

Figure 1.5 Layered Artificial Neural Networks

example to detect objects in an image or reproduce the output of a cost function based on the observed values. During inference, the most computationally intensive operation is the multiply-accumulate associated with the ANe activation, which can involve hundreds or thousands of multiply-accumulates. The ANe activation calculation for the rate-based ANe in Figure 1.2b is shown in (1.1).

$$\text{ANe activation} \quad A = f\left(\sum_{n=0}^{C_n} W_n \cdot A_n\right) \quad (1.1)$$

C_n is the number of pre-synaptic connections

W_n is the weight of a connection

A_n is the state of the pre-synaptic ANe

and $f(x)$ is the activation function such as ReLu [32]

1.3.2 ANN Layers

In Figure 1.2c and 1.5a, the ANN is shown to be constructed using layers of ANEs. It has long been known that a single layer of ANEs can be used to partition an n-dimensional input [6] using a linear combination of the inputs, as shown in Figure 1.6a. However, a more complex partition, as shown in Figure 1.6b, can only be achieved using multiple layers of ANEs. To ensure the multiple layers cannot be mathematically collapsed into a single layer, the activation function $f(x)$, as shown in Figure 1.2b must be a non-linear function.

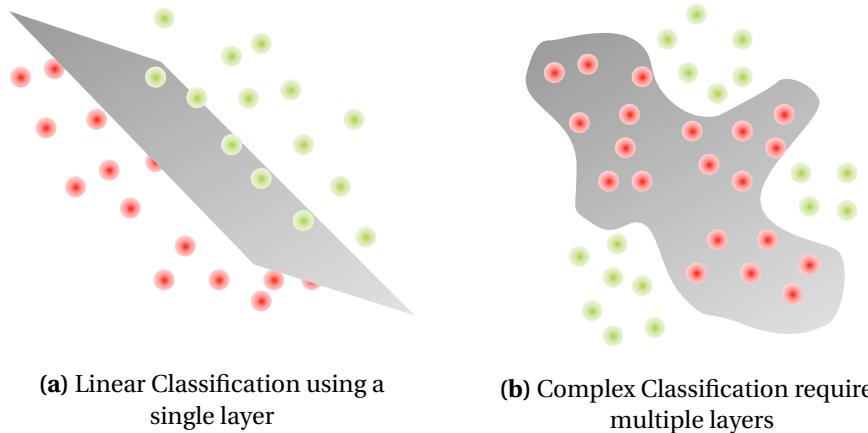


Figure 1.6 Classification using ANNs layers [6]

1.3.2.1 Deep Neural Networks

As mentioned, a single layer of neurons can be used as a linear classifier as long as the classes can be separated using a linear function. Even some simple cases cannot be linearly separated, an example often used is an exclusive-OR gate [6].

Even with a layered ANN, the final output comes from a single layer. To allow this final layer to linearly separate classes, the original input needs to be transformed into a space where the classes can be linearly separated. Deep Neural Networks (DNNs) are ANNs that incorporate many layers of ANEs, often that are tens of layers deep. The additional layers are incorporated to translate the space of the input so the various classes being identified can be separated using linear classifiers in the later layers.

Recently, an example of a DNN known as a Convolutional Neural Network (CNN) demonstrated high levels of efficacy when used to classify objects in images [27]. These CNNs use the early layers to identify low-level features and later layers are used combine these features into yet more high-level features [13][28][45]. Finally, the combination of high-level features is used to identify the required classes. This layering is shown in Fig. 1.7.

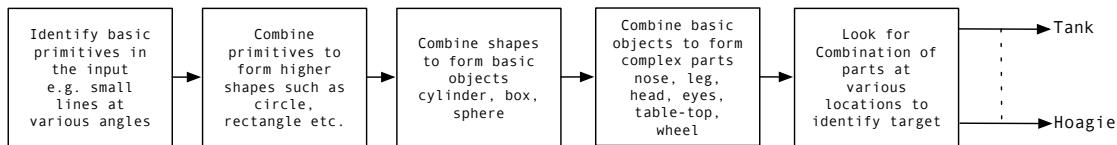


Figure 1.7 Deep network showing feature layers

In figure 1.7, the final layer is often a fully connected linear classifier with the output representing the probability of a particular class being present in the image. In practice, these DNNs can be used as classifiers or as function approximators.

1.3.2.2 Feature Layers

For the most part, different ANNs are characterized by how the ANEs are interconnected and the activation function employed.

The typical DNN layers are processed in a feed-forward fashion where the pre-synaptic ANes are formed from ANes in the previous layer. There are some types of DNN that also include recursive connections where the pre-synaptic ANes include ANes from the current layer. A popular recursive DNN is Long Short-term memory (LSTM) [19]. Although this work does not preclude supporting LSTM in the future, the focus of this work is on the feed-forward type DNN.

As described in Section 1.3.2.1, a DNN layer transforms the previous layer with each higher layer providing a coarser grained transformation. This is best seen in image recognition applications, where the early layers identify low-level shapes or features, such as angled lines. The following layers are used to identify higher order shapes such as circles, blocks etc. Although the features detected during the image recognition application are somewhat intuitive, it is believed that in less intuitive applications the DNN performs a similar fine to coarse feature extraction.

The connections between layers can be locally- or fully-connected. With locally-connected layers as shown in Figure 1.8, a layer's pre-synaptic ANes are formed from regions of the previous layer. With fully-connected layers as shown in Figure 1.9, a layers pre-synaptic ANes are formed from all ANes in the previous layer. In many cases, a DNN is constructed with lower layers being locally-connected and higher layers being fully-connected [27].

In early uses of locally-connected ANNs, the first layers weights were often hand-generated, an example being Gabor filters [28]. With automatically trained ANNs, the feature detectors at each layer are often created during training. Some contrived examples of locally-connected feature detectors are shown in Figure 1.10.

The pre-synaptic ANes of a locally-connected ANe are formed from a particular region-of-interest (ROI) of the previous layer using the weights from a feature filter. Another locally-connected ANe may use the same ROI but employs a different filter. In practice, for a particular ROI, a number of feature filters are employed resulting in a number of ANes being associated with the same ROI in the previous layer. To reiterate, these feature filters all operate on the same ROI. A different ROI will result in another group of ANes all using their own feature filters. The resulting locally-connected layer becomes a 3D layer with its X-Y coordinates representing a reference to a particular ROI and the

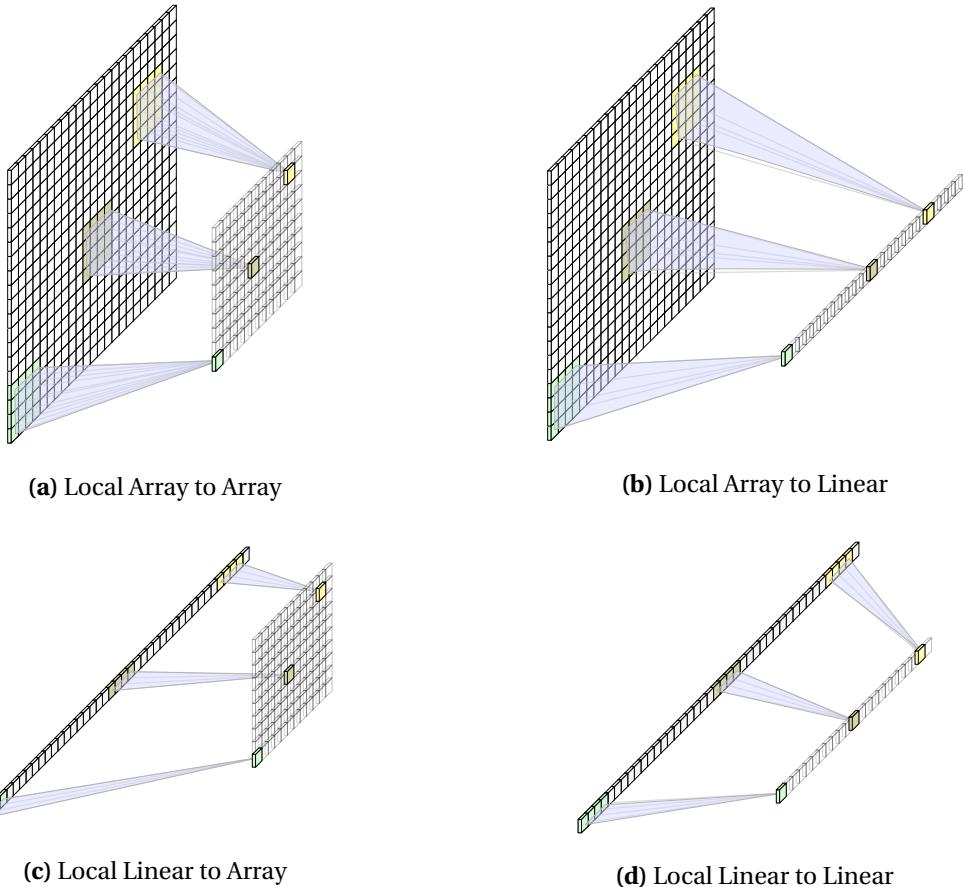


Figure 1.8 Locally Connected Layer to Layer Connection Types

Z-dimension representing the various filters applied to that ROI. An example 3D locally-connected layer can be seen in Figure 1.11.

So these locally-connected layers have multiple filters applied to the same ROI and the next layer becomes a 3D array with the Z-axis representing the features. The number of feature filters applied at each layer can be tens to hundreds of filters. The filters employed in a layer following one of these 3D locally-connected layers are themselves 3D. With tens to hundreds of features in the previous layer the number of weights associated with each filter is usually hundreds to thousands of weights.

The feature filters employed in the locally-connected layers can be unique to the regions of the previous layer or the same filters can be employed across the entire previous layer. In the case of employing the same filters across the entire input layer the ANN is known as a Convolutional Neural

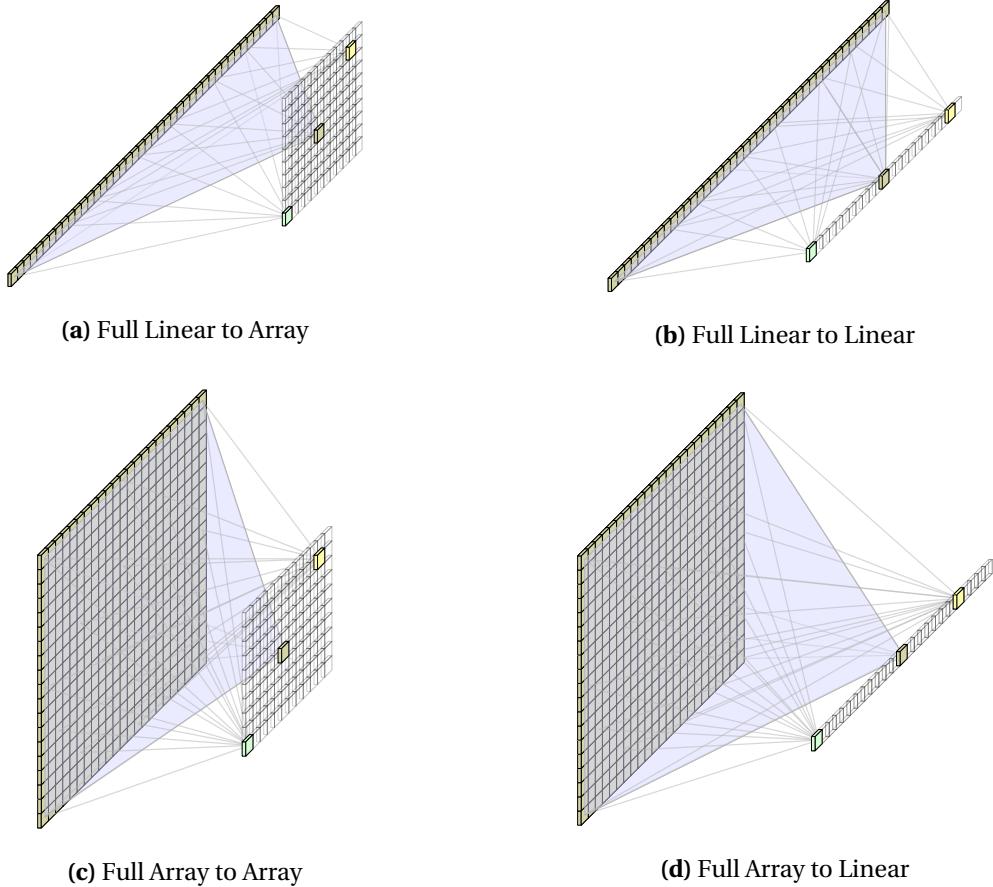


Figure 1.9 Fully Connected Layer to Layer Connection Types

Network (CNN). The CNNs are examples of ANNs that can take advantage of reuse. These CNNs can store the filter parameters in local SRAM and construct an entire feature plane. These CNNs are considered a subset of the generic case of DNN. This work considers the more general DNN case and supports acceleration of generic DNNs that includes CNNs.

An example of a DNN can be seen in Figure 1.12 with the layer configurations shown in Table 1.1. A CNN similar to this has demonstrated high levels of efficacy in image recognition applications. **Therefore, this work will use the parameters from Table 1.1 as a template for a baseline ANN for estimating the storage and processing requirements and the range of pre-synaptic fanins.**

To approach the capabilities observed in human behavior, such as object recognition, ANNs have become very large. The example shown in Figure 1.12, which is based on the work from [27], has

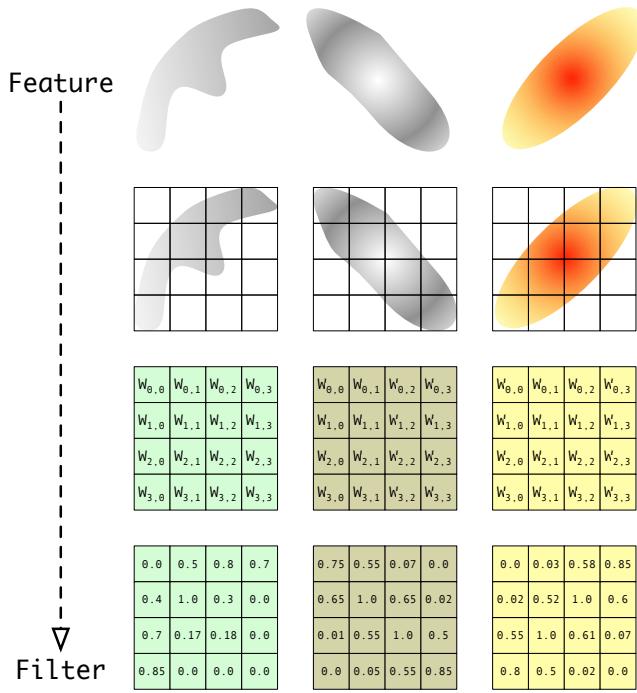


Figure 1.10 Features and locally-connected filters (kernels)

hundreds of thousands of ANEs and hundreds of millions of connection weights (see Table 1.1). These ANNs utilize these hundreds of thousands of ANEs to implement what a human would consider a relatively straightforward task. For example, a “useful” ANN similar to that described in [27] which was used to recognize up to 1000 different object classes, has a network size of approximately 650,000 ANEs and 630 million synaptic connections [26].

The increased performance of ANNs over classical methods in image recognition and voice recognition might suggest that ANNs will out-perform operations performed in other applications.

If ANNs fulfill their potential, systems employing ANNs will utilize them for various functions, such as engine monitoring, anomaly detection, navigation, etc., all within the same system. Considering the various functions a complex customer facing or embedded application system performs, it is likely that many real-world applications will employ multiple disparate instances of these usefully sized ANNs. Assuming these complex functions will require ANNs similar in size to figure 1.12 and [27], these implementations will be processing multiple large ANNs at or near real-time.

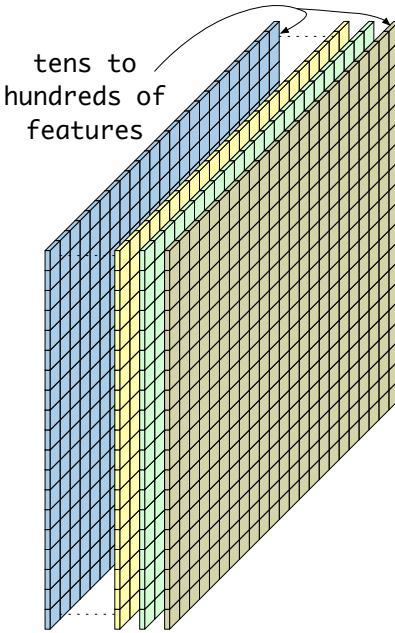


Figure 1.11 Single Layer constructed from 3D layers of Features

1.3.3 ANN Processing

Considering the storage required for the input, the ANe states and most significantly the weights for connections, the storage requirements results in gigabytes of memory. When these ANNs are required to be solved in fractions of a second, the processing and memory bandwidth becomes prohibitive.

As a metric, this work assumes that any useful ANN will be similar to that shown in Table 1.1 which utilizes more than 900 thousand ANEs and approximately 200 million parameters.

When it comes to estimating storage requirements for ANNs there is a lot of debate regarding the precision of number format for the parameters. There has been work on the impact of changing the precision of the number format employed during training and inference. These formats can vary between eight bit fixed-point to 64-bit double precision. For the baseline requirements this work assumes 32-bit single-precision floating-point (FP).

Assuming an ANN similar to that shown in Table 1.1 with 772 thousand ANEs and an average fanin to each ANe of 1650, a system employing 10 ANNs for various disparate functions and an average processing time of 16 ms suggests an average bandwidth of ~ 26 Tbit/s (see equation 1.2).

$$\begin{aligned}
\text{Maximum Bandwidth} &= \sum_{n=0}^{N_n} \left(\frac{\bar{N}_a \cdot \bar{C}_p \cdot b_w}{T_p} \right) \text{bit/s} \\
&= \sum_{n=0}^9 \left(\frac{772 \times 10^3 \cdot 1.65 \times 10^3 \cdot (32+1)}{16 \times 10^{-3}} \right) \\
&= \sum_{n=0}^9 2.63 \text{Tbit/s} \\
&\approx 26 \text{Tbit/s}
\end{aligned} \tag{1.2}$$

where N_n is the number of ANNs

N_a is the average number of ANEs

C_p is the average number of connections

b_w is the number of bits per parameter

and T_p is the processing time

Note: assumes ROI streamed to all lanes

When implementing ANNs, the memory requirements are also significant. The storage is required for the input, the ANe states and most significantly the parameters for each of the ANe's pre-synaptic connections. For the case shown in Table 1.1, there are 202 million parameters requiring 0.81 GB and 772 thousand ANEs requiring 3.88 MB storage. The storage required for 10 ANNs is of the order of 8.0 GB (1.3).

$$\begin{aligned}
\text{ANN Memory} &= \sum_{n=0}^{N_n} ((\bar{N}_p + \bar{N}_a) \cdot b_w) \text{Gbit} \\
&= \sum_{n=0}^9 ((202 \times 10^6 + 772 \times 10^3) \cdot 32) \\
&= \sum_{n=0}^9 6.49 \text{Gbit} \\
&= 64.9 \text{Gbit} \equiv 8.1 \text{GB}
\end{aligned} \tag{1.3}$$

where N_n is the number of ANNs

N_p is the number of parameters per ANN

N_a is the number of ANes per ANN

and b_w is the number of bits per parameter

The approximate system bandwidth and storage requirements are shown in Table 1.2.

Table 1.2 Estimated system bandwidth and storage design requirements

Parameter	Value
Bandwidth	26 Tbit/s
Storage	8.0 GB

Given the bandwidth and storage requirements shown in Table 1.2, the problem becomes “**how to meet these requirements within the power and space constraints expected for embedded systems**” and will be discussed

	Type	Layers									
		1	2	3	4	5	6	7	8	9	10
Dimensions	X	256	55	27	13	13	13	13	4096	4096	Fully
	Y	256	55	27	13	13	13	13	1	1	1024
	Z	3	96	96	256	384	384	256	1	1	1
Filter Dimensions	X	na	11	2	5	2	3	3	3	13	4096
	Y	na	11	2	5	2	3	3	3	13	4096
	Z	na	3	1	96	1	256	384	384	256	1
Stride	na	na	4	2	2	1	1	1	na	na	Aggregate
Pre-synaptic Fanin	196608	290400	69984	186624	43264	64896	3456	43264	4096	4096	16550
Number of ANs	na	na	na	na	na	na	na	na	1024	1024	772544
Number of Weights	na	34848	363	4	2400	4	2304	13	16777216	4194304	2.02×10^8

Table 1.1 Baseline ANN layer configuration [27]

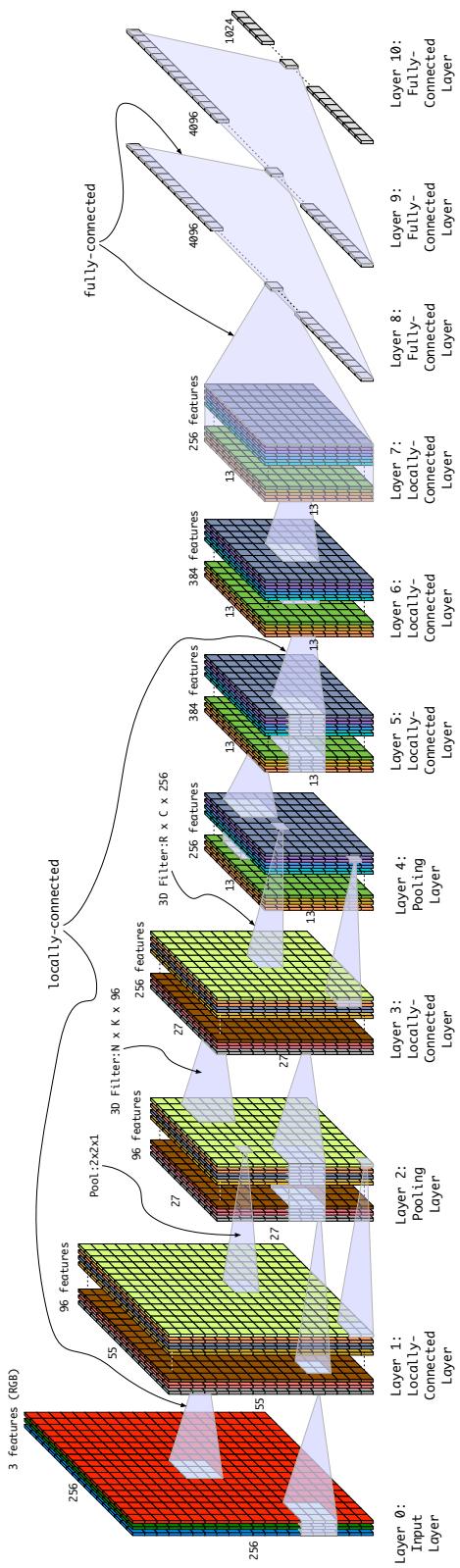


Figure 1.12 Baseline DNN showing layer order [27]

1.4 Motivation

1.4.1 The Problem

This work focuses on embedded applications employing disparate ANNs and therefore assumes there are limited opportunities for both weight reuse and batch processing.

Given the storage requirements shown in Table 1.2, it is generally accepted that DRAM is required to store the ANN parameters [2][9][31].

When considering systems that will employ multiple DNNs simultaneously, we assume that these embedded systems will require usable memory bandwidth on the order of tens of Tbit/s (1.2).

In these cases, **DRAM bandwidth is the bottleneck** [31].

1.4.1.1 Why not SRAM?

Why is it that much of the ASIC and ASIP ANN research employs SRAM as an intermediate store?

In practice, there are benefits if the processing elements can operate out of SRAM, such as higher performance and potentially low power. When compared to DRAM, SRAM has low latency. Also, the DRAM access protocol is much more complicated to implement than SRAM. The high latency and DRAM protocol force the system to interleave accesses. Overall, when compared to DRAM, SRAM is relatively easy to use.

Given that DRAM is used for the main memory storage, having the processing elements operate out of SRAM requires that the high cost of transferring data from the DRAM to the SRAM be absorbed by using that data multiple times or “reused.” So using SRAM for intermediate storage makes assumptions on the type of ANNs that can be supported and the application in which the ANN is being deployed. The primary requirement of the type of ANN and the deployed application to allow effective use of SRAM is “reuse,” so once parameters are transferred and stored in SRAM, these parameters can be reused such that the SRAM isn’t simply an intermediate memory but something akin to a cache.

In some ANNs there are reuse opportunities. A prime example is CNNs, where the connection weights are reused. In CNNs, common feature filters are passed across an input to form the next

layer. These filter “kernels” are typically held in SRAM and only the input is read from DRAM thus reducing the dependency on DRAM bandwidth. Even with DNNs where different filters may be used for different ROIs some filter reuse may be available. Another form of reuse is in cloud applications or in training where there is opportunity to reuse inputs while performing batch processing. But SRAM comes at a price, often physical layouts of ANN processors are dominated by the silicon area of the SRAM [8][23][24]. Because of the relatively large area required for SRAM, companies attempt to create custom SRAMs to minimize the area impact. So ASIC and ASIP ANN implementations that target applications that have considerable weight reuse and/or batch processing opportunities can effectively use SRAM as an intermediate store. But to reiterate, this work assumes the target application have limited or no opportunities for weight reuse or batch processing.

1.4.2 Alternatives

1.4.2.1 Graphics processing units (GPUs)

The requirements of these applications would be satisfied by employing multiple GPUs. In practice, GPUs are used to implement large ANNs and in some ANN architectures, such as CNNs, they are quite effective. However, we should not forget they are a) not optimized purely for ANN processing, b) restricted by available SRAM and c) power hungry. These limitations will limit the effectiveness of GPUs. Even in the case of newer GPUs that are employing 2.5DIC technology, the memory bandwidth will still be limited by available DRAM technology. For example, a 2.5D solution employing High Bandwidth Memory (HBM) would be limited to a maximum raw bandwidth on the order of 6 Tbit/s [35]. Also, it has proven very difficult, if not impossible to take advantage of the available memory bandwidth [16] [23].

A solution could employ multiple devices, but there would be significant power and real-estate issues. The typical high performance GPU consumes between 100 W and 200 W. A multiple GPU implementation would have a high real estate impact and a system power approaching a 1 kW. Overall GPUs have limited suitability to meet this work’s target application requirements [31].

1.4.2.2 ASICs/ASIPs

Much of the ANN application specific (ASIC/ASIP) research has focused on taking advantage of the performance and ease of use of static random-access memory (SRAM). These implementations can be shown to be effective with specific ANN architectures (e.g. CNNs), server applications or the small point problems but when a system requires multiple disparate ANNs in an embedded application, **existing implementations do not provide the required flexibility, storage capacity and deterministic performance.**

According to the Google paper [23] on their tensor processing unit (TPU) [23] ASIC, the architecture research community is paying attention to ANNs. Of all the papers at International Symposium on Computer Architecture (ISCA) 2016, nine discussed hardware accelerators for ANNs. Of those nine papers, all looked at CNNs but only two mentioned other types of ANNs. Unfortunately CNNs represent only about 5% of Google's datacenter ANN workload.

The applications targeted by the Google TPU [23] assume multiple requests, so reuse in the form of batch processing is still of great benefit, but the bulk of the requests in [23] are fully-connected DNNs and in these cases weight reuse is not as beneficial and the performance of the TPU is degraded when implementing these fully-connected DNNs.

Implementations that focus on CNNs can suffer from severe degradation in performance when targeting generic types of ANN, such as locally- and fully-connected DNNs and LSTMs. Implementations that try to provide adequate on-chip storage, sometimes in the form of embedded dynamic random-access memory (EDRAM) have to have many instances of the ASIP to provide the required total storage [31]. In these cases, the large number of instances results in huge performance capabilities that far exceed the requirements of this work's applications. The total power also becomes prohibitive.

Considering this work focuses on embedded applications employing disparate ANNs and assumes both weight reuse and batch processing do not apply, regardless of how implementations employ SRAM as an intermediate store, **DRAM bandwidth is the bottleneck.**

1.5 Three-Dimensional Integrated Circuits (3DICs)

Over the last couple of decades, the ever-shrinking world of Integrated Circuits (ICs) has enabled the introduction of devices for various uses such as personal computing and cell phones. As IC technology has shrunk, design complexity has grown to take advantage of the hundreds of millions of transistors available on a typical IC. These ICs have evolved from performing small functions to becoming systems on a chip.

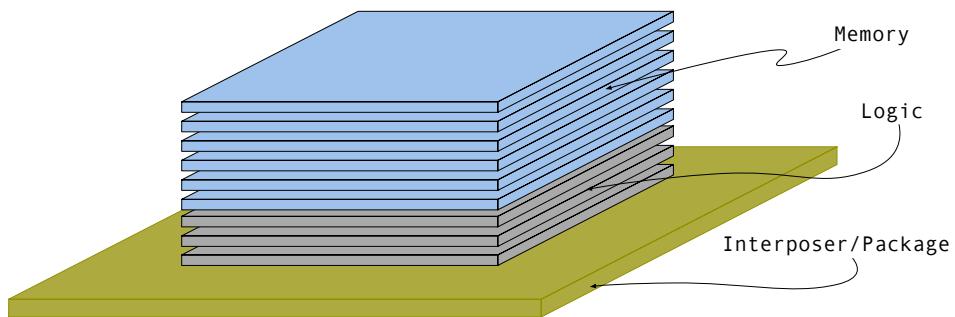
At some point, an IC has to interface with another function and often that communication involves the moving of data to and from memory and the increase in complexity often drives a need for higher memory data bandwidth. Integrated Circuit (IC) complexity has been doubling approximately every two years but the external interfaces are restricted by physical limitations. Within the System-on-Chip (SoC), the designer can take advantage of very wide interfaces – often thousands of bits wide – to increase bandwidth, but when data is moved to off-chip memory, the widest buses are usually hundreds of bits wide. One way to avoid this limitation is to employ 3DICs (see Figure 1.13). The advantages of 3DICs are well understood, reducing the amount of off-chip communication increases bandwidth and reduces power. The power reduction comes from not having to drive the relatively high capacitance inputs and outputs.

1.5.1 Pros and Cons

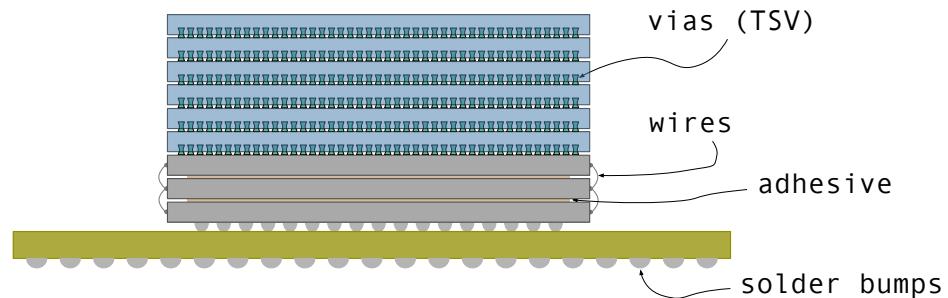
Taking advantage of 3DICs means stacking die on top of one another and making connections directly between the die. These connections can be in the form of wire made at the edges of the die or using vias buried in the die itself.

Below is a summary the benefits of 3DICs :

- Reduced Power
 - Mainly from not having to drive external outputs and receiving external inputs
- Increased Connectivity



(a) Different Die types stacked and mounted on an interposer/package substrate



(b) Connection Types

Figure 1.13 3DIC Stack of Die

- Maintaining very wide buses through the SoC increases bandwidth
- Ability to mix heterogeneous technology
 - Mixed Analog/Digital
 - Mixing memory technology and logic technology
- Increased density and mitigation against the slowing of Moore's Law
 - Using the vertical domain to increase perceived transistor per square millimeter
- Potentially lower costs by combining simpler die rather than building a large die
 - Yield benefits from combining higher yield die
- Possibility of novel architectures [25]

Some disadvantages of 3DICs are:

- Reliability
- Cost
 - Being a relatively new technology, it is still expensive
 - TSV technology is still unreliable

There is still some reluctance to fully embrace 3DICs but undoubtedly the various barriers will be broken down.

In [20] there are four definitions of 3DIC interconnects:

- 3D-Wafer level package [20]
 - In this case, different die are stacked and then connected using traditional bond bumps and/or bond wires at the periphery of the chips.
 - This technique provides better transistor density compared to traditional 2D-IC with improvements in interconnect density.

- 3D-Stacked SoC [20]
 - In this case, different die are stacked and then connected using TSVs. The TSVs connect the dies to intermediate metal layers known as global metal layers. This allows the individual die to maintain a high level of functionality and thus is similar to connecting functional building blocks, meaning the individual die are likely to be significant functional pieces of Intellectual property (IP).
 - Using TSVs provides a medium level of interconnect
- 3D-Stack IC [20]
 - In this case, different die are stacked and then connected using TSVs. The TSVs connect the dies to intermediate higher metal layers known as global metal layers. This infers the individual dies are not large functioning pieces of IP.
 - Using TSVs provides a high level of interconnect
- 3D-Integrated Circuit [20]
 - In this case there are not multiple dies. Instead, the additional silicon layers are deposited on top of each other with the final 3DIC device having multiple layers of transistors
 - Local metal layers are used, which along with TSVs provides a very high level of interconnect

A die stack with TSVs can be seen in Fig. 1.14.

1.5.2 Construction

There are other definitions on how the dies are bonded together:

- Wafer-to-Wafer
 - Current Electrostatic discharge (ESD) mitigation allows implementation of unbuffered input/output (IO) signals

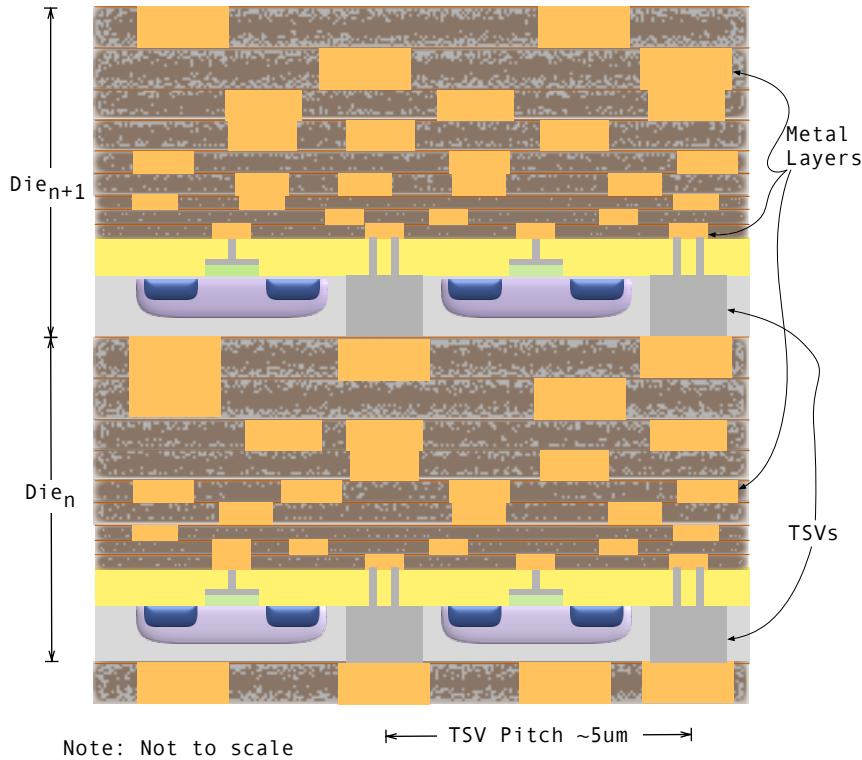


Figure 1.14 Die Stack profile [20] with TSVs

- Potential low yield because of lack of knowledge regarding Known Good Die (KGD)
- Die-to-Wafer
 - Will need additional ESD mitigation support
 - Higher yield because of KGD
- Die-to-Die
 - Will need additional ESD mitigation support
 - Higher yield because of KGD

This work is targeting 3DIC technology that supports 3D-Stacked SoC or 3D-Stack IC with high levels of interconnect. To avoid using large IO buffers for the TSV interconnect, this work assumes

that the 3DIC technology supports unbuffered interconnects which would suggest wafer-to-wafer bonding.

1.5.3 Design Guidelines

The technology roadmap in [20] and the information in [36] suggest 5 μm pitch TSVs is a reasonable design goal. The dimensions of these TSVs are of the same order of magnitude as high-level metal layers in a typical 28nm technology node so they are considered to have similar characteristics. Also, at an operating frequency of 500 MHz, these small TSVs are not considered to be transmission lines so no additional provisions are needed to route signals between die other than those imposed by the higher metal layers. However, to provide opportunities for future signal integrity analysis and improvements, this work assumes the TSVs are arranged in a signal-ground-signal array providing a one-to-one ratio of signal TSVs to power/ground TSV. So when accounting for area associated with TSVs, the number of signal TSVs is doubled.

As a large amount of TSVs are employed, TSV energy cannot be ignored. Most of the energy dissipated in the TSVs is associated with the charging and discharging of the capacitance. For TSVs with 2 μm radius on a 5 μm pitch, [3] suggests an average capacitance of 4.2 fF¹.

Assuming a supply voltage of 1.0 V, the power associated with a TSV is shown in (1.4).

$$\text{Energy to charge a TSV, } E_{tsv} = \frac{1}{2} \cdot C_{tsv} \cdot V^2 = \frac{1}{2} \cdot 4.2 \times 10^{-15} \cdot 1.0 = 2.1 \text{ fJ}$$

$$\text{Power per TSV, } P_{tsv} = E_{tsv} \cdot \text{bit rate}$$

normalizing to a clock of 1.0 GHz

$$\text{Power per TSV per Hz} = 2.1 \mu\text{W/Gbit/s/TSV} \quad (1.4)$$

The TSV design guidelines used by this work are summarized in Table 1.3.

¹[36] suggests a lower capacitance

Table 1.3 TSV design characteristics

Dimensions		Capacitance	Power
Pitch	Radius		
5 μm	2 μm	4.2 fF	2.1 $\mu\text{W}/\text{Gbit/s/TSV}$ [3]

1.6 DRAM Overview and Customizations

There are two types of memory employed in ASICs and ASIPs, static random-access memory (SRAM) and dynamic random-access memory (DRAM). Both of these technologies have a similar top-level block diagram that contains an array of storage elements, a means to address into a particular row of memory cells and a means to read and write a column of those cells. A basic block diagram is shown in Figure 1.15.

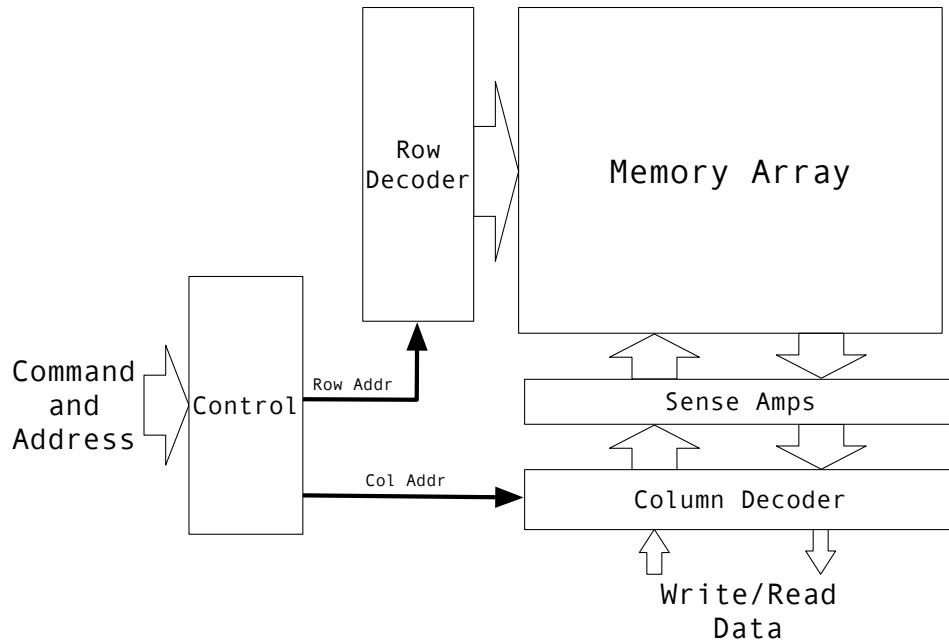


Figure 1.15 Typical Memory Block Diagram [22]

The SRAM cell takes six transistors (figure 1.16a) and the DRAM cell takes one transistor and one capacitor (figure 1.16b). This means the DRAM arrays provide five to six times more storage density when compared to a similar sized SRAM array.

The major disadvantage is the capacitor cannot hold a charge indefinitely because the leakage currents in ICs cause the charge to leak away. If kept unchecked, the stored value will dissipate and it

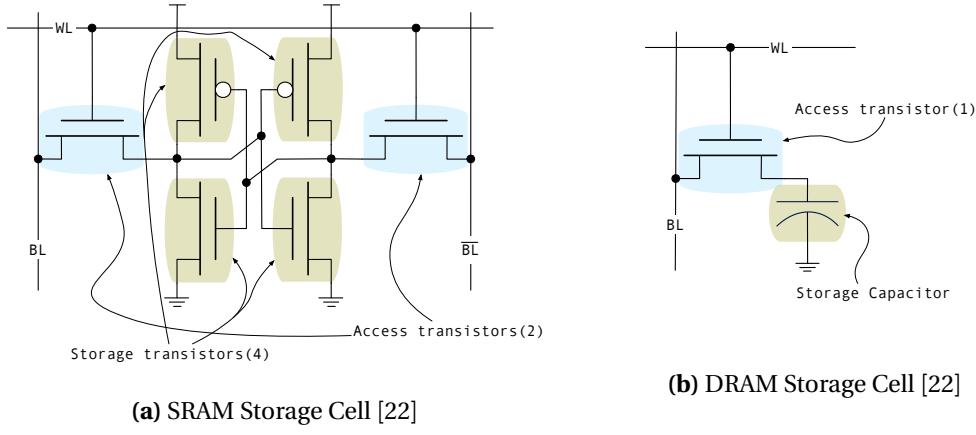


Figure 1.16 RAM Storage Cell Types

is this behavior that makes accessing a DRAM array more complicated than a similar SRAM array.

1.6.1 Accessing DRAM and SRAM

Accessing a typical SRAM involves providing an address and either reading or writing the contents of that location. The read or write often completes in one or two clock cycles depending on whether the SRAM employs internal registers which are used run the SRAM with a faster clock. The storage cell inside the SRAM is formed from cross-coupled transistors (see Figure 1.16a) which latch the contents and hold the contents indefinitely or until power is removed from the device. The storage structure employs six transistors and allows the access logic to be relatively simple and fast but has a relatively low density because of the number of transistors employed.

Accessing a “typical” DRAM is much more involved. It involves opening a page in a bank, reading or writing a portion of the contents of the page then closing the page. The reasons behind this added complexity is the memory cell inside the DRAM which is formed from a capacitor (see Figure 1.16b) which holds a charge reflecting a logic zero or one. When a page(row) of a DRAM memory array is read by the sense amps (see Figure 1.17), the process of sensing the charge on the capacitor causes the capacitor to discharge and lose its contents. To alleviate this problem when the read occurs, the entire contents of the page are transferred to registers, referred to as “page intermediate store” in

Figure 1.17. The process of transferring a page to the intermediate store is known as a “page open”. Once this transfer is complete, portions of the open page can be read similar to reading an SRAM. The problem is that if another read wants to access data that is not in the page, the page has to be closed and another page opened. This involves transferring the previously registered page back to the array to recharge the capacitors in the memory array storage elements. The next page can then be opened and transferred to the page intermediate registers.

In practice, the DRAM protocol is separated into “page” commands and “cache” commands. The page commands open and close pages and the cache commands read and write to the page intermediate store. The process of opening and closing pages is a relatively long time, typically 10-20ns. Once a page is open, accessing the intermediate store is much faster. So if the accesses are somewhat random and different pages are constantly accessed and pages are constantly being opened and closed, the average time to complete reads and writes are very large when compared to SRAM. To alleviate this issue, the DRAM is formed from more than one array of storage elements, between 8 and 32, known as banks. The idea is that while a page from one bank is being accessed, another bank’s page can be opened in preparation for a future read (or write). This access protocol is rather complicated and involves interleaving page commands and cache commands to multiple banks. The system memory controller logic must keep track of which pages are open inside the DRAM and for each memory request must determine if a page needs to be closed and another opened before reading (or writing) the intermediate store. If consecutive accesses are not sequenced carefully, the performance of DRAM can be poor. Some key characteristics of the DRAM used in this work can be seen in Table C.1.

1.6.1.1 Access Locality/Reuse and SRAM as a Cache

In general purpose computing, the sequence of accesses cannot always be controlled, so SRAM is typically used as the first level of memory with DRAM used as the primary storage. Using SRAM as this first level memory is called a cache and acts like a mirror of the DRAM contents. These caches have been used for decades to isolate the computing system from unpredictable access behavior of

the DRAM. The general idea behind caches is that most data exhibits spatial and temporal locality. This “locality” means that when a computer program uses a piece of data in memory, it is very likely that soon after other data “close” to that data will be used and/or the same data will be reused. So when a piece of data in memory is accessed, that data and a large block of data in close proximity to the requested data are transferred to the cache. The cache is designed to hold multiple of these blocks of data often resulting in tens of kB of SRAM. When other memory requests are made, the memory controller checks to see whether the block associated with the requested data is present in the cache. If the requested data is contained in a block present in the cache, this is considered a “hit” and the data in the cache is used. If the requested data’s block is not present, this is known as a “miss” and the slower main memory must be accessed. This access results in another “block” of data being transferred to the cache. If all the blocks in the cache are currently fully employed, one of the blocks must be freed up to make space in the cache for the new block. A block is chosen and transferred back to the main memory and the new block is read and transferred to the cache. To make effective use of the cache, the access behavior of the computer program must exhibit this locality behavior. If the cache blocks are large enough and the program’s access behavior exhibits locality, then employing SRAM is effective and the slower DRAM access times can be somewhat hidden.

As mentioned in Section 1.1 much of the ASIC and ASIP ANN research has focused on taking advantage of the performance and ease of use of SRAM. If the target application’s memory access behavior exhibits some locality and the SRAM cache can be made large enough to avoid high levels of cache misses, then this use of SRAM can be effective. But this work’s target application is such that the access behavior exhibits no or little locality (reuse) and block transfers between the cache and main memory would be constantly occurring. Under these circumstances, **DRAM bandwidth will be the bottleneck**.

This work focuses on using DRAM as the primary storage and managing the accesses to ensure the DRAM is used effectively. With the increased bandwidth achieved from the additional DRAM customizations discussed in Section 1.6.2.1 and 1.6.2.2, this work demonstrates DRAM bandwidths 10X faster than what is available with 2 or 2.5D solutions. This high level of DRAM bandwidth provides

this work the ability to process multiple disparate ANNs at or near real-time while being **significantly faster than state-of-the-art solutions**.

1.6.2 DRAM Customizations

In a typical DRAM, a bank may contain on the order of a few thousand pages and a page may contain on the order of a few thousand bits. Once the page is open, the user accesses a portion of the requested page over a bus. With PCB based DRAMs the bus might vary from four to 16 bits wide, but with 3D DRAMs, such as HBM the bus might be up to 128 bits wide. An ASIC, ASIP or GPU implementation may combine multiple devices to generate bus widths on the order of 1 kbit wide. When using 2.5D technology and HBM with their Pascal™GPU accelerator device, NVidia® achieve a raw DRAM bandwidth approaching 6 Tbit/s [35]². However, experience has shown [16] [23] that usable bandwidth will likely be much lower. Regardless, this existing technology does not achieve the required bandwidth (1.2).

To achieve increased DRAM bandwidth this work is proposing two changes to the Tezzaron®DiRAM4 [14] 3D DRAM. The most significant customization is to widen the databases to generate additional raw bandwidth. This is discussed further in Section 1.6.2.1. With the customizations discussed in Sections 1.6.2.1 and 1.6.2.2, this work demonstrates DRAM bandwidths >10X faster than what is available with 2 or 2.5D solutions.

1.6.2.1 Customization One: Very-Wide Bus

This work achieves the increase in bandwidth by proposing that the DRAM expose more of its currently open page. This means providing access to larger portions of the “Page intermediate store” shown in Figure 1.17. Without the limitations of having to transfer data beyond the chip stack, this work suggests exposing a larger portion of the page over a very wide bus. By staying within the 3D footprint, this bus can be implemented using fine pitch TSVs (see Figure 1.18). This work assumes the DRAM interface protocol uses Double Data Rate (DDR) with a bus width of 2048. Given the DiRAM4 employs

²datasheet also shows a total design power (TDP) of 300 W

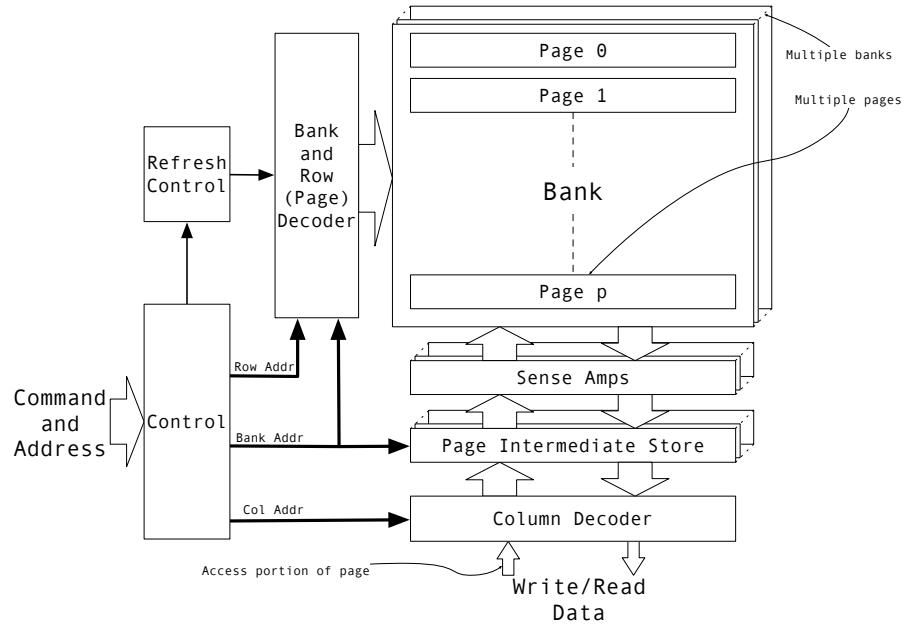


Figure 1.17 Typical DRAM Block Diagram [12]

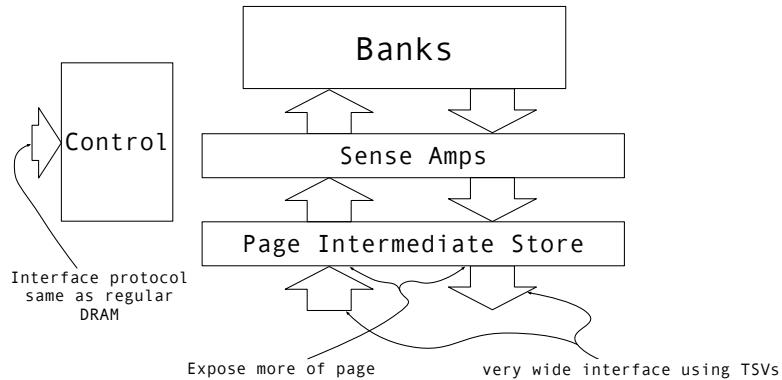


Figure 1.18 Exposing more of the DRAM page

a burst of two for read and write cycles, an entire DiRAM4 page of 4096-bits is accessed during each read or write.

1.6.2.2 Customization Two: Write Mask

When processing an ANN, to compute the activation of an individual ANe involves reading the pre-synaptic ANe activations and the weights of the connections between the pre-synaptic ANes and the

ANe being processed. Typically, groups of up to 32 ANEs are processed. Assuming the state of an ANe is represented by a single-precision floating-point (binary32) number format, the memory used to store the states of the group of ANEs represents only up to 25 percent of a DRAM page. If a partial page needs to be written, the write controller will have to read the page, then modify the page with the new data and write the modified page back to memory. To minimize writing partial pages, the write controller has temporary registers to store two pages of data. The write controller accumulates the ANe states in these temporary registers before writing back to memory. If the write controller receives any data that is not destined for the pages associated with the temporary registers, the contents of one of these page registers must be written back to memory to accommodate the new data. If the register chosen to be written back to memory only contains a partial page of data, the write controller would need to perform a read/modify/write operation. The ratio of reads to writes is high, hundreds or thousands to one so this additional read/modify/write will not adversely affect the performance. However, a proposed customization to the DRAM is the addition of a write data mask to the DRAM write path thus avoiding this read/modify/write. This work assumes single precision floating point for ANN weights and activation, so a mask bit will be provided on a word basis or every 32 bits.

1.7 The Solution

This work assumes that to support all types of disparate ANNs, the system needs to be able to operate directly from the DRAM. Considering DRAM is required to meet the main storage requirements of usefully sized ANNs, if an implementation can ensure the DRAM bandwidth can meet the system requirements, why use SRAM as an intermediate memory and waste the significant silicon area it consumes? The question becomes, can an implementation employ DRAM with minimal SRAM and meet the system requirements?

This work's implementation operates directly out of DRAM, but not just DRAM, 3D-DRAM. This work has designed a system that can stay within the physical footprint of the 3D-DRAM and thus can leverage the benefits of 3DIC. The benefits of 3DIC, which are reviewed in Section 1.5 include reduced energy, reduced area and increased connectivity and bandwidth.

Given the problem description the primary design considerations that drove the architecture of this work are :

- DRAM is required for storage of ANN parameters
- Target applications are unable to take advantage of memory reuse opportunities and therefore not able to achieve high performance using local SRAM
- Target applications will likely apply many disparate ANNs to perform various system functions
- Target applications will have space and power limitations

When performing inference in ANNs, the computational hotspot is the ANe pre-synaptic summation shown in Figure 1.2b and (1.1). This ANe summation involves hundreds or thousands of multiply-accumulates of the pre-synaptic ANe activations and corresponding connection weights. In this work, the ANe activations and weights are stored in DRAM with minimal local SRAM. Therefore, because of the complex access protocol associated with DRAM, one of the main objectives is to demonstrate the 3D-DRAM can be accessed while maintaining the required average bandwidth to the processing elements.

The system has to process thousands of ANes concurrently and do this with minimal unused bus cycles. Therefore, the system must decode instructions, configure the various functions, pre-fetch and pipeline DRAM data and perform the actual activation calculation. To maximize the processing bandwidth, these operations are all performed concurrently enabling this work to demonstrate the ability to meet and exceed the required processing bandwidth as shown in (1.2).

1.8 Novelty

The novelty of this work includes:

- An extensible architecture that can simultaneously process multiple disparate real-time ANNs
 - With low power and real estate demands
- Proposing a custom 3D-DRAM providing a ~32X bandwidth benefit compared to standard 3D-DRAM
 - The 3D-DRAM could be employed in other applications
- An ANN system that operates on very wide data directly out of the customized DRAM
 - Avoiding a dependency on SRAM which would impose ANN size restrictions
- Custom instructions and data structures that facilitate operating directly out of 3D-DRAM
 - Maximizing processing bandwidth by ensuring effective use of the 3D-DRAMs very wide data-bus
 - Instruction format allow system functions to operate concurrently

1.9 Summary

This research explores a 3DIC solution using a custom organized 3D-DRAM in conjunction with unique data structures and custom processing modules to significantly reduce the area and power

footprint of an application that needs to support the processing associated with multiple ANNs. This work's system will provide at or near real-time performance required for systems employing multiple disparate ANNs while staying within acceptable area and power limits and will provide greater than an order of magnitude benefit over comparable solutions.

There will always be questions regarding the suitability of this work's target application, the baseline ANN and the binary32 number format, but providing an extensible architecture should mitigate these. Given different processing and/or number format requirements, with reasonable modifications this work could provide a solution to most ANN system requirements.

Some state-of-the-art implementations are reviewed in Chapter 2. An overview of the proposed system is described in Chapter 3 with more details in Chapter 5. An overview of the instruction architecture is given in Chapter 4. Simulation results are shown in Chapter 6. The conclusion and further work are discussed in Chapter 7.

CHAPTER

2

STATE-OF-THE-ART

For the most part, large scale ANNs have been implemented using GPUs. In some cases, such as CNNs, these GPUs are quite effective when the ANN parameters can be reused in the GPUs local SRAM.

Much of the ASIC and ASIP research has focused on CNNs [2][10][16]. In some cases, implementations have focused on solving specific processing “hot-spots” [10]. Almost all ASIC and ASIP solutions employ arrays of PEs each with local processing capability and local memory. For many of these, the size of the ANN supported is limited by the size of the local memory or they are limited to ANNs that have reuse opportunities such as CNN or have high batch processing opportunities. In some cases the area consumed for local memory can exceed 65% of the processing element die [8][24].

Most implementations accept DRAM is required as the main storage for ANN parameters, but some such as Neurostream [2], TPU [23] and NnSP [15], still load weights and ANe states to local SRAM prior to processing. Others, such as DianNao [11] are moving away from external DRAM toward

EDRAM, thereby acknowledging you need both capacity and DRAM bandwidth. But EDRAM still has capacity and technology availability issues and in the case of [31] still utilizes more than 47% of the silicon area.

In the case of NnSP[15], the paper discusses caching data to bridge the speed gap between external memory and the PE but does not provide details on how to ensure data locality when reading a DRAM cache line and how to minimize the impact of DRAM protocol.

NnSP [15] employs synchronous dynamic random-access memory (SDRAM) but still transfers parameters to a local cache. There is very little detail regarding network size and supported types but the use of cache imply reuse and therefore suggest shared parameter ANNs such as CNNs.

Neuflow[16] is limited to CNNs and the external memory is Quad data rate (QDR) SRAM and thus will be limited by the network size.

NeuroCube[24] and Tetris [17] employ Hybrid Memory Cube (HMC) 3D-DRAM in a 3DIC system. The memory data is transferred from the DRAM to a PE. The PE provides local SRAM that allows NeuroCube[24] to provide higher performance for shared weight ANNs or batch processing. The overall bandwidth is still limited by the HMC interface bandwidth.

Eyeriss[10] focuses on CNNs and specifically on the convolution “hot-spots.” It does not support the pooling operations although a local CPU can support these. However, it does not support the memory intensive fully-connected classifier layers or non-weight-shared locally-connected layers [29][42]. Eyeriss cannot be effectively applied to locally-connected ANNs such as Deepface [42].

The DianNao family of ASICs [8] [11] originally used external DRAM to store ANN parameters but still uses direct memory access (DMA) along with SRAM as an intermediate store. However, the recent versions of the ASIC acknowledge that local SRAM limits the size of the ANN the device is able to support and have moved from external DRAM to EDRAM [31]. However, the local EDRAM is still limited to 36 MB and therefore requires multiple instances to support DNNs similar to the types of DNN this work is targeting. To support these useful DNNs, they have to instantiate up to 64 devices to accommodate the required DNN parameter storage resulting in a power approaching 1 kW and an area of 430 mm². These multi-instance implementations can be used to support batch processing

[31] but not the single input systems this work is targeting.

The Neurostream[2] ASIP uses 3DIC along with HMC 3D-DRAM. The Neurostream[2] acknowledges that DRAM is required to support useful DNNs but again data is transferred from the DRAM to local SRAM prior to processing. The local SRAM allows Neurostream[2] to provide high performance for shared weight ANNs such as CNNs but the performance of ANNs without reuse opportunities are limited by the HMC interface bandwidth.

The Google TPU [23] utilizes a large local 24 MB SRAM along with a 256x256 systolic array and a 30 GB/s external DRAM interface. It gains performance by storing parameters within the array and by performing large batch processing but acknowledges that it is bandwidth limited when implementing fully-connected ANNs. It also states that their experience of implementing ANNs in the Google server farms suggests that these fully connected ANNs represent the bulk of their processing requirements. The simpler CNNs only represent 5% of the servers ANN processing requirements. It should also be stated that [23] suggests that GPU solutions cannot reach the required performance targets.

CHAPTER

3

SYSTEM OVERVIEW

As mentioned in Section 1.4, this work’s target application implements multiple ANNs, each of which have limited opportunities for both weight reuse and batch processing. These applications require DRAM to be employed for main storage of ANN parameters and local SRAM is of limited use. Under these conditions, the DRAM bandwidth is the system bottleneck.

This work proposes employing 3DIC technology along with a customized 3D-DRAM with a very wide data-bus to meet the system requirements. This work is able to maintain a significantly higher bandwidth over existing ASIC or ASIP solutions by physically staying within the 3DIC footprint and taking advantage of high density TSVs. The objective is to demonstrate that a pure 3DIC system can implement multiple disparate ANNs within reasonable power and area constraints. The 3DIC system die stack (figure 3.1) includes the 3D-DRAM with a system manager below and one or more processing layers below the manager.

3D-DRAM has recently become available in standards such as HBM and HMC and proprietary

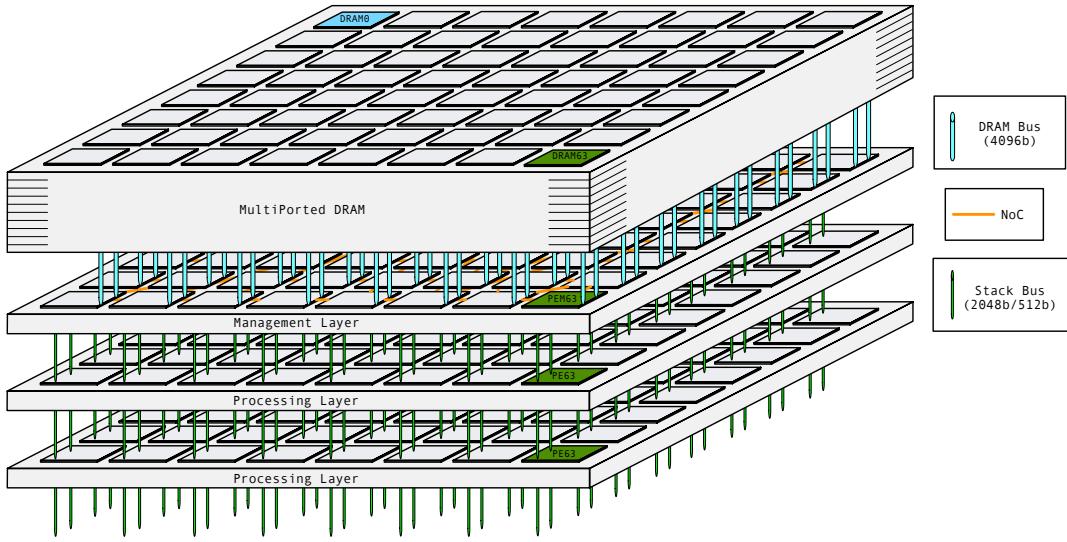


Figure 3.1 3DIC System Stack

devices such as the DiRAM4 available from Tezzaron. These technologies provide high capacity within a small footprint. In the case of HMC, HBM and DiRAM4 [14], the technology can be combined with additional custom layers to provide a system solution.

The question becomes, can a useful system coexist within the same 3D footprint?

This work targeted a baseline system with:

- Computations requiring binary32
- Tezzaron Dis-Integrated 3D DRAM (DiRAM4) [14] for main memory
- 28nm ASIC technology

The work includes customizing the interface to a 3D-DRAM, researching data structures to describe storage of ANN parameters, designing a memory manager with unique micro-coded instructions and a PE layer. The targeted 3D-DRAM, the Tezzaron DiRAM4 employs 64 disjoint memories arranged in a physical array.

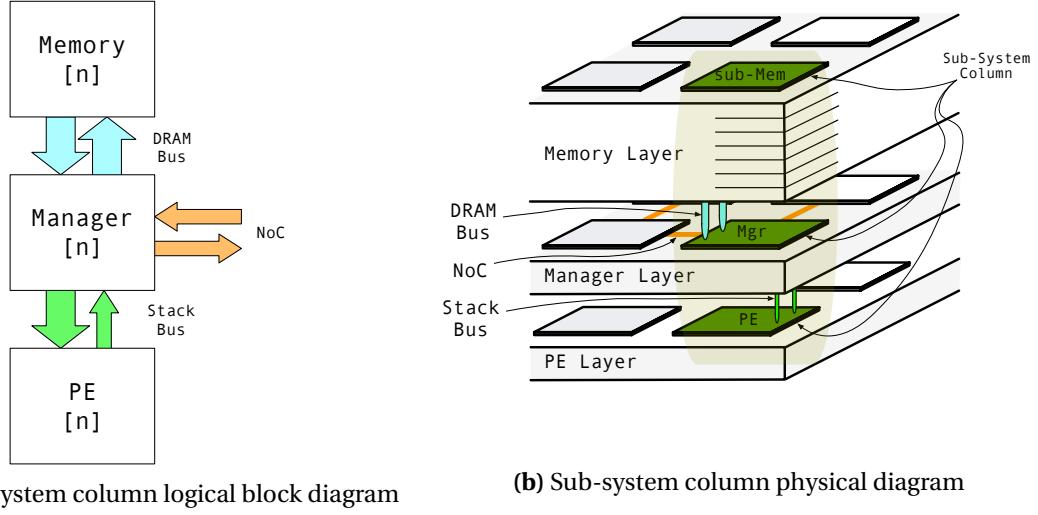


Figure 3.2 Sub-System Column (SSC)

3.1 Sub-System Column

The overall system is constructed from an array of sub-systems known as a Sub-System Column (SSC) that combines the manager logic, DRAM and PE logic. The various steps to process an ANN are provided in the form of instructions (see section 4.1). The instructions are downloaded by a host system to instruction memory residing in the manager. The manager decodes these instructions and based on the instruction contents is responsible for coordinating SSC operations and managing the reading and writing of DRAM. With the processing of an ANe, the manager reads pre-synaptic states and connection weights from the DRAM, provides that data to the PE which operates on data and returns any results back to the manager. There are other instructions that specifically deal with coordination between SSCs but the main workload is the processing of the ANes.

To allow future modifications to support different number formats, different PE configurations and to allow additional features to be implemented without a significant increase in logic area, the SSC has been designed for extensibility. The SSC is designed to operate on one of the disjoint sub-memories within the DiRAM4 (see Figure 3.4). As shown in Figure 3.2, the SSC includes the DiRAM4 sub-memory (referred to as the SSC memory), a manager module and a PE module.

The customizations described in Section 1.6.2.1 means each SSC memory provides the manager with a 2048-bit DDR data bus. The DiRAM4 has 64 sub-memories so there are 64 SSCs. The SSC has been designed as a standalone unit and does not have a direct knowledge of the other SSCs in the system and any required coordination between SSCs is embedded within an instruction rather than in the hardware.

To process an ANN, the user must allocate the individual ANes to an SSC. Once partitioned, an ANN's pre-synaptic connection weights must be stored in the SSC memory associated with the ANe. The states of the pre-synaptic ANes must also be stored in a dependent ANe's allocated SSC memory. Details on where the parameters and ANe states are stored are described in Chapter 4.

The baseline SSC is designed with 32 execution lanes, each of which can simultaneously process two streams of binary32 words. The customized DiRAM4 provides a 2048-bit cache line at the system clock rate to the manager. This 2048-bit bus is sub-divided into 64 binary32 words. The manager layer reads the 2048-bit cache line from the DRAM, multiplexes the 64 words onto 32 execution lanes with two words per execution lane and passes the execution lane data to the PE layer. The primary function is to direct a pre-synaptic ANe state and connection weight to the two words in an execution lane where they are multiplied and accumulated by the PE layer.

The manager layer has individual stream read memory controllers to allow the individual streams in the execution lanes to operate independently. In this way, the ANe states and the connection weights can be stored in different configurations depending on the ANN type and ANN partitioning. The 32 execution lanes can be used for processing a group of one to 32 individual ANes or for processing one ANe.

3.2 Processing a group of ANes

In the case of a group of ANes, all the ANes must be associated with a ROI or a fully-connected layer. The pre-synaptic ROI, or all the ANes in the case of a fully-connected layer, are broadcast to one of the streams of all execution lanes. The other stream is used for the individual ANe's connection weights. The PE Streaming Operation block (StOp) performs 32 simultaneous floating-point (FP)

multiply-accumulate (MAC) operations while the data is being streamed from the manager layer to the PE layer. Once all the ANe states and weights have been streamed to the PE, the StOp passes the result of the MAC to the PE Single-Instruction Multiple-Data (SIMD) that then applies the activation function. Typically the SIMD then sends states of the group of ANes back to the manager.

Considering the ROI pre-synaptic ANe states and weights for k th ANe as arrays $[A]$ and $[W_k]$ with elements A_p and $W_{k,p}$ respectively, the state of the ANe is the dot-product of the two arrays followed by the activation function (3.1).

Figure 3.3a shows how the group's weights and states are directed to execution lanes.

$$\text{State of } k\text{th ANe, } S_k = f([W] \cdot [A]) \quad (3.1)$$

$$\text{where } \mathbf{W} = [w_{k,0}, w_{k,1}, \dots, w_{k,n}], \quad \mathbf{A} = [a_0, a_1, \dots, a_n]$$

and n is the pre-synaptic fanin

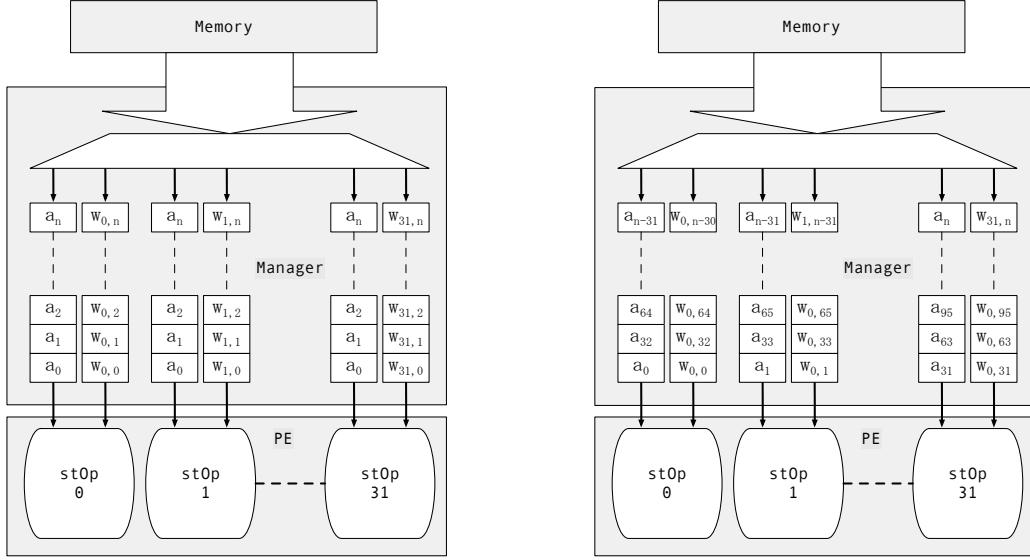
3.3 Processing a single ANe

In the case of a single ANe, the pre-synaptic ANe states are vectored across one of the streams of all execution lanes. The connection weights are vectored across the other stream of all the execution lanes. How the weights and states are directed to execution lanes is shown in Figure 3.3b.

3.4 Sub-System Column (SSC) Blocks

3.4.1 Customized DRAM : Dis-Integrated 3D DRAM (DiRAM4)

The DiRAM4 [14] employs multiple memory array layers in conjunction with a control and IO layer. The memory is formed from 64 disjoint sub-memories each providing upwards of 1 Gbit with a total capacity of at least 64 Gbit. Unlike traditional DRAM, the DiRAM4 has two independent channels that are accessed using DDR signaling on the control signals. Each channel has 32 banks and 4096 pages



(a) Multiplexing DRAM data for a ANe group
 $k \in 0 \dots 31$

(b) Multiplexing DRAM data for a single ANe
 $k = 0$

Figure 3.3 Multiplexing DRAM data to execution lanes

per bank with 4096 bit/page.

The standard DiRAM4 has a 32-bit read databus and a 32-bit write databus enabling simultaneous read and write. Both read and write databuses employ DDR signaling. The read and write transactions are burst-of-two providing 64 bits per read/write. When accessing a DRAM, a read and write are often referred to as a cache line. The device is designed to operate at 1 GHz although this work targeted a 500 MHz clock frequency.

This work is proposing customizations to the DiRAM4 which are outlined in Section 1.6. One of these proposed changes is to widen the read and write databuses to 2048-bits. Using the same burst-of-two means each read and write will access an entire page. A cache line becomes 4096-bits. Another proposed change is to add mask bits to the write databus to avoid having to perform read/modify/writes when writing back data much smaller than the new large cache line.

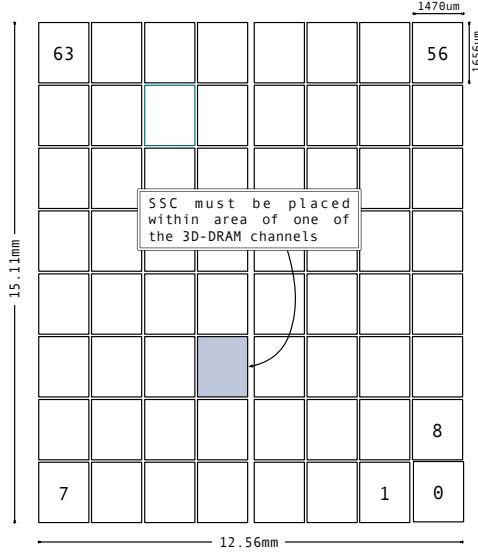


Figure 3.4 DRAM Physical interface layout showing area for SSC

3.4.2 Layer Interconnect

To provide high connection density, high bandwidth and low energy, the layers are connected using fine-pitch through-silicon-vias (TSVs). We can take advantage of the area and bandwidth benefits provided by the TSVs by ensuring the system stays within the 3D footprint. The high density interconnect provided by TSVs allows the system to take advantage of the very wide DRAM bus provided by the DRAM customization described in Section 1.6.2.1. The wide interconnect between the manager and PE is also implemented using TSVs. The interconnect between the manager and PE is referred to as the stack bus. The interconnect between the manager and DRAM is referred to as the DRAM bus.

3.4.3 Stack Bus

The stack bus is bidirectional with a 36-bit Out of Band (OOB) configuration bus from the manager to the PE, a 2048-bit “downstream” data bus from the manager to the PE and a 512-bit “upstream” data bus from the PE to the manager.

The 36-bit OOB bus is designed to send configuration packets to configure the StOp and SIMD blocks inside the PE. A configuration packet primarily contains the StOp function to be performed on

the downstream data and the operation the SIMD is to perform on the result from the StOp. It also includes the size of the downstream data stream and an operation identification tag.

The 2048-bit downstream stack bus is designed to carry 32 execution lanes of data with each execution lane containing two operand buses. The two operand buses are designed to carry streams of binary32 numbers. This allows the downstream stack bus to simultaneously stream 32 execution lanes, each with two 32-bit argument streams. Typically these streams are the weights and pre-synaptic ANe states to calculate the states of up to 32 ANes. The streams can also be configured to calculate the state of a single ANe where the weights and pre-synaptic ANe states for the single ANe are sent in parallel across the downstream stack bus and a reduction operation can be performed in the PE by the StOp and SIMD.

The 128-bit upstream bus is primarily designed to send the results of a downstream operation. Typically this is the state of up to 32 ANes. The upstream bus is packetized with the result data contained in the data portion of the upstream packet. Additional information includes the operation identification tag provided by the decoder in the downstream OOB operation configuration and the number of data words. The upstream bus is not as wide as the downstream bus because the ratio of downstream operands to result data is a function of the pre-synaptic fanin. Based on the average fanin from the baseline ANN shown in Table 1.1, a 512-bit bus exceeds the required average bandwidth. The stack bus OOB downstream and upstream signaling can be seen in Figure 3.5.

3.4.3.1 Common Bus Signaling

With almost all interfaces in this system, additional control signals are used to a) validate, b) delineate and c) flow control messaging between blocks. This “common” bus protocol signal group includes three signals, VALID, CNTL and READY. The single VALID signal is high when the bus contains valid information. The two-bit CNTL signal is used to delineate by encoding start of message (SOM), middle of message (MOM) and end of message (EOM). Because of the asynchronous nature of the system, most interfaces employ small first-in first-out queues (FIFOs). When these FIFOs are almost full, the source is flow controlled by the READY signal. The point at which the FIFO signals the source to stop

sending is based on the latency of the particular interface. The common signaling protocol will be seen in all waveform diagrams as seen in Figure 3.5.

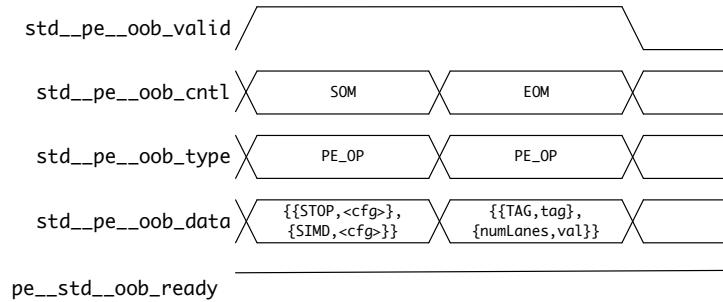
3.4.4 DRAM Bus

The interface to the DiRAM4 is similar to traditional DRAM interfaces with control signals including bank address and multiplexed page/cache address bus. There are separate 2048-bit DDR read and write databases (see section 1.6.2.1). In total there are 4180 signals in the DRAM interface. Other than the wide databases, the interface protocol is as described in [14]. A timing diagram showing a read and write to the DiRAM4 is shown in Figure 3.6.

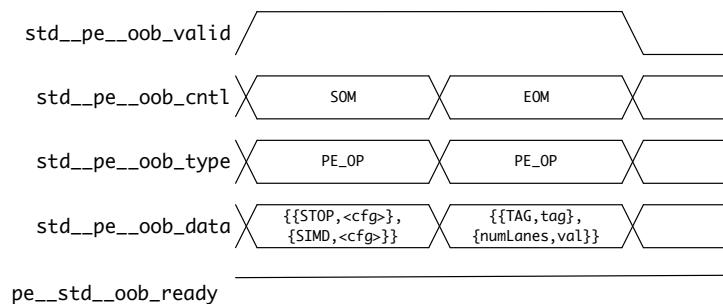
The bandwidth of the DRAM bus is designed to ensure data can be constantly maintained to the stack downstream bus. Because an entire cache line is accessed for each read request, there is an extreme case when back-to-back requests result in up to 32 DRAM page open commands. It is possible that this sequence of page opens could be followed by page closes resulting in a period when no useful data is being read from the DiRAM4. This case is shown in Figure 3.7. To accommodate this, the DRAM bus has twice the raw bandwidth of the downstream stack bus. Therefore under this worst case scenario the higher bandwidth data from the DiRAM4 must be buffered, as shown in Figure 3.8a. These per channel 32 kbit FIFOs are placed in the read path as shown in Figure 3.8. These FIFOs are instantiated in the manager MRCs as described in Section 5.1.4. A question arises that if the DRAM interface has to employ large local storage in the form of FIFOs, why is this different from other implementations that employ large amounts of SRAM? The difference is that this FIFO storage is to accommodate the DRAM interface protocol and pipelining to allow concurrent processing and does not impose a ANN size limitation unlike systems that employ the storage in a cache-like structure.

3.4.5 Manager Layer

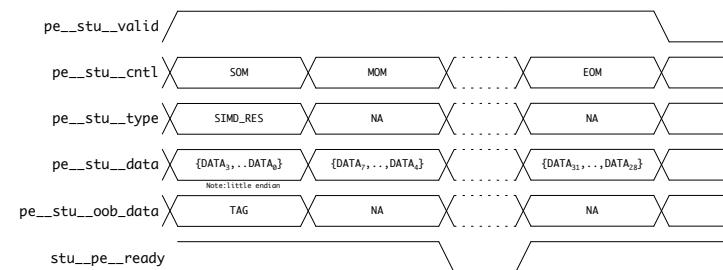
The Manager block is the main SSC controller and the memory controller for the DiRAM4 memory. The operations required to process an ANN are formed from individual instructions that are decoded by the Manager. These instructions (for more detail see section 4.1) are sub-divided into descriptors



(a) Stack downstream OOB Bus



(b) Stack downstream Data Bus



(c) Stack upstream Bus

Figure 3.5 Stack Bus signaling

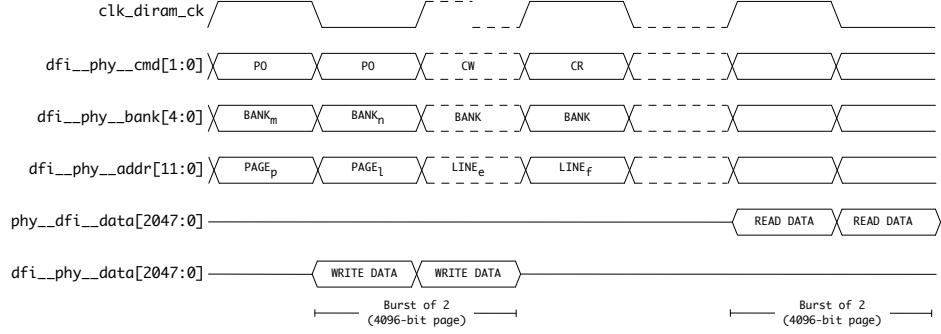


Figure 3.6 Read and Write request to DiRAM4 [14]

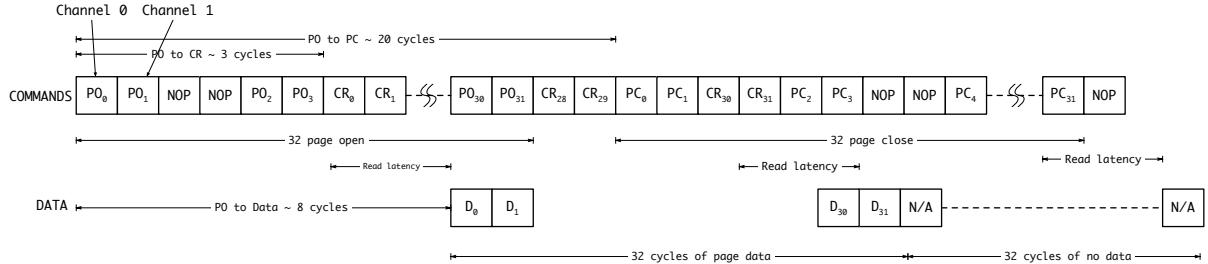
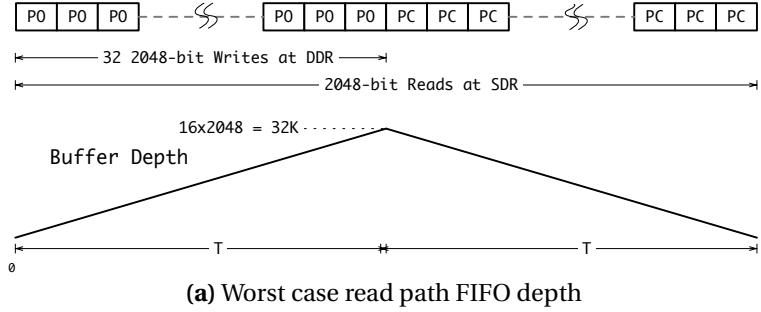


Figure 3.7 Worst case PO/PC sequence

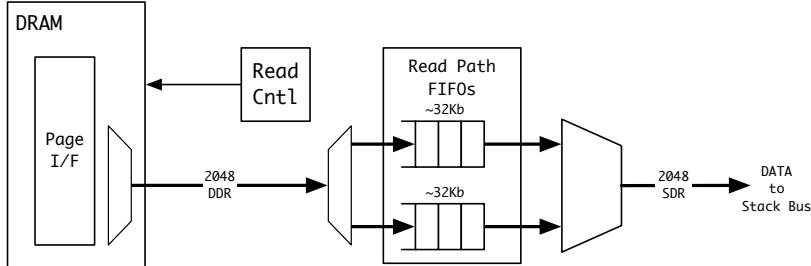
to describe memory read operations, processing engine operations, memory write operations and general system operations for synchronization. The manager reads these system instructions from an instruction memory, decodes the descriptors and configures the various blocks in the system. The configuration includes:

- Initiating operand reads from DRAM
- Preparing the processing engine (PE) to operate on the downstream stack bus operand streams
- Preparing the result processing engine to take the resulting ANe activations from the PE via the upstream stack bus and write those results back to the DRAM
- Replicating the resulting ANe activations to neighbor managers over the NoC for processing of other ANN layers

The most common instruction is to perform state calculation on a group of ANes. This instruction contains four descriptors (see Figures 3.9 and 4.3), an operation descriptor, two memory read



(a) Worst case read path FIFO depth



(b) Read path buffering diagram

Figure 3.8 DRAM Read Path Buffering

Operation descriptor	Arg0 read descriptor	Arg1 read descriptor	Result write descriptor
----------------------	----------------------	----------------------	-------------------------

Figure 3.9 Four descriptor instruction (4-tuple)

descriptors and a memory write descriptor.

The operation descriptor describes the operations the PE will perform, which typically includes a MAC to be performed by the StOp followed by a ReLu [32] function to be performed by the SIMD. The manager extracts the operation information, embeds the information in an OOB packet and sends the packet to the PE over the downstream OOB bus. The two memory read descriptors are used to define the memory locations of the downstream stack data buses. There is one memory read descriptor for each of the two-operand streams in the execution lanes. One typically defines where the pre-synaptic weights are stored and the other defines where the pre-synaptic ANe states are stored. The architecture is designed to cause an instruction to compute the state of multiple ANes or to cause an individual ANe's state to be computed. If a group of ANes is computed, the pre-synaptic connection weights

for the ANes are stored interleaved and when read from DRAM are directed to one of the streams of each of the execution lanes. If a single ANe is computed, the pre-synaptic connection weights for the ANes are stored linearly and when read from DRAM are directed to one of the streams of each of the execution lanes. The pre-synaptic ANe states are stored in row-major order and when read are broadcast to the other stream of each execution lane.

3.4.6 Processing Engine Layer

The PE contains two main processing modules, the StOp and the SIMD block. Both the StOp block and the SIMD have 32-execution lanes. The execution lanes inside the StOp contain functions required to perform ANe related operations. The SIMD will be a 32 execution lane, 32-bit SIMD engine based on the device described in [39].

The PE is configured by the manager to perform operations on two operand data streams from the manager. The StOp is able to operate on this data directly from the manager at the full bandwidth rate of the stack bus so it does not have to be stored in local SRAM prior to processing. There is a small FIFO to provide buffering to allow asynchronous configuration of the StOp block and the source of the streaming data in the manager. The FIFO also allows the two argument streams in each of the execution lanes to wander with respect to each other and with respect to the other lanes.

The architecture is expandable to allow various functions to be provided in the StOp. The current baseline implementation includes the MAC operation. There is one MAC per execution lane allowing up to 32 simultaneous pre-synaptic ANe weight · state computations on the two operands from the two streams in each execution lane. These computations can be for a group of up to 32 ANes or for a single ANe as shown in Figure 3.10.

If it is for a group of ANes, the SIMD only has to perform the activation function on the 32 results from the StOp. If a single ANe is being processed, the SIMD must accumulate the result from each execution lane's StOp before applying the activation function. The activation function currently implemented is the ReLu. The ANe states can be sent back to the manager over the stack upstream bus or retained for further processing such as pooling or softmax calculations.

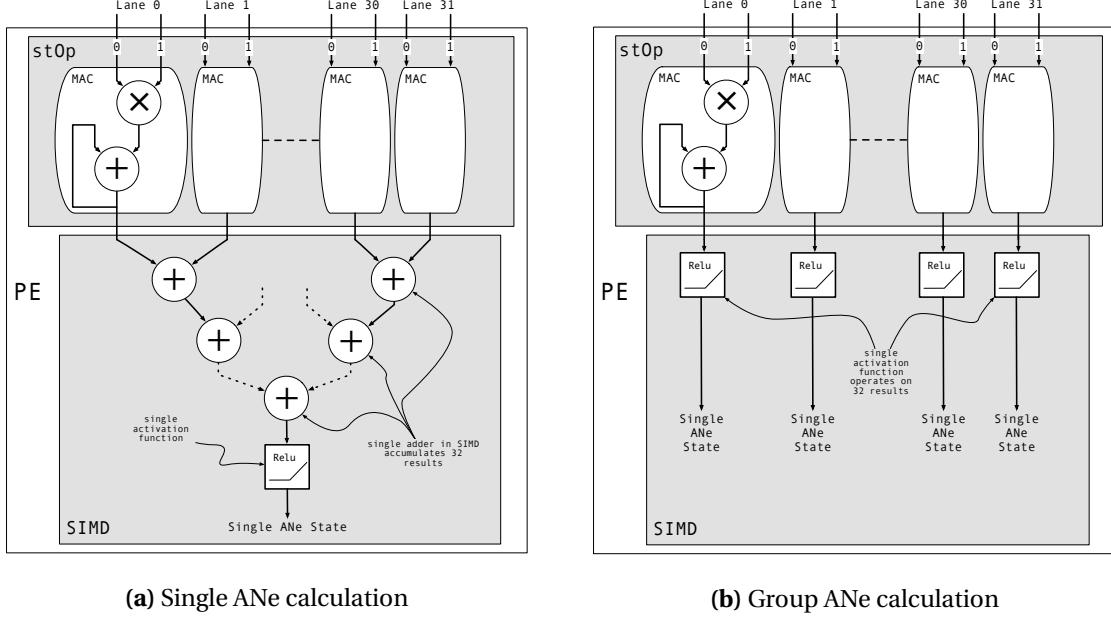


Figure 3.10 PE ANe calculation

3.4.7 Inter-Manager Communication

An NoC provides host to SSC and SSC to SSC communication. A four port NoC within each management block is used to communicate with the host and other SSCs using a mesh network as shown in Figure 3.11. The NoC packet format can be seen in Figure 3.12.

The managers in the middle of the array use all four ports to connect to adjacent managers, the managers in the corners of the array connect to two adjacent managers and the managers at the edges connect to three adjacent managers, as shown in Figure 3.11. The host is connected to one (or more) of the managers at one edge of the array.

During configuration and/or computations, it may be required to replicate data to other SSCs. During ANe computations, an SSC only reads ANN weights and states from its local DRAM and not DRAM of other SSCs. In some cases, such as fully-connected layers and locally-connected layers with ROI overlap, the ANe computation in an SSC for layer n may include ANe states from layer $n - 1$ that were computed in another SSC. In this case, when ANes are being computed for layer $n - 1$,

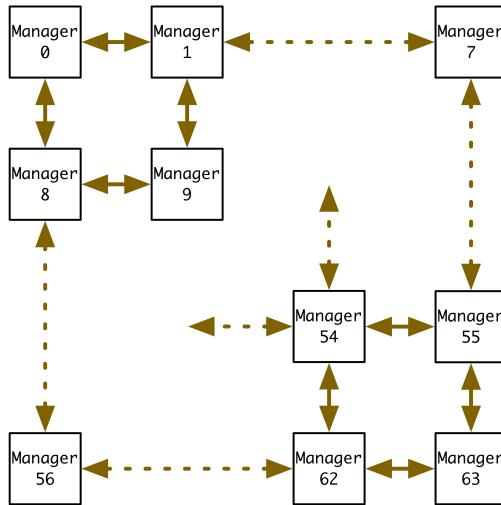
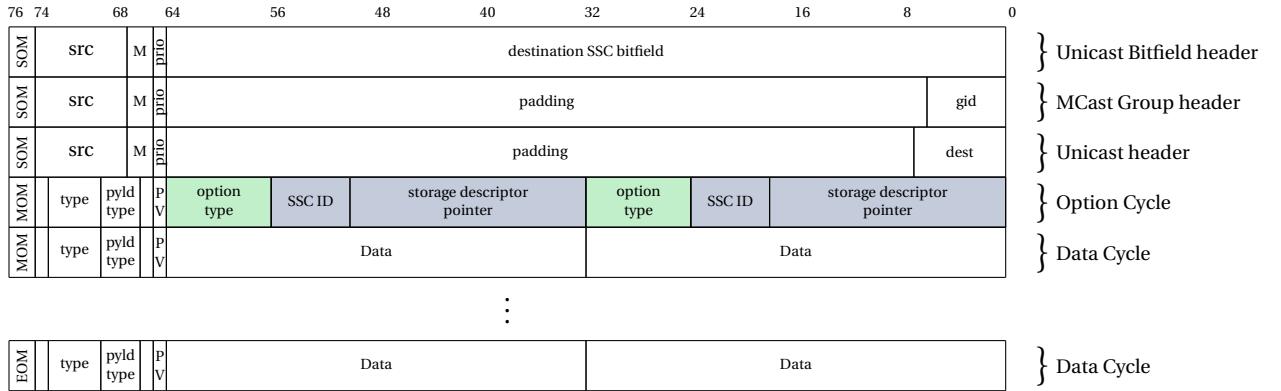


Figure 3.11 NoC manager connectivity

the operation instruction contains information as to where the result should be written back for computing ANes in layer n . If a particular layer $n - 1$ ANe is required by another SSC when computing a layer n ANe, the result is sent to all dependent SSCs via the NoC. The instruction contains one or more storage descriptors (see section 4.1.2.2) that identify the destination of the operation result. If the result must be replicated, there are multiple storage descriptors. When the PE returns the result, the manager examines the storage descriptors and determines to which SSC the result should be sent. The manager creates an NoC header that includes information on all the destination SSCs. This is encoded as a 64-bit field, although to provide extensibility the NoC header also supports a unicast address and multicast group. The multiple storage descriptors and the result data are placed in the data portion of the NoC packet. As the packet traverses the NoC, it is replicated to outgoing ports based on the destination bit field. When the destination SSC receives the packet, it extracts the storage descriptor and writes the data to its local DRAM.

When computing an ANN, a group of SSCs will be assigned to the ANN and the individual ANes will be distributed over the group of SSCs. The NoC is employed to replicate ANe states from one SSC to other SSCs in the group. It is anticipated that all the SSCs in the group will be within four NoC hops of each other. When the PE passes the ANe states back to the manager, the results may be written to



transport the result packets in approximately two hundred clocks. The NoC packet shown Figure 3.12 contains a header cycle, 16 data cycles and one option cycle. Assuming a group of eight SSCs have been assigned to an ANN, a result will need to be replicated to up to seven other SSCs. This will require four options cycles to carry the seven storage descriptors resulting in an NoC packet size of 21 cycles. The verification identified the system introduces an inter-frame gap of two cycles which means it takes on average 23 cycles to transport a packet between SSCs. Because of the store-and-forward behavior of the NoC design, when a group of ANe states needs to be replicated to all other SSCs, it takes six frame times in the network for all the packets to reach their destinations requiring a total of 138 cycles, which is within the 200 cycle requirement.

Based on the seven frame times to transport the packets to all destination SSCs, this means the NoC has an effective bandwidth of 30 Gbit/s (3.2). The raw bandwidth of each NoC port is 32 Gbit/s (3.3) and the raw mesh bandwidth is 256 Gbit/s (3.4).

$$\text{NoC effective bandwidth} = \frac{N_p \cdot W_p \cdot b_w}{N_f \cdot c_f \cdot t_c} = \frac{8 \cdot 32 \cdot 32}{6 \cdot 23 \cdot 2 \times 10^{-9}} \approx 30 \text{ Gbit/s} \quad (3.2)$$

$$\text{NoC port bandwidth} = \frac{b_i}{t_c} = \frac{64}{2 \times 10^{-9}} = 32 \text{ Gbit/s} \quad (3.3)$$

$$\text{NoC mesh bandwidth} = \frac{N_s \cdot b_i}{t_c} = \frac{8 \cdot 64}{2 \times 10^{-9}} = 256 \text{ Gbit/s} \quad (3.4)$$

where

N_p	is the total number of SSC packets
W_p	is the number of words per packet
N_f	is the number of frame times
C_f	is the number of cycles per frame
b_i	is the number of data bits per interface
t_c	is the cycle time

3.4.7.1 NoC Future Work

The NoC designed in this work was primarily to create a representative NoC for the area and power analysis. There are situations based on poor SSC assignment where excessive frame hops are required

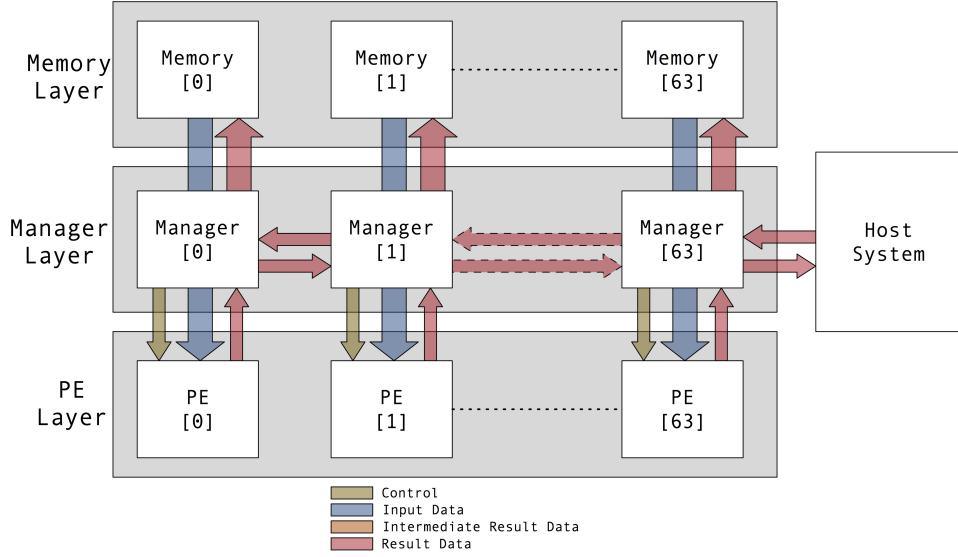


Figure 3.13 System Flow Diagram

and could potentially violate the maximum transport time. Each NoC has a forwarding table indicating which port should be used to send to a destination SSC or the host. The forwarding table has a default setting based on shortest path but it is anticipated that it be dynamically reconfigured by the host. It is anticipated that the forwarding table configuration will be based on the ANN SSCs assignments to provide more optimum routing given the ANe state replication associated with a particular operation. An analysis of the hop usage will likely suggest a more optimal routing table rather than the fixed shortest path currently employed. This will be left to future work.

3.5 Summary

A control and data flow diagram of the stack showing the 64 sub-system columns can be seen in Figure 3.13.

One of the primary objectives is to ensure the system can maintain the required average bandwidth (see Table 1.2) while operating directly out of DRAM over a range of pre-synaptic fanins. To achieve this the system decodes instructions and concurrently sends configuration information to various system functions. It concurrently pre-fetches and pipelines data to absorb the latencies associated

with DRAM. All system functions pipeline their configuration data to ensure the main processing pipeline is not starved of data and/or operations. This parallelism allows this system to constantly stream data while results from previous operations are being processed, broadcast to other SSCs and written back to SSC memory. The instructions, the structures for describing the operations and the structures describing how data is retrieved and stored have been designed to provide extensibility. A detailed description of the instruction architecture and data structures is given in Chapter 4. The baseline ANN described in Table 1.1 was used to define a collection of pre-synaptic fanin tests and those tests were used to ensure the average bandwidth can be maintained. The results of those tests can be seen in Chapter 6.

The second objective is to determine if an extensible 3DIC system can be designed employing a customized 3D-DRAM. The 3D-DRAM employed is the DiRAM4 and estimated areas are extracted from data provided by Tezzaron® [36]. The feasibility of the customizations has been verified with consultation with Tezzaron®. The physical details for the register-transfer level (RTL) design of the primary modules are given in Chapter 5. The overall area details with respect to the 3DIC stack are shown in Chapter 6.

CHAPTER

4

SYSTEM OPERATIONS

As mentioned in Chapter 3, an SSC has 32 execution lanes allowing the simultaneous processing of up to 32 ANes. When processing a group of ANes the basic operations to determine their states are:

- Manager streams the states of the pre-synaptic ANes to the PE
- Manager streams the weights of the pre-synaptic connections to the PE
- Each execution lane in the PE operates directly on the two argument streams using the StOp block
- The PE SIMD block takes the 32 results from the StOp block and performs the activation function to generate the ANe states
- The PE SIMD packetizes the ANe states and sends the packet to the manager
- The manager replicates the ANe state data over the NoC to any dependent SSCs

- If the local SSC is dependent on the result, the manager saves the ANe state data in local SSC DRAM

Although the primary focus of this work is effective use of a novel 3D-DRAM along with an expansive system to process ANNs, a complete system needs to also provide support features for tasks such as downloading the ANN parameters to memory from a host system and a host system also needs to download new inputs and upload ANN outputs. This work has created a system and an infrastructure to include these support tasks. Although not all of the features required for an actual final product have been included in this work's design and verification effort, the infrastructure required to easily add the missing features has been provided. This work has developed an instruction architecture to support and describe the operations associated with the processing of an ANN along with the support tasks required when employing this work in a product implementation.

4.1 Instructions

An instruction is coarse grained in as much as it provides the information to perform all the operations associated with a high level task. The manager is responsible for instruction decode and coordinating the various data flows and configuration of the modules throughout the SSC. The PE is responsible for the main algorithm operations using a combination of its StOp and SIMD blocks. The information in the instruction provides the information to control these finer grained functions within the manager and the PE along with the various system support tasks.

To provide the fine grained information, an instruction is partitioned into sub-instructions called descriptors. An instruction can contain one or more descriptors and each of these descriptors contains the information to control a specific operation(s) within the SSC to perform the high level task. For example, to process a group of ANes, the instruction contains a descriptor to communicate where the pre-synaptic ANe states are stored, another to communicate where the connection weights are stored, another describes what activation function should be used and another where in memory the resulting ANe states should be stored.

4.1.1 Instruction Types

There are two instruction types currently defined, a **configuration** instruction and a **compute** instruction. The configuration instruction has been defined to deal with synchronization and data downloads and uploads which includes ANN parameters and input, SSC instructions, SIMD and StOp operation pointers and SIMD instructions. The compute instruction has been defined to deal with computing the states of a group of ANes. Although this work's focus has been on the compute instruction – as it has the most influence on system performance – configuration instructions have been defined and tested to provide an extensible system.

A generic instruction is an n-tuple where the tuple elements are descriptors and the number of descriptors can vary based on the high level task being performed. These descriptors are decoded and used to configure the various functions in the SSC that will take part in completing the instruction. The contents of a single descriptor may be sent to multiple functions and in some cases the manager doesn't even parse the contents of the descriptor but immediately passes it to a dependent function. This allows the system to concurrently prepare for the tasks involved with the instruction.

4.1.2 Compute Instruction

The compute instruction typically contains four descriptors for configuring the tasks associated with processing a group of ANes. The instruction can be seen in Figure 4.1 and includes:

- Operation descriptor containing:
 - StOp operation
 - SIMD operation
 - Number of active lanes
 - Operand Vector length
- Two memory read descriptors containing:

Operation descriptor (MAC+ReLU)	Read descriptor (Input ROI)	Read descriptor (pre-synaptic ANe states)	Write descriptor (store ANe states)
------------------------------------	--------------------------------	--	--

Figure 4.1 Typical compute instruction (4-tuple)

- Addresses for the pre-synaptic ANe states and connection weights for the two argument streams to the PE
- Read data to execution lane multiplex method (broadcast/vectored)
- Memory write descriptor containing:
 - DRAM address for ANe states

The descriptor also employs an n-tuple format where the elements contain configuration options required by the operation. The option elements within a descriptor are a two-tuple with option and associated value and are referred to as option tuples. These option tuples include a type and value, which contain information such as storage descriptor pointer (see Section 4.1.2.2), PE operations and the number of operands. The length of the value field is currently eight bits or 24-bits. The 24-bit value field is referred to as an extended tuple and is currently used for memory address, number of operands and configuration options. In Figure 4.2 we see the format of a 5-tuple operation descriptor and a list of option types is shown in Table 4.1.

To pull it all together, Figure 4.3 demonstrates a four-tuple compute instruction with details shown for each of the descriptors. Figure 4.3a shows the compute instruction that contains three 4-tuple and one 5-tuple descriptors. The memory write descriptor shows two storage option elements, which indicates the resulting ANe states need to be saved in the memory of two SSCs. The waveform in Figure 4.3b is from the verification environment and shows the instruction as it is read out of the manager's instruction memory. The waveform also shows an example of the common interface signaling described in Section with the signals `wum_wud_icnt1` and `wum_wud_dcnt1` being used to delineate the instruction and descriptors respectively. As can be seen in Figure 4.3b, the instruction memory transfers three descriptor elements per cycle shown on the bus signals `wum_wud_option_type`

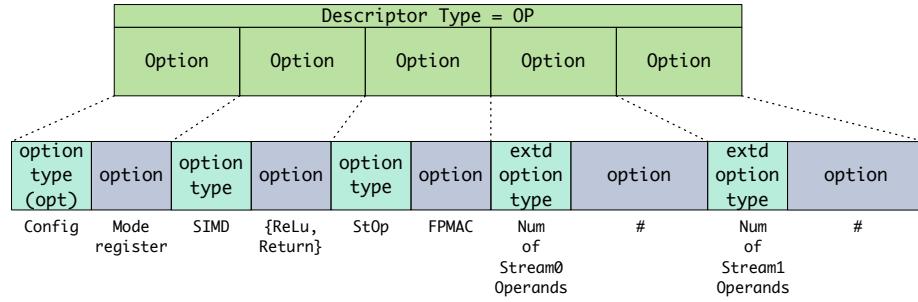


Figure 4.2 Operation descriptor (5-tuple example)

Source			
Type	Type Code	extd	Value Description
NOP	0	N	no operation
source	1	N	Used to define the source of any data, such as memory or PE
target	2	N	Used to define the target for any data, such as memory or PE
transfer type	3	N	How data will be directed, vector or broadcast
number of lanes	4	N	number of active execution lanes used in operation
StOp pointer	5	N	pointer to PE StOp operation table in PE controller
SIMD pointer	6	N	pointer to PE SIMD instruction memory
Memory storage descriptor	7	Y	pointer to storage descriptor used in memory read or write
num of arg0 operands	8	Y	number of operands sent to execution lane stream 0
num of arg1 operands	9	Y	number of operands sent to execution lane stream 1
num of return packets	10	Y	number of response packets generated by PE
config sync	11	Y	Synchronization
config data	12	Y	Data transfers
status	13	Y	status information

Table 4.1 Option tuple functions

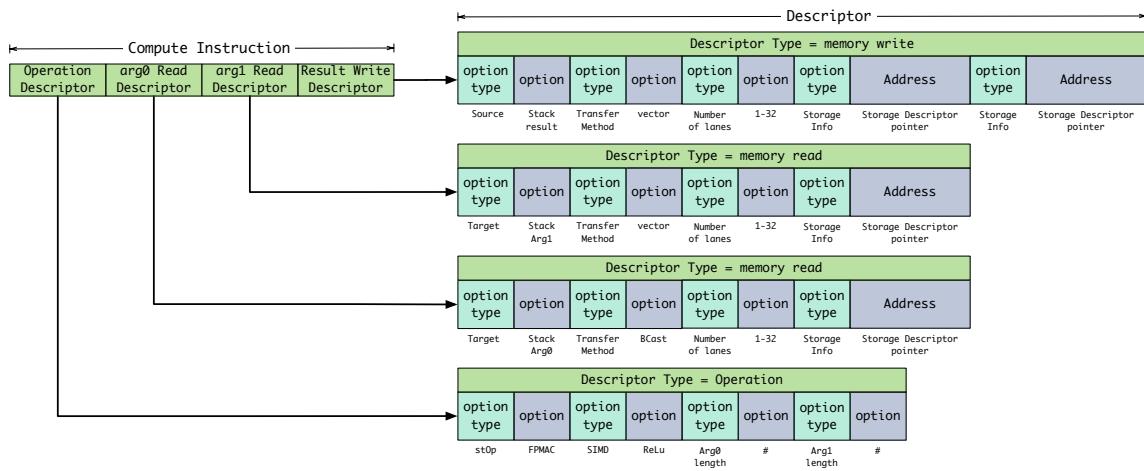
and `wum_wud_option_value`.

Note: The signal name convention used between blocks in the RTL is `<src>_<dest>_<signal_name>`.

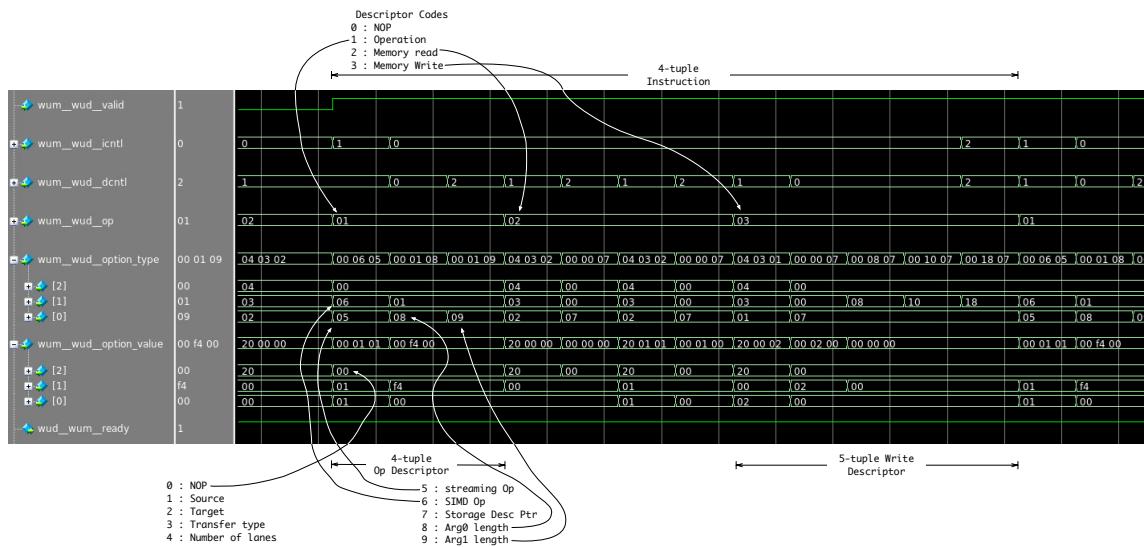
In Figure 4.3b, `wum` and `wud` refer to “work unit memory” and “work unit decode” which correspond to the manager instruction memory and instruction decoder respectively.

4.1.2.1 Accessing of Pre-synaptic ANe states and connection weights

A part of the research is determining how to store the ANN input and parameters in such a way as to effectively make use of main DRAM bandwidth. To provide parameters for the up to 32 execution lanes within the PE, the ANe parameters are stored in consecutive address locations. With one read to the DRAM, we access 128 words. This provides four weights for each of the 32 ANes being processed. These weights are sent to each lane of the PE over four cycles. We will discuss memory efficiency later,



(a) Compute instruction and descriptors



(b) Instruction memory waveform

Figure 4.3 Compute Instruction details

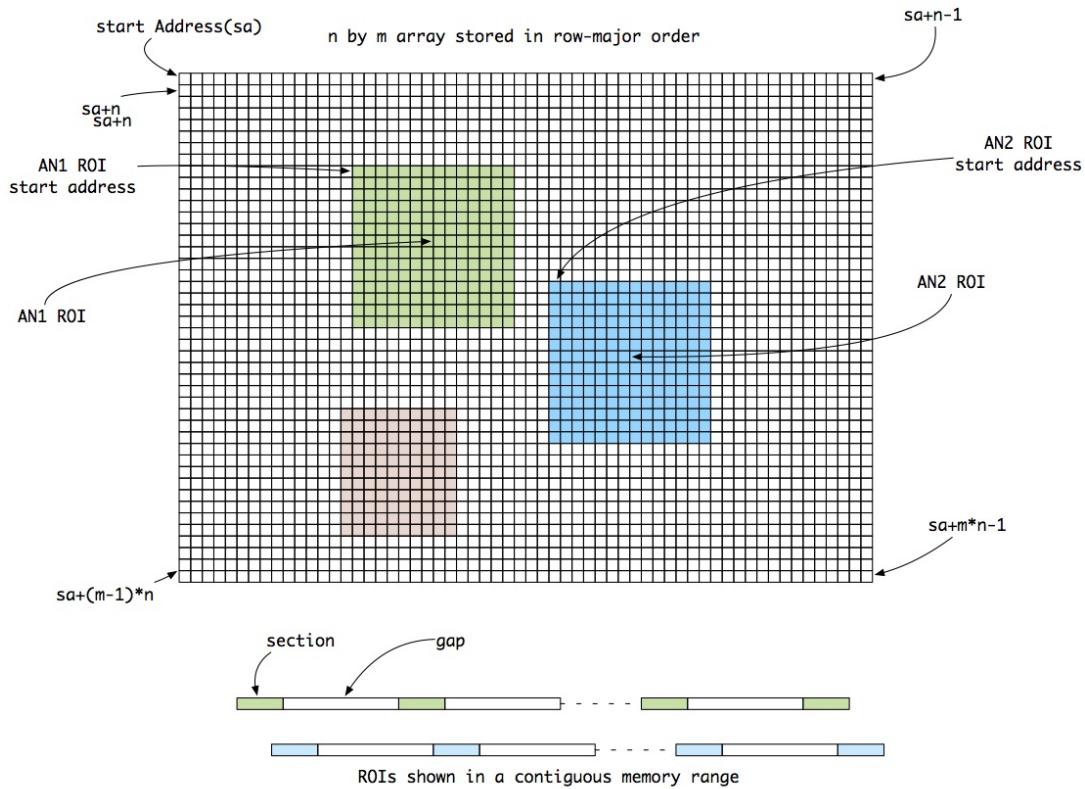


Figure 4.4 ROI Storage

but by taking advantage of the multiple DRAM banks along with pre-fetching and buffering, we are able to make very efficient use of the available bandwidth.

Providing the pre-synaptic ANe states to a particular ANe presents us with an interesting problem. Multiple layers of ANes whose pre-synaptic neurons are from the previous layer, represent DNNs. These previous layers represent the input to a given layer with the first layer's input being the actual input to the ANN. For the sake of generality, the ANN input array elements are considered as ANe states. Any given ANe operates on a ROI within the input array. In the case of fully-connected layers, the entire input is the ROI. In the case of locally-connected layers, the ROI is a portion of the input array. Figure 4.4 shows an input to a ANN layer in the form of a 2-D array along with the ROI of two locally-connected ANes.

The various connection weights are stored in multiple contiguous sections. However, it's not

possible to arrange the pre-synaptic ANe states in the locally-connected ROI in such a way that each ANe's ROI can be stored in contiguous memory locations. Assuming the input array is stored in row-major order, a locally-connected ROI is drawn from disjoint sections of memory. These disjoint sections contain a number of ANe states, in this case 14, and the sections are separated by a gap of a number of memory addresses. When the parameters are accessed while performing a particular operation, the memory controller within the manager must be informed of the start address and the lengths of the sections and gaps. In practice groups of ANes share a common ROI, so often when reading an ROI from the DRAM it is broadcast across a group of execution lanes. The read efficiency problem is solved by again taking advantage of the DRAMs banks and pages. To describe the reading of a locally-connected ROI, this work proposes a data structure called a "storage descriptor" to describe the ROI storage locations.

4.1.2.2 Storage Descriptor

Although disparate groups of ANes may have different start addresses for their ROIs, a commonality is observed in the ROI section lengths and gaps. So for each ANe group, the group's ROI starting address is stored along with a pointer to a common set of section lengths/gaps. This structure is termed a storage descriptor.

This storage descriptor contains, among other things, the start address of the ROI and a pointer to a section/gap descriptor. Many storage descriptors point to a common section/gap descriptor. This avoids having to have a unique section/gap descriptor for each ANe group.

Figure 4.5 shows the structure of the storage descriptor. The start of descriptor (SOD), middle of descriptor (MOD) and end of descriptor (EOD) are used to delineate each storage descriptor in memory.

4.1.2.3 Writing ANe state results to memory

When the PE has processed the group of ANes, the new ANe states are sent back to the manager and stored in DRAM in the row-major array format described earlier. When an operation is complete,



Figure 4.5 Storage Descriptor

in almost all cases one word per lane is written back to DRAM. Considering a DRAM page contains 128 words, the system typically writes a quarter of a page and this is a relatively inefficient use of DRAM bandwidth. However, the pre-synaptic fanin typically far exceeds 100 elements and in the baseline ANe shown in Table 1.1 the average fanin is 1650. So the read-to-write ratio is very high and the inefficient write has little impact on the overall performance.

As discussed in Section 3.4.7, in many cases the results have to be provided not only to the local SSC DRAM but also to other SSCs memory. This is handled by examining the write storage descriptors and if at least one storage descriptor address references another SSC's memory, all the write storage descriptors in the instruction are included in the NoC packet (see Figure 3.12).

4.1.3 Configuration Instruction

The configuration instruction is used for :

- Data transfer
 - Download Instructions from host to SSC manager instruction memory
 - Download Sync group data from host to SSC manager
 - Download ANN parameters and input from host to SSC memory
 - Upload ANN output to host from SSC memory
- System synchronization
 - Send a sync message to a group of SSCs
 - Wait for sync messages from a group of SSCs or host
 - Pause instruction fetch
 - Flush PE operations

The configuration instruction contains one descriptor and there are two configuration types that are characterized by the descriptor option tuple contents. All configuration descriptors start with a configuration option tuple. There are two configuration tuple types, sync and data. The sync and data option types are extended types that have a 24-bit option value. The 24-bit option value is used to define one of up to eight mode registers, each of which defines the type of configuration operation and various options. The configuration option tuple can be seen in Figure 4.6.

The data transfer configuration instruction is shown in Figure 4.7a. The descriptor is a 2-tuple with a configuration data option type and a storage option type. The sync configuration instruction is shown in Figure 4.7b and the descriptor is a 1-tuple with a sync configuration option type.



Figure 4.6 Configuration tuple

The data option value mode register [41] defines the type of data transfer along with information to aid the transfer. The storage option type contains a storage descriptor pointer which specifies the

address of memory transfers. An overview of the configuration instructions is given in Sections 4.1.3.1 and 4.1.3.2 with additional details in Section 5.1.2.2.

4.1.3.1 Configuration Data Instruction

There are currently four mode registers defined, an instruction download register, a sync group download register, a memory download register and a memory upload register. The contents of the mode register specify the type of transfer, the size of the transfer and some additional flags. When transferring to or from memory, an additional descriptor element contains a storage descriptor defining where the data should read or written.

4.1.3.2 Configuration Sync Instruction

The option element is a sync option whose value contains a mode register. There are currently four mode registers defined, a send, wait, pause and a flush register. The contents of the send and wait mode register specify the group of SSCs to be synchronized. The send register causes a sync NoC packet to be sent to all SSCs in the group. The wait register causes the manager to wait for a NoC sync packet to be received from all SSCs in the group. The pause mode register causes the instruction fetch logic to pause the specified number of clock cycles. The flush mode register causes the instruction fetch logic to wait for all outstanding PE operations to be returned before continuing.

4.1.4 Multiple Instruction Functions

When instructions or data are downloaded, in some cases there are tasks in the system that must be performed by chaining instructions together. This is the case when downloading the PE operation pointers and the SIMD instructions. In these cases, the host will have to first download the data to the SSC local DRAM in conjunction with a SSC configuration download instruction. The data is then transferred to the PE using an operation instruction with the StOp configured as a NOP so the data will pass through the StOp with the small local SRAM as the target. The PE controller will then transfer the contents of the SRAM to SIMD instruction memory or the operation pointer memory. As an example,

loading the SIMD instruction memory requires the procedure described in algorithm 1.

Algorithm 1 Load SIMD Instruction memory

```

1: // last compute instruction
2: COM:[ [ op: [ stOp:fpmac , simd:relu , numop0:<n> , numopl:<n> ] ]
3:      [ mr: [ tgt:std0 , txfr:bcast , lanes:<l> , StoD:<ptr> ] ]
4:      [ mr: [ tgt:std1 , txfr:vec , lanes:<l> , StoD:<ptr> ] ]
5:      [ mw: [ src:stu , txfr:vec , lanes:<l> , StoD:<ptr> , StoD:<ptr> ] ]
6:
7: //----- Start download -----
8: // make sure all compute instruction are complete
9: CFG:[ [ cfg: [ sync: [ flush ] ] ]
10: CFG:[ [ cfg: [ sync: [ send: [host ] ] ] ]
11: CFG:[ [ cfg: [ sync: [ wait: [host ] ] ] ]
12: // Host sends release
13:
14: // Host starts simd instruction download to SSC memory
15: // Next SSC instruction prepares wr_cntl for data from Host
16: CFG:[ [ cfg: [ data: [ mem_dn:<m> , StoD:<ptr> ] ]
17: CFG:[ [ cfg: [ sync: [ pause: [ind ] ] ] ]
18: // fetch paused waiting for release, wr_cntl ready for Host data
19: // wr_cntl releases fetch when data transfer complete
20: COM:[ [ op: [ stOp:ld_simd , simd:nop , numop0:<m> ] ]
21:      [ mr: [ tgt:std0 , txfr:bcast , lanes:1 , StoD:<ptr> ] ]
22: // manager sending instruction data to PE using compute operation with NOPs
23: // flush PE to ensure instruction data complete
24: CFG:[ [ cfg: [ sync: [ flush ] ] ]
25: //----- end of download -----
26:
27: // continue with compute instructions
28: COM:[ [ op: [ stOp:fpmac , simd:relu , numop0:<n> , numopl:<n> ] ]
29:      [ mr: [ tgt:std0 , txfr:bcast , lanes:<l> , StoD:<ptr> ] ]
30:      [ mr: [ tgt:std1 , txfr:vec , lanes:<l> , StoD:<ptr> ] ]
31:      [ mw: [ src:stu , txfr:vec , lanes:<l> , StoD:<ptr> , StoD:<ptr> ] ]

```

4.2 Host Instructions

The host is responsible for transferring ANN parameters and input data to the SSC and for transferring ANN output data back to the host. It is also responsible for downloading configuration data including SSC instructions, PE and SIMD instructions and configuration tables such as storage pointer tables and PE, StOp and SIMD operation tables.

The host will use the NoC to carry the commands and data to accomplish the transfers. The commands will employ the same option tuple method as described in Section 4.1.3.

A transfer of data to the SSC can be solicited or unsolicited. In the unsolicited case, the host initiates the transfer by sending the first NoC packet with two option tuples along with the first data as shown in Figure 4.8. This first packet contains a configuration data option tuple (see Figure 4.6) that contains a mode register used to define the type of transfer. The second tuple has a storage descriptor pointer defining where in memory to write the data. Once the first packet has been received, the SSC expects only to receive data packets as shown in Figure 4.9. In the solicited case, the transfer is initiated from a configuration data transfer instruction in which case the main controller receives the mode register and storage descriptor from the instruction decoder. An example instruction can be seen in Figure 4.7a. In this case, the host will only send data packets as shown in Figure 4.9. Additional information on the configuration data option tuple mode registers can be found in Sections 5.1.2.2.7 and 5.1.2.2.8.

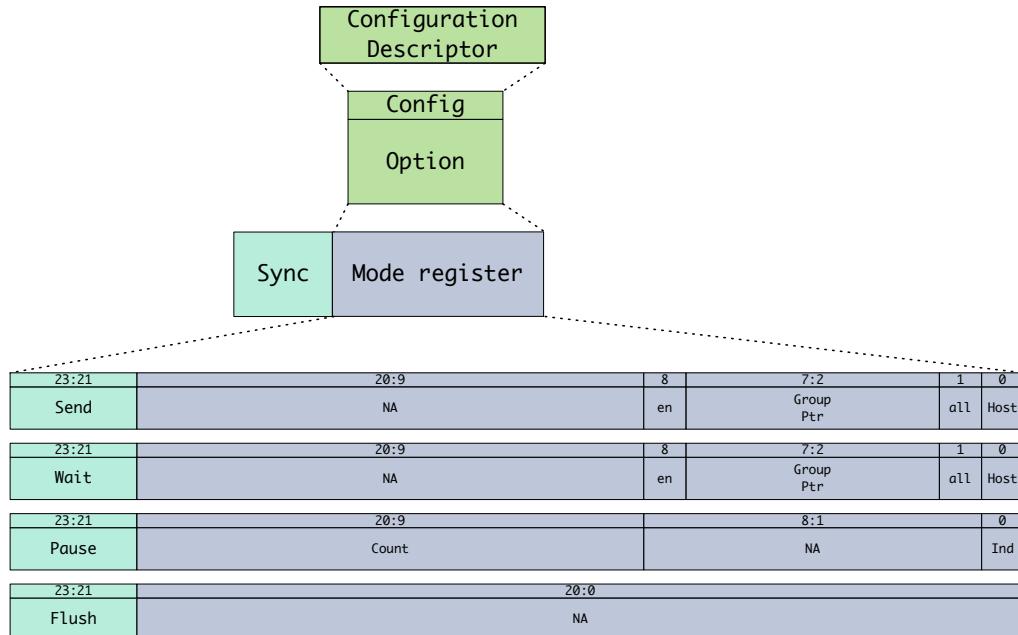
In the case of a data download, the main controller sends the storage tuple to the memory write controller and indicates it's a DMA. This will inform the memory write controller to expect further data packets and associate those with the current storage descriptor. In the case of a data upload, the main controller will check the status of the instruction decode and if it has not been halted by a sync instruction, it will halt the instruction decoder. The main controller then sends the storage tuple to the lane 0 memory read controller.

The data to be transferred is included in separate packets as shown in Figure 4.9. For downloads, the host sends the data and the main controller passes the packets to the memory write controller. In

the case of reads, the main controller receives data from the memory read controller, packetizes the data and sends it to the host. If the amount of data exceeds the 32 data word maximum transmission unit (MTU) of the NoC, the data will be fragmented.



(a) Data transfer instruction



(b) Sync instruction

Figure 4.7 Configuration instruction types



Figure 4.8 Host unsolicited download first NoC packet

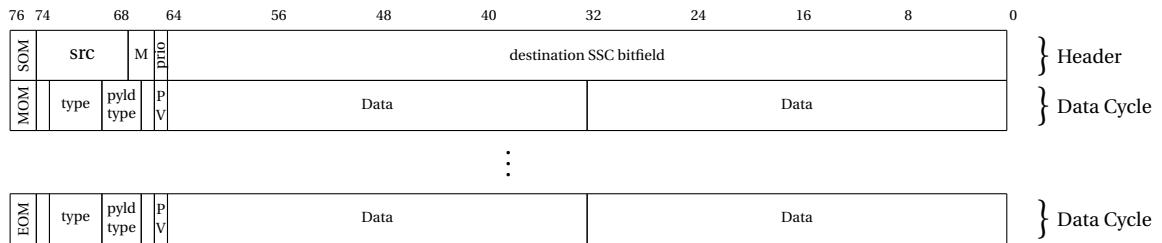


Figure 4.9 Host transfer NoC data only packet

CHAPTER

5

DETAILED SYSTEM DESCRIPTION

A detailed flow diagram and block diagram of the sub-system column can be seen in Figures 5.1 and 5.15 respectively.

5.1 Manager

A block diagram of the manager can be seen in Figure 5.2.

5.1.1 System controller

The system controller is responsible for initialization and general system configuration. A block diagram can be seen in Figure 5.3.

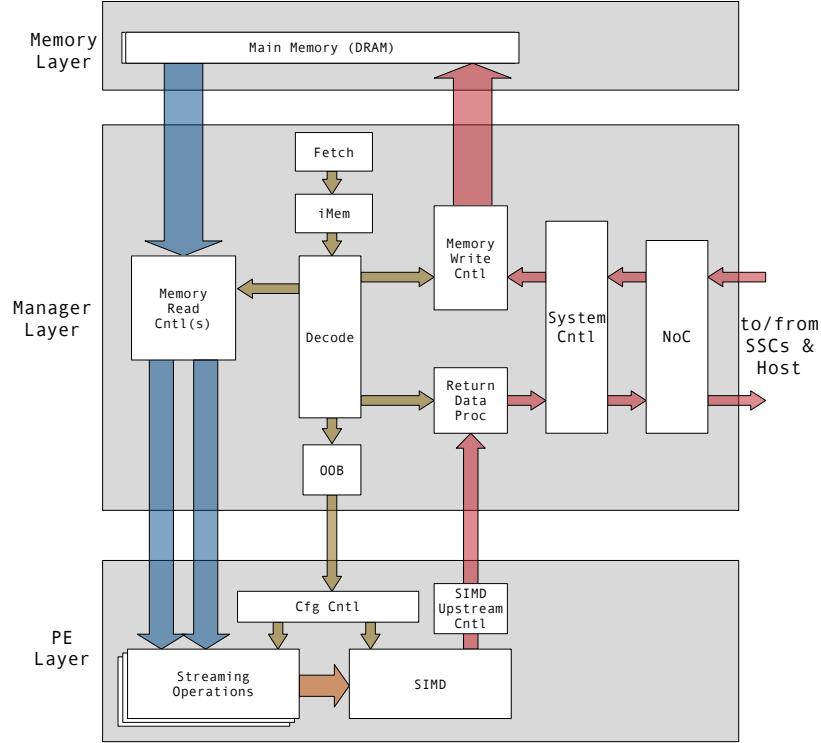


Figure 5.1 Sub-System Column (SSC) Flow Diagram

5.1.1.1 Initial Boot

The system controller is responsible for performing the initial Bootstrap Protocol (BOOTP) process. This involves quiescing the system after reset and downloading the initial boot instructions from the host. After reset, the controller keeps the instruction fetch logic disabled and expects unsolicited NoC packets from the host that contain the initial boot code. The host will unsolicitedly send the BOOTP code over the NoC to each SSC. The controller in each SSC decodes the NoC packets and writes the data directly into instruction memory. Currently each NoC MTU packet contains 16 instruction memory entries (see Figure 5.4). The number of initial instruction entries is hard-coded but it is anticipated a standard on-chip protocol such as Joint Test Action Group (JTAG) will be used to set initial BOOTP configuration. Once the host has sent the BOOTP code, each SSC controller releases its instruction fetch block.

It is anticipated that the initial BOOTP code will not contain operational instructions but only

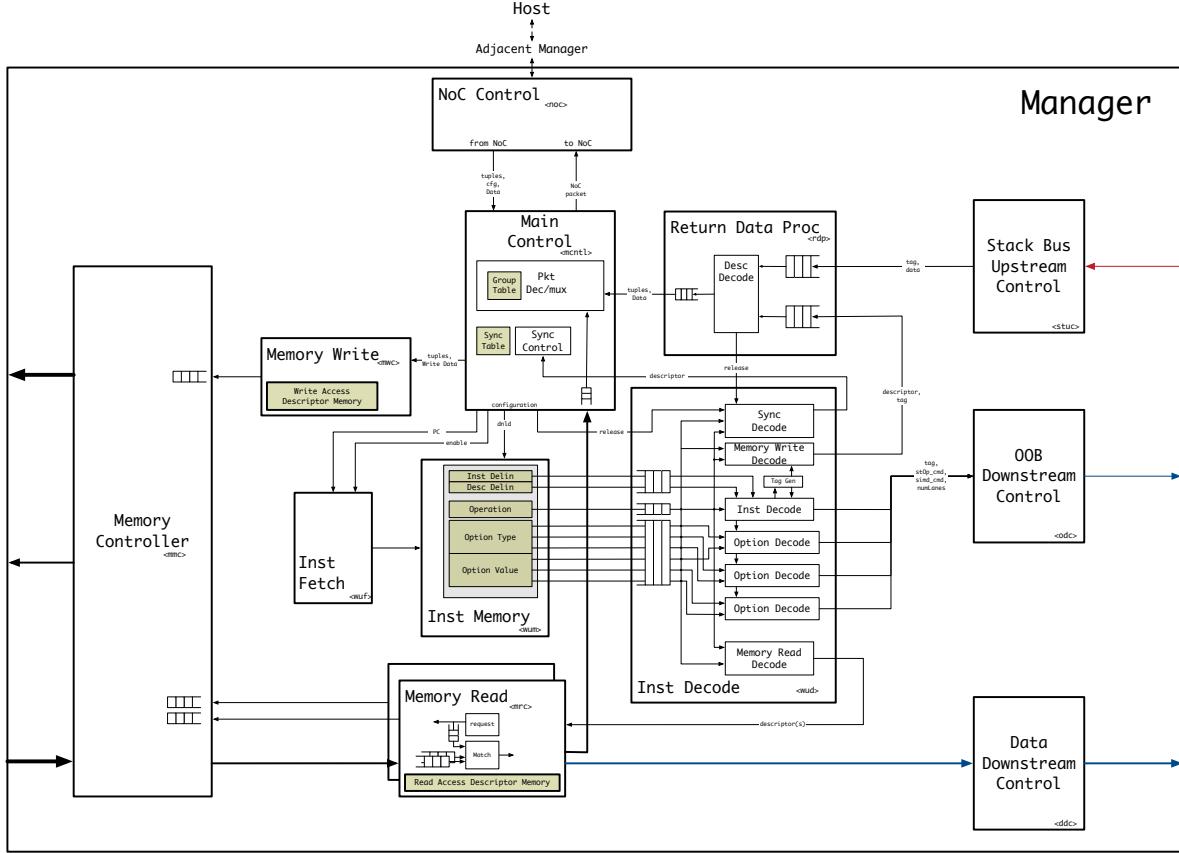


Figure 5.2 Manager block diagram

configuration instructions to download configuration tables, the PE SIMD instruction memory and the ANN parameters. For example, to download the ANN parameters, the BOOTP instructions will include a configuration sync and a configuration data download. The first BOOTP instruction will be a configuration sync sent to the host to indicate the initial BOOTP is complete and ANN parameters are ready to be received. After the configuration sync is sent to the host, the configuration data instruction will prepare the SSC to receive data from the host and the host will then start sending configuration data. The size of the data is known by both the SSC via the instruction and the host. The ANN parameters can be downloaded using multiple downloads. Once ANN parameters and system tables are downloaded, the last instruction can be another configuration data download instruction to load operational code.

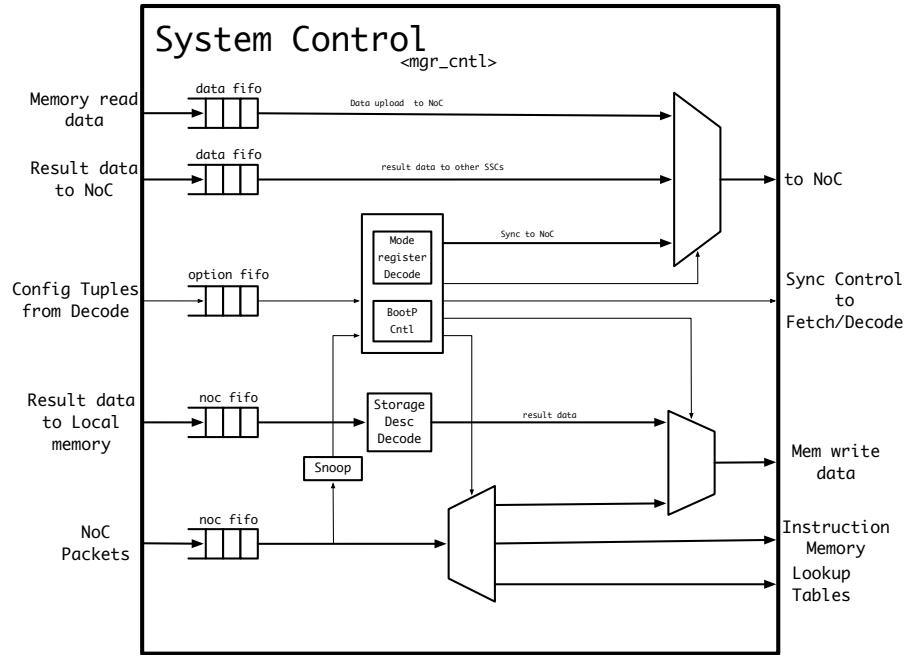


Figure 5.3 System controller

5.1.2 Instruction Decoder

In Figure 5.2, instructions are read from instruction memory by the instruction fetch block and the output of the memory is passed to the instruction decoder block.

5.1.2.1 Compute Instructions

The operation descriptor is decoded and a StOp pointer and a SIMD Program Counter (PC) pointer are extracted. A sequential tag is generated and along with the StOp and SIMD pointers is immediately sent to the PE inside an OOB control packet. The StOp pointer specifies what streaming operation is to take place on the data directly streamed to the PE. The SIMD pointer points to the start of the function that will be used to process the result from the StOp. The PE will immediately start preparing for downstream operand data. If the SIMD operation includes result data being returned to the manager, the tag will be included in the upstream result data packet.

The memory read descriptors are decoded to identify the target memory read controller and

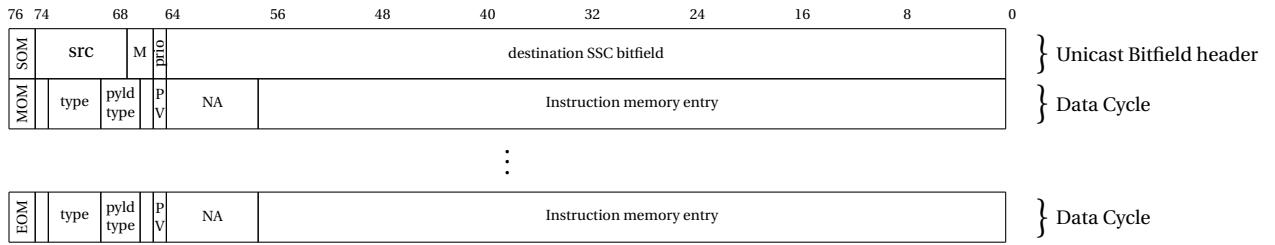


Figure 5.4 Host boot code download NoC packet

the storage descriptor pointer directed to the appropriate memory read block. There is a memory read block associated with each of the operand streams which is responsible for generating memory requests and directing the DRAM data to all the execution lane streams. A request block inside the memory read block immediately starts pre-fetching the memory data by sending memory requests to the DRAM. A stream block inside the memory read block immediately starts waiting for data from the DRAM and will direct DRAM data to the appropriate execution lane.

The memory write descriptor is decoded and the storage descriptor pointer extracted and along with the tag is sent to the return data processor. Currently the system only allows in-order data and any tag mismatch will cause an error. In the case where multiple upstream packets are associated with a tag, the RDP ensures the correct number of upstream data packets is observed.

At this point all the blocks that take part in a compute operation on a group of ANEs are performing the various tasks. As mentioned in Section 4.1.2, the inputs to many blocks employ FIFOs. This allows blocks to pipeline tasks to absorb any latencies, and allows blocks to start sending data to a destination block before that block has been configured to receive data. The FIFO will assert a flow control signal until the block is ready to receive. In practice, the instruction decode logic batch-decodes up to eight instructions and sends descriptor contents to dependent blocks where they are also pipelined in FIFOs. The head of the FIFO is processed immediately once the previous task has been completed, thus avoiding any decode latencies.

5.1.2.2 Configuration Instructions

As shown in Figure 4.7, the configuration instructions are responsible for :

- Performing synchronization between the SSC and other SSCs or the host system.
- Performing data download and upload operations

The configuration packets are broken into two types, configuration data and configuration sync.

The configuration instruction contains a single descriptor with one, two or three option tuples.

The configuration sync instruction descriptor contains a single configuration sync option tuple.

The 24-bit option value contains a 3-bit mode register identifier and a 21-bit mode register. There are currently four mode registers defined, “Send”, “Wait”, “Pause” and “Flush.” Each register has fields specific to the mode as seen in Figure 4.7b.

The configuration data instruction descriptor contains a configuration data option tuple and a storage descriptor tuple and optionally a target tuple. The configuration data option value is 24 bits with a 3-bit mode register identifier and a 21-bit mode register. There are currently four mode registers defined, “instruction download”, “sync group download”, “memory download” and “memory upload.” Each register has fields specific to the mode as seen in Figure 4.7a.

5.1.2.2.1 Sync Send

The Sync Send mode register fields can be seen in Register 5.1. The decoder sends the sync option tuple to the system controller. A sync NoC packet is constructed based on the register contents sent over the NoC. If the group pointer enable bit is set, the system controller uses the pointer to index into a 64-element table. Each table entry is a 64-bit bitfield indicating which SSCs are part of the group. If the “*all*” flag is set, the sync packet will be sent to all SSCs. if the “*host*” flag is also set, the host bit in the NoC packet will be set.

Register 5.1 SYNC SEND MODE REGISTER

Mode Reg ID		Not used			Group Pointer enable			Sync Group pointer			Send to all SSCs			Send to host		
23	21	20			9	8	7			2	1	0				
0	0	0		NA		en		gPtr		all	host					

5.1.2.2.2 Sync Wait

The Sync Wait mode register fields can be seen in Register 5.2. The decoder sends the sync option tuple to the system controller. The instruction decoder will stop processing any more instructions until the release signal is asserted from the system controller. The wait option informs the system controller to start expecting sync send packets from other SSCs and/or the host. Based on the register values, the system controller will assert the release signal to the instruction decoder only after sync send packets have been received from all specified sources. If the group pointer enable bit is set, the system controller uses the pointer to index into a 64-element table. Each table entry is a 64-bit bitfield indicating from which SSCs a sync send packet should be received. If the “*all*” flag is set, all sync send packets are expected from all SSCs. if the “*host*” flag is also set, a sync send packet is expected from the host.

Register 5.2 SYNC WAIT MODE REGISTER

Mode Reg ID		Not used			Group Pointer enable			Sync Group pointer			Receive from all SSCs			Receive from host		
23	21	20			9	8	7			2	1	0				
0	0	1		NA		en		gPtr		all	host					

5.1.2.2.3 Sync Pause

The Sync Pause mode register fields can be seen in Register 5.3. The decoder sends the sync option tuple to the system controller and ceases decoding instructions. If the “*indefinitely*” flag is set, instruc-

tion decode will not restart until the release signal is asserted from the system controller, otherwise the decoder will wait a number of clock cycles specified by the count field and then restart decoding instructions.

Register 5.3 SYNC PAUSE MODE REGISTER

Mode Reg ID		Wait <count> cycles		Not used		Indefinitely	
23	21	20	9	8	1	8	
0	1	0	count	NA		Ind	

5.1.2.2.4 Sync Flush

There are currently no fields implemented in the sync flush register. This is designed to pause instruction decode until all outstanding commands sent to the PE have been returned to the manager. The decoder sends the sync option tuple to the return data processor and pauses processing instructions. When the return data processor receives all outstanding tags, it asserts a release signal to the decoder.

5.1.2.2.5 Instruction download

The Instruction Download mode register fields can be seen in Register 5.4. The decoder sends the data option tuple to the system controller. The system controller will ensure the fetch and decode blocks are disabled and then expects to receive the number of instruction entries from the host over the NoC. Once the instruction memory is loaded, the system controller will enable the fetch and decode logic.

Register 5.4 INSTRUCTION DOWNLOAD MODE REGISTER

Mode Reg ID		Number of entries		Not used		continue	
23	21	20	8	7	1	0	
0	0	0	number	NA		en	

5.1.2.2.6 Sync group table download

The Sync group register fields can be seen in Register 5.5. The decoder sends the data option tuple to the system controller. The system controller will ensure the fetch and decode blocks are disabled and then expects to receive the number of sync groups from the host over the NoC. The group table entry is 64-bits and eight are contained in an NoC packet. Once the sync group table is loaded, the system controller will enable the fetch and decode logic.

Register 5.5 SYNC GROUP TABLE DOWNLOAD MODE REGISTER

Mode Reg ID			Not used		Number of groups		Not used		continue	
23	21	20	16	15	8	7	1	0		
0	0	1	NA	number			NA	en		

5.1.2.2.7 Memory download

The Memory download mode register fields can be seen in Register 5.6. The decoder sends the data option tuple to the system controller. The system controller will ensure the fetch and decode blocks are disabled and then expects to receive the number of bytes from the host over the NoC. As each data packet is received, it is passed to the memory write controller along with the storage descriptor for writing to DRAM.

Register 5.6 MEMORY DOWNLOAD MODE REGISTER

Mode Reg ID			Number of bytes	
23	21	20		0
0	1	0	number	

5.1.2.2.8 Memory upload

The Memory upload mode register fields can be seen in Register 5.7. The decoder sends the data option tuple to the system controller. The system controller will ensure the fetch and decode blocks are disabled. The controller sends the storage descriptor and target option tuples to the stream 0 memory read controller. The read controller generates read requests to the DRAM and sends the corresponding read data to the system controller. The system controller takes the streaming data from the read controller and partitions it into NoC MTU sized blocks, then embeds the block of data in an NoC packet and sends the fragmented data to the host using multiple data NoC packets.

Register 5.7 MEMORY UPLOAD MODE REGISTER

Mode Reg ID			Number of bytes
23	21	20	0
0	1	1	number

5.1.3 Main memory controller (MMC)

The MMC is responsible for taking read requests from the two MRCs and write requests from the MWC. The read requests the MMC receives from the MRCs include channel, bank, page and cache line addresses. Read requests also contain an operation identifier. Write requests from the MWC include channel, bank, page and cache line addresses. The MMC processes read requests from the two MRCs and write requests from the MWC. The write requests are given priority with the and the read requests processed in a round-robin fashion. the three requestors providing a small priority to write requests. In addition, the MMC processes all read requests associated with an operation before processing requests from the next operation. This is because memory requests are pre-fetched and memory requests for the next operation will be received before the MRC has streamed data for the previous operation. This avoids the case where a request from the next operation could block a

request from the previous operation causing a deadlock.

The MMC does not reorder requests to improve DRAM efficiency. The MMC does keep track of open pages within banks to avoid unnecessary page open and page close commands. It should be noted that refresh has yet to be implemented but this is not anticipated to be a significant impact and will be done in future work. The MMC ensures the DRAM protocol is observed and this has been verified using Tezzaron DiRAM4 verilog models.

5.1.4 Memory read controller (MRC)

The MRC is provided with multiple option tuples by the decoder block, a storage descriptor pointer, the number of execution lanes, the target for read data and the transfer type. The MRC contains the DRAM read FIFOs described in Section 3.4.4 and shown in Figure 3.7. A block diagram of the MRC can be seen in Figure 5.5.

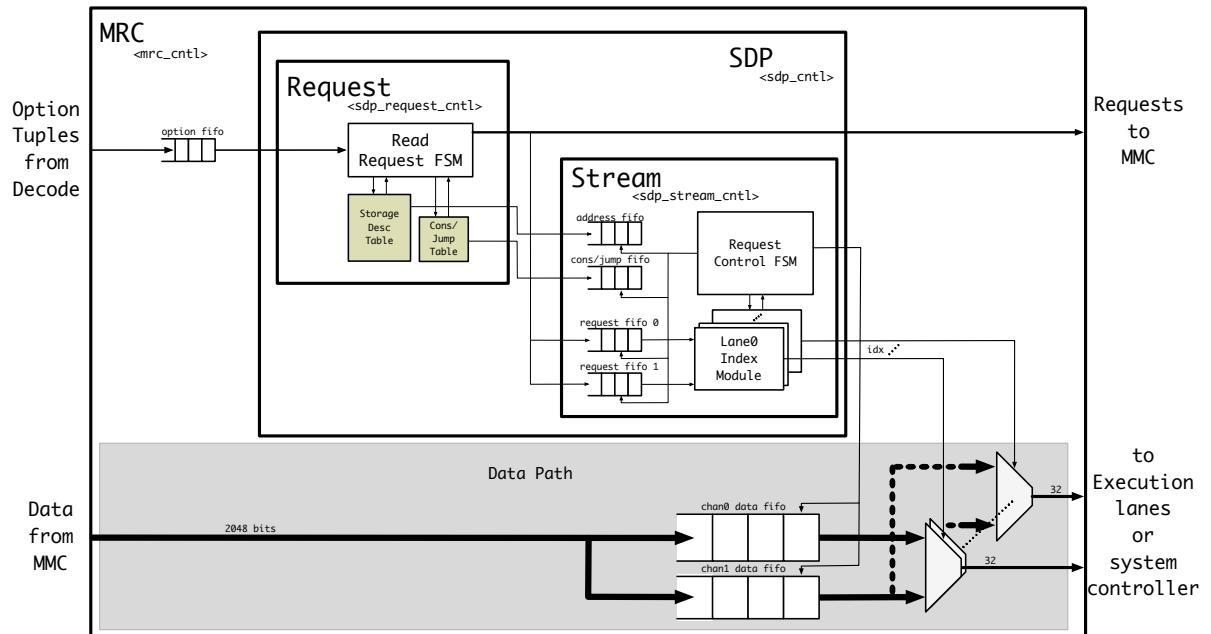


Figure 5.5 MRC block diagram

The MRC is one of the most complicated blocks in the system. It is also the largest, mainly because of the size of the read data path FIFOs described in Section 3.4.4. There are two MRC blocks instantiated in the manager, one for stream 0 and one for stream 1 in the execution lanes.

The MRC uses the storage descriptor to identify the start address of the data and how the address should be incremented. The storage data processor (SDP) block contains a request block and a stream block. The request block uses the storage descriptor pointer to index into a small memory containing the actual storage descriptor. As described in Section 4.1.2.2, the storage descriptor itself contains a start address and a pointer to a table containing consecutive and jump fields. The starting address and consecutive/jump fields allow the request generator to make disjoint memory requests based on the ROI of the data. If the ROI is contiguous in memory, then there is one consecutive field and no jump field.

The request block generates memory requests based on the storage descriptor contents and number of lanes and sends the requests to the MMC. The request information is also sent to the stream block. As the request block processes the storage descriptor start address and consecutive/jump fields it also sends the information to the stream block.

The stream block is responsible for taking the 2048-bit memory data from the MMC and directing words to each execution lane. As the data is returned from the MMC, it is placed in a channel 0 FIFO and a channel 1 FIFO. The stream block has a per-execution-lane index module, each of which generates an index to the channel and word. The index is used to multiplex the data to the execution lane stream. The per-execution-lane index module is required to account for bank and page boundaries in the ROI as described in Sections 4.1.2.1 and 4.1.2.2. The lane index module uses the storage descriptor information to generate a memory location address that includes channel, bank, page and word addresses. The location address is then matched to the request at the head of the two request FIFOs, and if a match occurs, the data is passed to the execution lane bus. If there is no match, the index module requests a read of the data FIFOs. The request control finite-state machine (FSM) will only perform the read if all lane index modules are asserting the read request. This is done to account for data crossing a bank, page or cache line boundary allowing the execution lanes to get out of sync.

5.1.5 Return data processor (RDP)

The RDP is responsible for taking the result data from the PE and determining into which SSC it should be stored (see Figure 5.6).

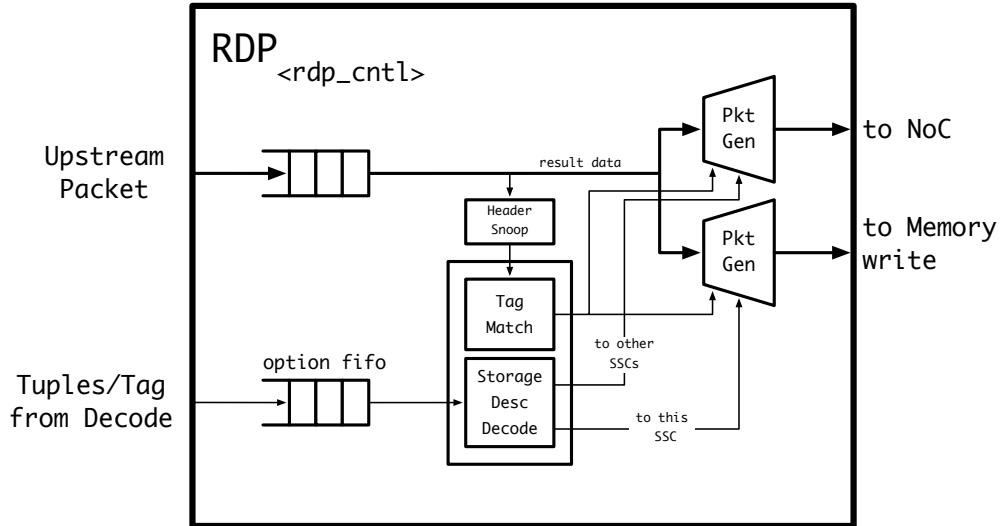


Figure 5.6 Result data processor

The RDP receives a descriptor from the decoder that includes storage descriptor information and the tag associated with an operation sent to the PE. The information is stored in a FIFO and as data is returned to the manager over the upstream stack bus, the RDP matches the return data tag with the head of the FIFO. Currently the return data is in order but to support an expansive architecture the tag is provided to and checked by the RDP. When the data is matched to the tag, the RDP examines all the storage descriptor pointers. The pointers include the SSC index in the Most Significant Bits (MSBs) and the RDP constructs an SSC bit-field. Once the descriptors have been parsed, if one of the destinations is the local SSC DRAM, the RDP passes the descriptor and data to the system controller that in turn passes it to the MWC. If the destination(s) include other SSCs, the RDP provides the SSC bitfield along with the data to the NoC.

5.1.6 Memory write controller (MWC)

The memory write controller (MWC) receives data from two sources, the NoC via the system controller and the RDP. The MWC uses the storage descriptor provided with the data to identify the write address. The MWC does not store the data immediately, it places the data in a small crude cache that has enough storage for two pages per channel. A block diagram of the MWC can be seen in Figure 5.7. As

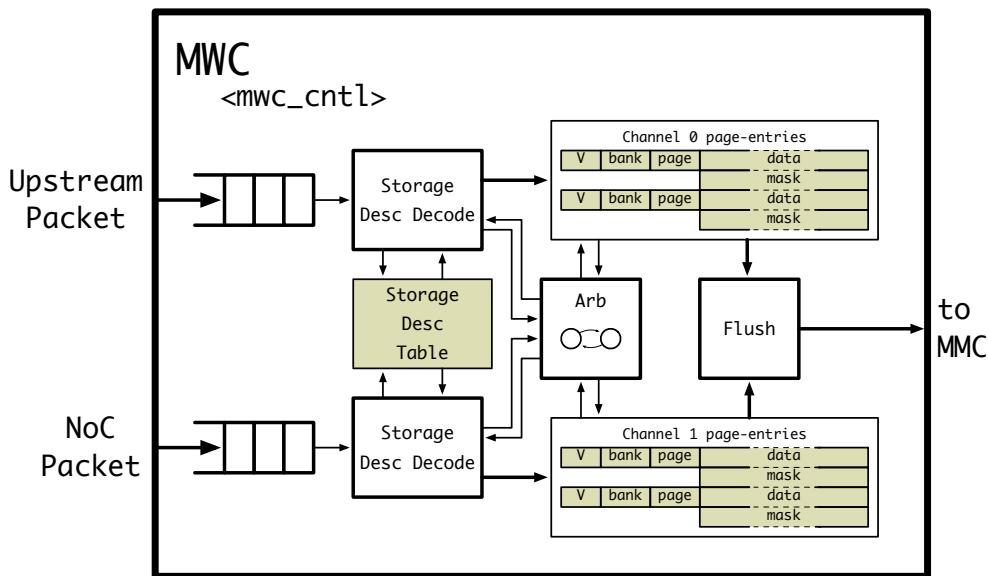


Figure 5.7 MWC block diagram

data is received from the RDP or NoC, the addresses are compared to the cache entry and if a page address match occurs and the corresponding word is invalid, the data is stored in the cache. If a page miss occurs, the contents of one of the entries is written to memory and the new data is stored in the cache.

This crude cache was provided for two reasons: First, even though write efficiency was not anticipated to be an issue as described in Section 4.1.2.3, it was anticipated that the addresses from multiple operations may occur in consecutive address and by coalescing data would make writes more efficient. The more important reason was to provide expansibility. By accounting for a small

cache, future implementations can be created with less logic area than they would otherwise need.

5.2 Processing Engine

A block diagram of the PE can be seen in Figure 5.8.

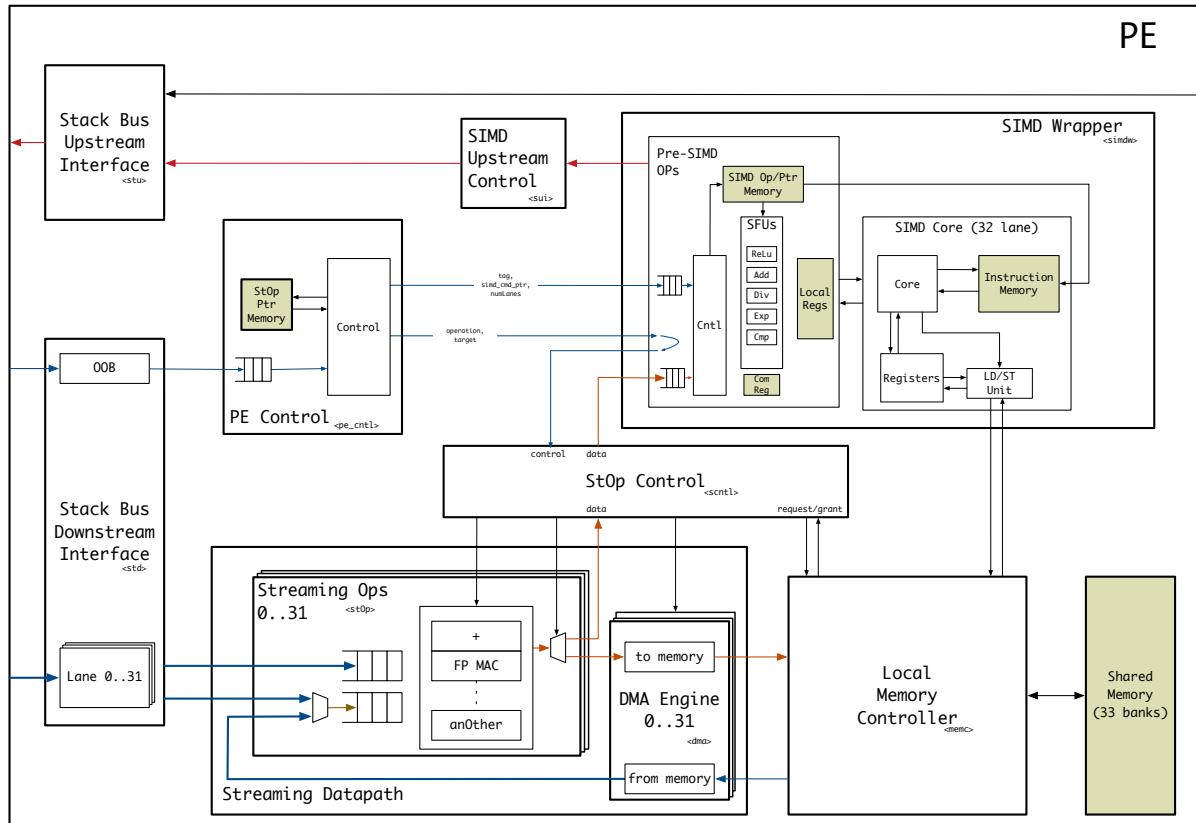


Figure 5.8 PE block diagram

5.2.1 Configuration

The manager sends configuration information to the PE over the downstream OOB bus. The OOB packet contains option tuples used by the PE controller to configure functions within the PE. The controller extracts the StOp and SIMD operation pointers from the appropriate option tuple value.

The StOp pointer is used to point to a local StOp configuration that contains the various configuration data required by the StOp function. The configuration data includes:

- StOp operation type
- Number of active execution lanes
- Source of the argument data, which can be downstream data from the manager or from the small local SRAM
- Destination of the result data, which can be the SIMD and/or the small local SRAM

Once the information is provided to the StOp block and the pointer provided to the SIMD, the operation is immediately started. Currently only StOp and SIMD pointer option tuples are used. An example of the downstream OOB transactions can be seen in Figure 5.9. This example shows both normal and extended option tuples.

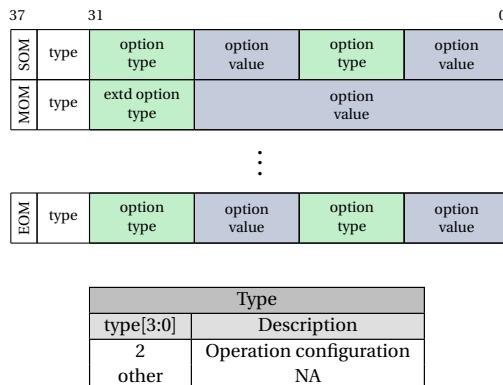


Figure 5.9 Downstream OOB data transactions



Figure 5.10 Downstream OOB simulation waveform

5.2.2 PE Controller

The PE controller takes OOB control packets stored in an interface FIFO and extracts the StOp and SIMD pointers. The SIMD pointer is passed to the SIMD wrapper along with the tag associated with the operation. The SIMD wrapper contains an interface FIFO and the PE controller will pause if the SIMD wrapper interface asserts flow control. The StOp pointer is used to index into a small SRAM that contains the long instruction word used to select the StOp function and control the source and destination of the data. The contents of the memory is shown in Table 5.1. When the StOp has completed the operation, the PE controller takes the next operation from the FIFO. A block diagram of the PE controller can be seen in Figure 5.11.

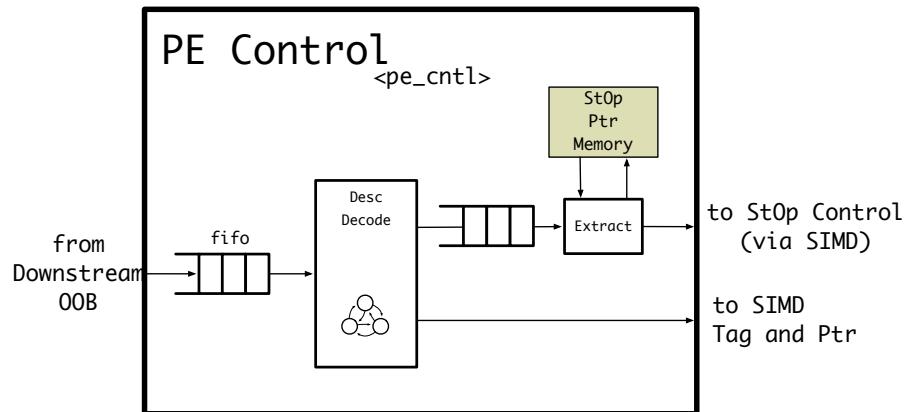


Figure 5.11 PE controller block diagram

Streaming Operation	Source Address stream 0	Type 0	Source Address stream 1	Type 1	Destination Address
---------------------	-------------------------	--------	-------------------------	--------	---------------------

Field Definitions			
Field	Width	Description	Comment
StreamingOp	21	Controls streaming operation	Further decoded in stOp block
Source Address 0	24		
Source Address 1	24	Local memory address	if using local memory
Destination Address	24		
Type	4	Data type	WORD only

Table 5.1 stOp pointer memory fields

5.2.3 Streaming Operations

There is a StOp module for each execution lane and the StOps are designed to operate on data passed from the manager at or near line-rate. If line-rate cannot be maintained, an interface FIFO flow-control mechanism is employed to slow the data from the manager. The StOp is designed to perform either a MAC operation, which is employed during ANe state calculation, or a multiply operation which is used during a Softmax [47] calculation (see Appendix A). The StOp receives the operation from the PE controller and decodes the fields as shown in Register 5.8 and Table 5.2. In the case of a MAC operation, each lane produces a single result that is passed to the SIMD. In the case of a multiply operation, each lane generates a result on each clock cycle and each is passed to the SIMD. Note that based on the destination specified, the result can also be placed in local SRAM.

It should also be stated that while the StOp is processing the current data, the SIMD may be operating on the result of the previous operation. It is expected the SIMD will have completed the previous operation before the StOp completes the current operation, but again, if necessary, a flow control mechanism between SIMD and StOp will be engaged if the SIMD is not ready.

There is provision for one of the inputs into the StOp module to be from the local memory via a local DMA controller. Although this is designed and tested, this processing path is for future work. A block diagram of the StOp can be seen in Figure 5.12.

5.2.4 SIMD

The SIMD takes the operation from the PE controller and indexes into a small memory. This SIMD pointer memory contains a PC for the SIMD along with control information for processing the StOp result inside the wrapper. A four-element vector allows up to four cascaded operations using the SFUs, to operate on the data from the StOp. The wrapper contains four SFUs, a ReLu, a binary32 adder, a binary32 exponential, a binary32 comparator and a binary32 divider. The ReLu operation is a simple operation and is performed in parallel on all 32 results from the StOp. The divide, exponent, comparator and adder are more complex and there are single instantiations of each of the SFUs. When processing a group of values from the StOp, each value is processed serially.

The most common usage of the SFUs is during ANe state calculation. In this case, the ReLu is used as one of the four operations with the other three being a “send” followed by three NOPs. The ReLu is preceded by the MAC operation in the StOp. Once the SFU operations have concluded, the result is sent directly back to the manager over the upstream bus.

For this work, a high level of importance was not assigned to the actual SIMD because a) the functionality provided by the SIMD is to provide a level of flexibility for future work and b) the choice of SIMD is currently under consideration. Therefore an actual SIMD was not simulated and the data from the StOp was processed only by the special functions provided in the wrapper as described above and shown in Figure 5.13. However, it was important to include the SIMD in the area portion of the study. To account for the SIMD, in the area layout study a placement blockage is used based on a 32 lane, 32-bit SIMD from [39]. In practice, the SIMD will take the result data from the StOp and/or the SFUs and perform a pre-programmed operation starting at the PC indicated by the SIMD operation pointer provided by the PE controller. It is expected that in future work the SIMD would also be given control of the SFUs allowing for more flexible and complex processing.

5.2.5 SFUs

The adder is used during single ANe processing as described in Section 3.4.6 and shown in Figure 3.10b. The adder is also employed in the softmax calculation used in the final stage of ANN classifiers. In this case, when processing the ANe, the operations performed are the ReLu, the exponent, and an accumulation using a local register. The ANe states are sent back to the manager. All the ANes are processed and the local register contains the summation of the ANe states. The cells are then parsed a second time using the divider and the stored local value to calculate the softmax. Other than the ReLu, the SFUs are not considered hot-spots in the processing of an ANN and because of this multicycle paths are employed during synthesis allowing these complex functions to meet timing.

5.2.6 DMA Controller and Local memory

The PE provides some local SRAM for future use. Both the StOp and the SIMD will have access to the local memory. In the case of the StOp, one of the StOp inputs can be sourced from the local memory using a DMA controller. The DMA controller is controlled from fields in the instruction word in the PE controller. The paths from the local memory to the StOp and from the StOp to the local memory have been tested.

5.2.7 Upstream controller

The upstream controller takes the data from the SIMD and a tag from the PE controller and sends it to the manager. The format of the upstream transactions can be seen in Figure 5.14. The upstream packet format accommodates both data and tag to be transmitted or just the tag. The reason for allowing tag-only operation is to accommodate a sync flush operation without data being returned to the manager. Currently only the tag and data mode has been implemented.

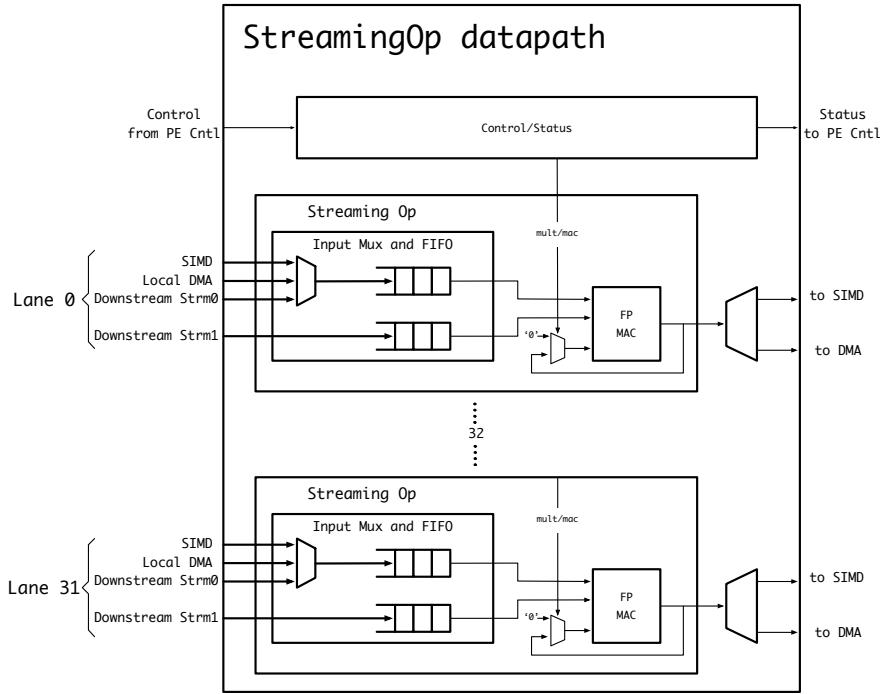


Figure 5.12 Streaming operations block diagram

Register 5.8 STREAMING OPERATION CONTENTS

Function								Destination				Stream 1 source				Stream 0 source			
16	12	8	6	5	3	2	0												
opCode				dest				src1				src0							

Source		Destination		Function	
src*[2:0]	Description	dest[2:0]	Description	opCode[4:0]	Description
1	Local memory	1	Local memory	2	MAC
2	Downstream bus	4	SIMD	8	Multiply
4	SIMD	other	NA	31	NOP
other	NA			other	NA

Table 5.2 Streaming operation fields

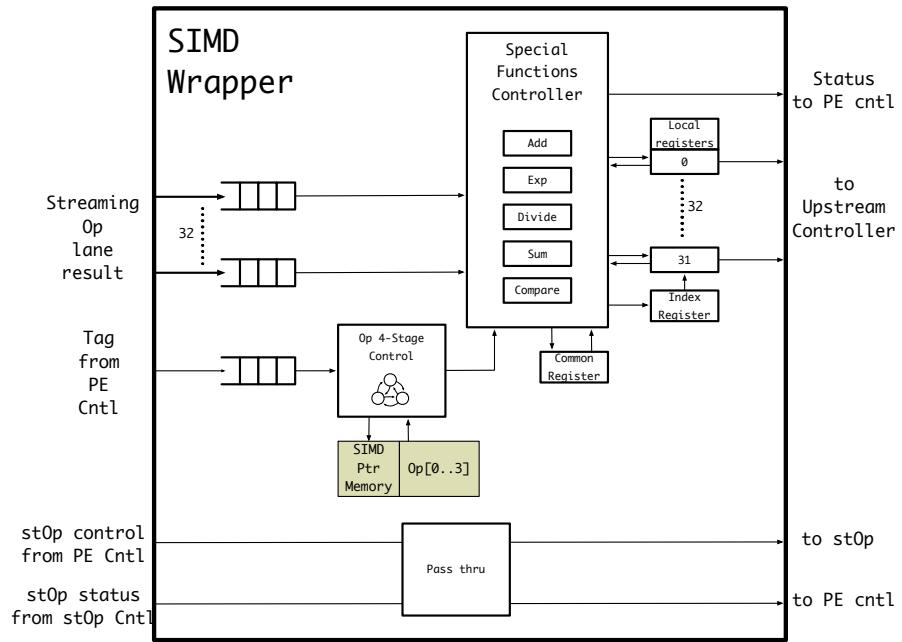


Figure 5.13 SIMD wrapper block diagram

Special Function Table Codes		
code[3:0]	Description	Comment
0	NOP	
1	ReLU	
2	Sum/Save in local register	accumulate and save to register pointed to by index
3	Sum/Save in common register	accumulate with common register and save
4	Exponent	performed on each valid StOp result
5	Divide with common register	denominator is the common register
6	Reciprocate common register	save to same common register [idx]
7	Compare with common register	save max to same common register [idx]
8	Send	Local registers sent to upstream
9	Send null	Send a null packet upstream
10	Clear local register	
11	Clear common register	
12	Clear index register	
13	Increment index register	no data operations
14-15		NA

Table 5.3 SIMD wrapper special function codes

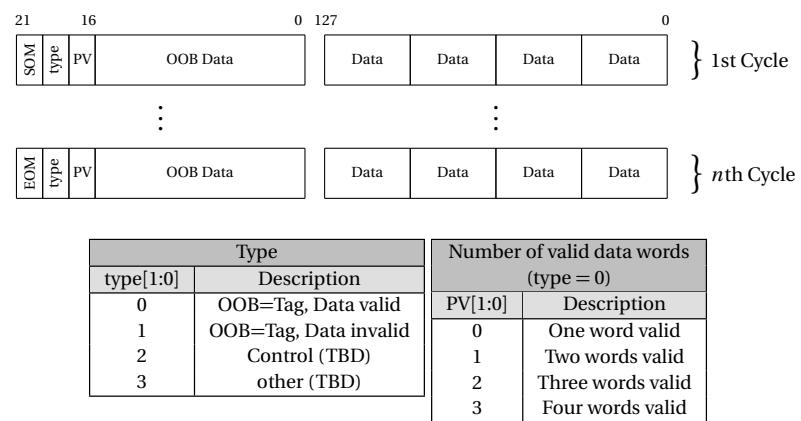


Figure 5.14 Upstream data transactions

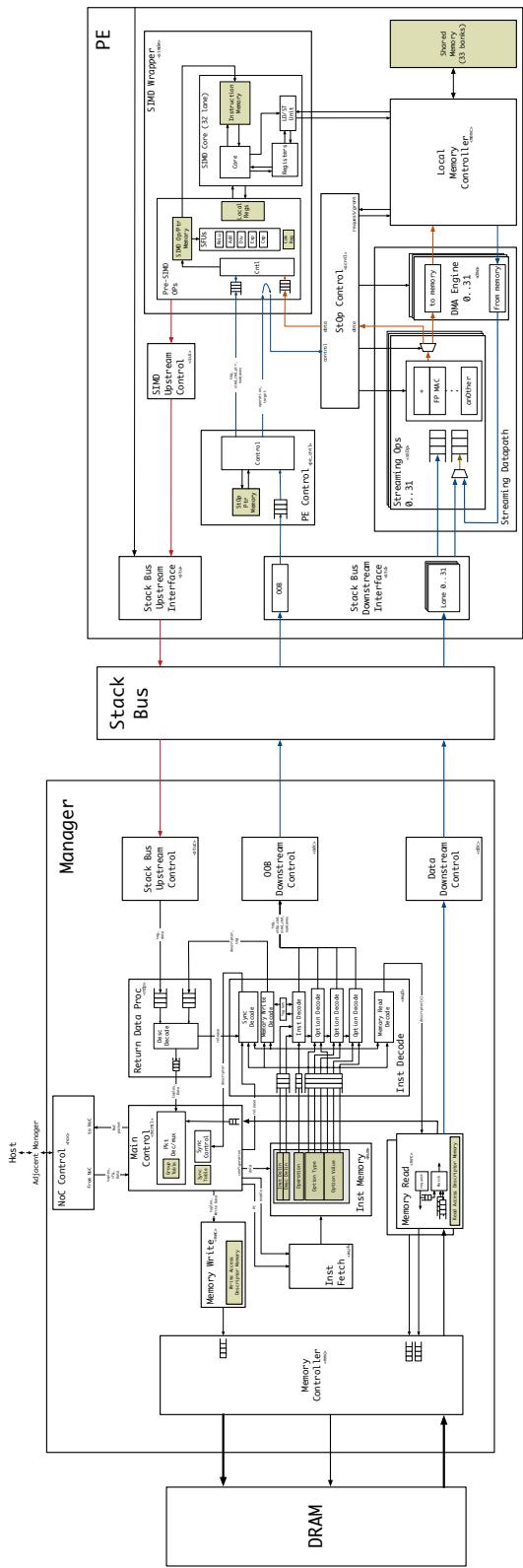


Figure 5.15 Sub-System Column (SSC) Block Diagram

CHAPTER

6

RESULTS

The objective of this work was to design a system able to accelerate multiple disparate ANNs in embedded systems. Given that these systems cannot effectively utilize SRAM, the main objective was to demonstrate a system that can operate efficiently using a customized 3D-DRAM with a wide data-bus.

3DIC technology was exploited, including 3D-DRAM, because a major theme of this work is that 3DIC provides many benefits, including reduced energy use, lower area requirements, and high bandwidth. The bandwidth boost can be significantly increased by using a customized DRAM. Therefore, it was necessary to show that the proposed system can maintain the required data bandwidth while staying within the physical footprint of the 3D-DRAM.

The target technology node chosen was 28 nm, mainly because it's the technology node employed for some recent GPUs and other ASICs such as [23]. The design was synthesized using 65 nm libraries and then scaled to 28 nm. This was necessary because certain library components were unavailable

in 28 nm.

6.1 Power and Area Scaling estimates

Some representative portions of the design were synthesized using 28 nm libraries to obtain scaling numbers. The area and power results from these synthesis runs are shown in Table 6.1. The final scaling numbers are shown in Table 6.2.

Node	Type	Frequency	Internal	Net switching	Leakage
65 nm	Logic	100 MHz	66.9 mW	1.53 mW	2.02 nW
28 nm	Logic	100 MHz	13.2 mW	1.26 mW	4.9 μ W

(a) Example logic only design synthesis power

Node	Type	Frequency	Internal	Net switching	Leakage
65 nm	Memory	100 MHz	2.36 mW	0.6 μ W	5.4 pW
28 nm	Memory	100 MHz	43.8 μ W	NR	443 μ W

(b) Example design with memory synthesis power

Node	Type	Area from synthesis	Area from Datasheet
65 nm	Logic	879593 μ m ²	NA
28 nm	Logic	327822 μ m ²	NA
65 nm	Memory	210230 μ m ²	57014 μ m ²
28 nm	Memory	75605 μ m ²	20281 μ m ²

(c) Example design area

Table 6.1 Example design area and power

Logic Area	Memory Area	Logic power			Memory power		
		Internal	Net switching	Leakage	Internal	Net switching	Leakage
2.68	2.78	5.07	1.21	4.12×10^{-4}	5.07 ^a	NA	NA

^athis number is conservative based on Table 6.1b

Table 6.2 68 nm to 28 nm scaling numbers

6.2 Physical Placement

Based on the DiRAM4 layout shown in Figure 3.4, the dimensions of the area available for each SSC are $1470\text{ }\mu\text{m}$ by $1656\text{ }\mu\text{m}$ or approximately 2.43 mm^2 . Using the logic and memory scaling numbers from Table 6.2, the effective dimensions at 65 nm is $2431\text{ }\mu\text{m}$ by $2738\text{ }\mu\text{m}$ providing a design area at 65 nm of approximately 6.66 mm^2 . The area contribution of each block within the Manager and PE can be seen in Table 6.3.

Table 6.3 Area contribution

Block name	Instances	Percentage contribution
Memory Controller	1	20.6 %
NoC	1	6.9 %
Read Control	2	47.1 %
Write Control	1	6.7 %
Instruction Proc	1	1.7 %
Return Data Proc	1	1.6 %
System Controller	1	1.6 %
TSV	NA	6.9 %
Misc	NA	6.8 %

(a) Manager

Block name	Instances	Percentage contribution
Operation Decode	1	3.1 %
Return Data Control	1	1.5 %
SIMD Wrapper	1	15.1 %
SIMD	1	17.9 %
Streaming Operations	32	40.0 %
Streaming Op Control	1	1.9 %
Local Memory + Control ^a	1	16.4 %
TSV	NA	3.7 %
Misc	NA	0.4 %

(b) PE

^aA small amount of scratchpad memory was provided between StOps and SIMD but in practice could be much smaller. It is not used in any of the fanin tests.

The layout utilization for the manager and PE were 82.3 % and 53.6 % respectively. Those numbers

strongly suggest the manager was more challenging to place and route. To get a sense of the physical feasibility of the system, both the manager and PE were placed and routed without design rule check (DRC) using Synopsys® IC Compiler™. A route congestion analysis was performed which indicated very few congested areas, which was most likely due to the wide datapath nature of the design. A preliminary place and route for the Manager and PE are shown in Figure 6.1. The physical placement and congestion of the manager and PE suggests a relatively low risk of encountering problems in completing a full detailed place and route.

6.3 Synthesis

An example logic-only design was synthesized using the 28 nm and 65 nm libraries to determine the appropriate frequency scaling between the two technology nodes. In each case, the frequency was increased until negative timing slack was observed. This suggested that to achieve an operating frequency of 500 MHz at 28 nm, the design should be synthesized at 65 nm using a frequency of 193 MHz. All blocks in the system were synthesized using a typical library and no blocks had issues synthesizing using the typical library at this relatively low frequency, suggesting a target frequency of 500 MHz at 28 nm is relatively low risk.

6.4 Logic Verification

The primary control and data paths of the system have been simulated in a SystemVerilog environment using Mentor Graphics® ModelSim™. To ensure high bandwidth utilization can be maintained, a group of tests representing convolutional and fully-connected layers was simulated. The operations simulated were based on the expected lower and upper limits of pre-synaptic fanin. These test cases were based on layers similar to CONV2 and FC-7 from [27] which represent a pre-synaptic fanin of 225 and 4000 respectively. Additional test cases were employed representing pre-synaptic fanins of 294, 300, 500 and 1000. Both locally-connected (CONV) and fully-connected (FC) type fanins were tested. The tests labeled CONV-500 and FC-500 represent convolutional and fully-connected tests

respectively, both with a pre-synaptic fanin of 500. The results showing sustained average bandwidth can be seen in Table 6.4.

Table 6.4 Fanin bandwidth tests

Test	Average Bandwidth at frequency	
	500 MHz	700 MHz
CONV2 [27]	~25 Tbit/s	~35 Tbit/s
CONV-294	~26 Tbit/s	~37 Tbit/s
CONV-300	~27 Tbit/s	~37 Tbit/s
CONV-500	~29 Tbit/s	~41 Tbit/s
CONV-1000	~31 Tbit/s	~43 Tbit/s
CONV-2500	~32 Tbit/s	~45 Tbit/s
FC-350	~28 Tbit/s	~39 Tbit/s
FC-500	~29 Tbit/s	~41 Tbit/s
FC-1000	~31 Tbit/s	~43 Tbit/s
FC-7 [27]	~32 Tbit/s	~45 Tbit/s

Considering the baseline system shown in Figure 1.1, the distribution of ANe operations and expected bandwidth are shown in Table 6.5.

Table 6.5 Baseline ANN expected bandwidth

Layer	Operation fanin	Equivalent test	Percentage of operations	Expected bandwidth
1	363	CONV-300	44%	11.7 Tbit/s
2	4		Performed during previous layer	
3	2400	CONV-2500	28%	9.1 Tbit/s
4	4		Performed during previous layer	
5	2304	CONV-1000	10%	3.0 Tbit/s
6	3456	CONV-2500	10%	3.2 Tbit/s
7	3456	CONV-2500	7%	2.1 Tbit/s
8	43264	FC-7	1%	0.2 Tbit/s
9	4096	FC-7	1%	0.3 Tbit/s
10	4096	FC-7	<1%	0.1 Tbit/s
			Total	29.7 Tbit/s

6.5 Power Estimates

To estimate power consumption, parasitics were extracted from the preliminary layouts and simulated against CONV2. The activity file generated by this simulation was used by the Synopsys® Primetime-PX™ power analysis tool to obtain power and bandwidth estimates. The DRAM accesses were captured and DRAM power dissipation calculated from [14]. The power dissipated in the TSVs was estimated from [30]. These estimates were scaled using the scaling numbers from Table 6.1 and used to estimate power dissipation for operating frequencies of 500 MHz and 700 MHz using a 28 nm technology node. The estimated overall power and per-block contributions are shown in Table 6.6.

Table 6.6 Power Estimates

Technology node	Clock frequency	Total expected power	Testcase
28 nm	500 MHz	75.4 W	CONV-294
28 nm	700 MHz	101.2 W	CONV-294

(a) Power Dissipation

Block name	Percentage contribution
Manager	56.4 %
PE	35.1 %
DRAM	6.0 %
DRAM TSVs	1.5 %
Stack Bus TSVs	1.0 %

(b) Power contribution

6.6 Summary

The bandwidth performance shown in Table 6.4 shows a sustained average bandwidth that meets the requirement shown in Table 1.2. In a full system with data transfers between a host and the SSC, there will be idle times but this should have a relatively low impact as it involves mostly input download

and result upload. The focus of the verification environment was the ANe processing which involves the most complex operations in the system, including memory management, flow control, etc. The list of features implemented and verified is shown in Table 6.7.

Feature	Limitations		Method	Comment			
Major processing datapaths	N		Self checking	Instructions generated using python scripts Instructions manually generated			
SIMD Wrapper special functions	N		Visual				
Sync Send	Host only						
Sync Wait	Host only						
Memory Upload	Host from DRAM						
Memory Download	Solicited	Host to DRAM					
	Unsolicited						
BootP	N						

Table 6.7 System features implemented

The design was conservatively coded to ensure there were opportunities for logic reduction. For example, in all the modules FIFOs were employed at primary interfaces and the depths of those FIFOs were generous. In the design of FSMs, the number of states was assigned conservatively. The major interfaces between blocks are all registered, but this is considered a requirement when designing a system of this size. The area scaling numbers employed are within a reasonable range [40]¹. Therefore the place and route feasibility study suggests that embarking on productizing a system based on this work would be relatively low risk.

The power numbers were higher than expected with a power per unit area of 395 mW/(mm² GHz) but comparable to the 389 mW/(mm² GHz) from [11] and higher than the 304 mW/(mm² GHz) from [2] and the 191 mW/(mm² GHz) inferred from [23]. Therefore there may either be opportunities for power reduction – such as clock gating – or a fully completed place and route may result in lower power numbers.

¹work done by Schabel et al, yet to be published

6.6.1 Recent state-of-the-art comparison

When making comparisons it is important to consider whether a solution meets the requirements and not how high it performs against a particular metric. This work places emphasis on meeting the requirements and not on maximizing a particular metric.

Both Neurostream [2], NeuroCube [24] and Tetris [17] employ HMC 3D-DRAM to provide capacity and a main memory bandwidth of 1 Tbit/s. Neurostream [2] obtained 22.7 GOP/s/W and claimed a 3.5X performance improvement over a representative GPU while using the same binary32 number format as this work. To achieve this performance Neurostream must use its 1 Mbit SRAM because the DRAM could only support a sustained 8 GFLOPS.

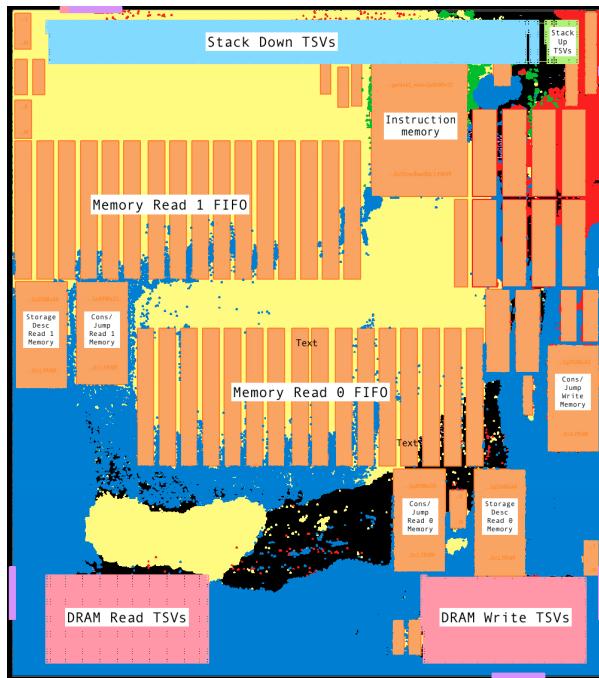
Considering that DRAM bandwidth is the bottleneck, for Neurostream [2] to meet the bandwidth requirements of this work's target application, it would have to instantiate tens of devices. This would result in a power dissipation in excess of 1 kW. The floating point operations (FLOPs) performance and capacity would be impressive but it would far exceed the requirements of the system. Note that this work achieves 22.4 GOPs/s/W, which is similar to Neurostream [2].

Although both NeuroCube [24] and Tetris [17] do not support the same binary32 number format, we can still consider the DRAM bandwidth as being the bottleneck. To meet the bandwidth requirements of this work's target application, they would have to instantiate 32 devices. Even though the power numbers of both NeuroCube [24] and Tetris [17] do not represent a binary32 system, the resulting power dissipation of 223 W and an area of 544 sq. mm still represent an improvement from this work of 3X and 6X respectively.

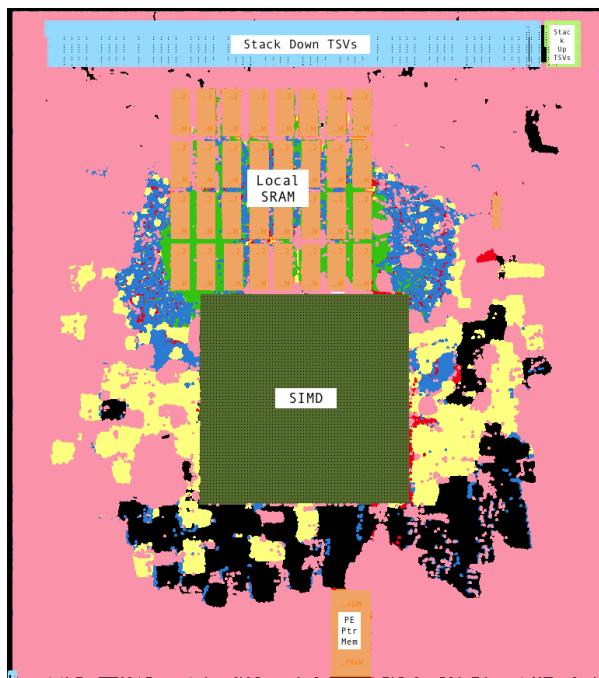
A similar scenario occurs when comparing Tensor processing unit [23] although the number formats are very different and [23] acknowledges the system relies on batch processing to achieve the very high performance. It uses DRAM with a bandwidth of 288 Gbit/s, which means this work provides greater than 90 times improvement. It should be noted in the case of [23] this is an unfair comparison as they do acknowledge the device relies on batch processing and ANN parameter “reuse” and they also state performance degrades when running fully-connected ANNs and therefore this device is not appropriate for this work's target applications.

The DaDianNao [11] device achieves a higher local memory density by employing EDRAM instead of SRAM. A single device has 288 Gbit of EDRAM with a bandwidth of 2.48 Tbit/s. To approach the bandwidth of this work would require ten devices dissipating 167 W providing this work a 2.2X improvement. To approach the storage provided by this work would require over two hundred devices. In fact [11] states that to implement one of their example ANNs, which requires 1.39 GB of storage would require a 36-node system. They also reference a 64-node system that they claim would outperform a GPU by 450X. So, even assuming a 64-node DaDianNao system, this work would provide a 25X area improvement and a 14X power dissipation improvement.

The NeuroCube [24], Tetris [17], Neurostream [2] and DaDianNao [11] most closely match this work and they offer significant speed improvements over GPU. This work provides at least a 6X power improvement and a 3X area improvement over this state of the art, in the target application.



(a) Manager



(b) PE

Figure 6.1 Manager and PE Die layouts

CHAPTER

7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

There have been many attempts to accelerate ANNs. Many have shown excellent performance mainly when implementing CNNs. The improvement mostly comes from the ability to hold kernel weights and/or ANe states in local SRAM. Another method of employing local memory is often due to pooling of batch requests, especially in server applications. This local storage allows the system to take advantage of the low latency and random access benefits of SRAM while performing multiple operations on that data. When considering applications where this local storage cannot be used effectively, all these implementations suffer a degradation in performance.

This work considers embedded applications where a system is processing requests with a disparate set of ANNs. A further assumption is that in embedded applications there are restrictions on power consumption and area. In this type of target application local SRAM would not be effective

and performance is based on DRAM bandwidth. This work considers Three-Dimensional Integrated Circuit (3DIC) technology and a customized Three-dimensional dynamic random-access memory (3D-DRAM) is proposed. The customized 3D-DRAM combined with a design based on custom instructions and operation descriptors allows the system to achieve high levels of usable memory bandwidth. There is no doubt existing CNN accelerators that take advantage of batch processing achieve a performance that is difficult to better, but applying these systems to this work's target application exposes those systems' DRAM bandwidth limitations. This work demonstrates a 3DIC system that at the surface provides relatively low floating point operations per second (FLOPS), but considering the target application is memory bound, this work potentially provides a 10-50X improvement over existing ASICs/ASIPs or GPUs. In reality, it is difficult to compare performance against existing GPU or ASIC solutions because they target different application spaces.

7.2 Future Work

This work focused on providing the infrastructure for an expansive architecture. The design infrastructure implemented was designed to allow additional functionality to be added without a high logic area impact to maintain the validity of the current area study. This work focused on the ANe state calculations and assumed instructions and configuration tables are pre-loaded. Therefore future work should include adding the data transfer functionality described in Section 3.

There is a level of acceptance that in some cases, lower precision is acceptable in ANN inference. Therefore further work should include manager and PE changes to support half-precision floating-point (binary16). In the manager, supporting binary16 would require additional muxing logic when directing words in the wide DRAM bus to execution lanes as shown in 5.1.4, but the bulk of the design should remain relatively intact. Supporting binary16 in the PE would be relatively straightforward.

This work does not put a high level of importance on the PE as the functionality provided by the PE is relatively straightforward and primary emphasis was ensuring the PE fits within the 3DIC footprint. However, there is opportunity to research different PE architectures such as systolic arrays and a more function-rich SIMD.

This work focused on providing an array of SSCs to match the array of DRAMs interfaces provided by the DiRAM4, but further research should include ganging DRAM interfaces to a coarser array of SSCs. In practice, this may also be synergistic with alternative PE architectures, such as employing a PE with a large systolic array [23].

The instruction decode logic is currently coded using an FSM. In accordance with creating an expansive architecture it might be a better to use a small processor to decode and control system functions.

The NoC employed in this work was primarily to create a representative NoC for the area and power analysis. Therefore, there are opportunities to explore alternative NoC designs including dynamically changing the NoC routing table to meet the processing needs of different SSC assignments. This includes considering the routing of the ANe state replication associated with a particular operations.

BIBLIOGRAPHY

- [1] Aizenberg, I. et al. *Multi-Valued and Universal Binary Neurons: Theory, Learning and Applications*. Springer Science & Business Media, 2013.
- [2] Azarkhish, E. et al. “Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes”. *arXiv preprint arXiv:1701.06420* (2017).
- [3] Bamberg, L. & Garcia-Ortiz, A. “High-Level Energy Estimation for Submicrometric TSV Arrays”. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **25**.10 (2017), pp. 2856–2866.
- [4] Brette, R. “Philosophy of the Spike: Rate-Based vs. Spike-Based Theories of the Brain”. *Frontiers in Systems Neuroscience* **9** (2015), p. 151.
- [5] Brunel, N. & Rossum, M. C. W. van. “Lapicque’s 1907 paper: from frogs to integrate-and-fire”. *Biological Cybernetics* **97**.5 (2007), pp. 337–339.
- [6] Bullinaria, D. J. A. *Introduction to Neural Computation : Neural Computation*. <http://www.cs.bham.ac.uk/~jxb/inc.html>. Accessed: 2017-09-08.
- [7] Carnevale, N. & Hines, M. *The NEURON Book*. Cambridge University Press, 2006.
- [8] Chen, T. et al. “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning”. *ACM Sigplan Notices*. Vol. 49. 4. ACM. 2014, pp. 269–284.
- [9] Chen, Y. et al. “DaDianNao: A Machine-Learning Supercomputer”. *47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014, pp. 609–622.
- [10] Chen, Y.-H. et al. “14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks”. *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE. 2016, pp. 262–263.
- [11] Chen, Y. et al. “DianNao Family: Energy-efficient Hardware Accelerators for Machine Learning”. *Commun. ACM* **59**.11 (2016), pp. 105–112.
- [12] *DDR3L SDRAM*. D 8/17 EN. Micron Technology Inc. 2015.
- [13] Deeplearning4j Development Team. *Introduction to Deep Neural Networks*. <http://deeplearning4j.org>. Accessed: 2018-01-02.
- [14] *DiRAM4-64Cxx Cached Memory Subsystem*. Rev. 0.04. Tezzaron Semiconductor. 2015.
- [15] Esmaeilzadeh, H. et al. “NnSP: embedded neural networks stream processor”. *48th Midwest Symposium on Circuits and Systems, 2005*. IEEE. 2005, pp. 223–226.

- [16] Farabet, C. et al. “Neuflow: A runtime reconfigurable dataflow processor for vision”. *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE. 2011, pp. 109–116.
- [17] Gao, M. et al. “Tetris: Scalable and efficient neural network acceleration with 3d memory”. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2017, pp. 751–764.
- [18] Hinton, G. E. et al. “A Fast Learning Algorithm for Deep Belief Nets”. *Neural Computation* **18**.7 (2006). PMID: 16764513, pp. 1527–1554. eprint: <https://doi.org/10.1162/neco.2006.18.7.1527>.
- [19] Hochreiter, S. & Schmidhuber, J. “Long short-term memory”. *Neural computation* **9**.8 (1997), pp. 1735–1780.
- [20] ITRS. *International Technology Roadmap for Semiconductors 2.0, Interconnect*. 2015.
- [21] Izhikevich, E. M. “Which model to use for cortical spiking neurons?” *IEEE Transactions on Neural Networks* **15**.5 (2004), pp. 1063–1070.
- [22] Jacob, B. et al. *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.
- [23] Jouppi, N. P. et al. “In-datacenter performance analysis of a tensor processing unit”. *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM. 2017, pp. 1–12.
- [24] Kim, D. et al. “Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory”. *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE. 2016, pp. 380–392.
- [25] Kim, S. W. et al. “Ultra-Fine Pitch 3D Integration Using Face-to-Face Hybrid Wafer Bonding Combined with a Via-Middle Through-Silicon-Via Process”. *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*. 2016, pp. 1179–1185.
- [26] Krizhevsky, A. et al. *ImageNet Classification with Deep Convolutional Neural Networks*. <http://image-net.org/challenges/LSVRC/2012/supvision.pdf>. Accessed: 2016-08-30.
- [27] Krizhevsky, A. et al. “Imagenet classification with deep convolutional neural networks”. *Advances in neural information processing systems*. 2012, pp. 1097–1105.
- [28] Kwolek, B. “Face Detection Using Convolutional Neural Networks and Gabor Filters”. *Artificial Neural Networks: Biological Inspirations – ICANN 2005: 15th International Conference, Warsaw, Poland, September 11-15, 2005. Proceedings, Part I*. Ed. by Duch, W. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 551–556.

- [29] Le, Q. V. "Building high-level features using large scale unsupervised learning". *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 8595–8598.
- [30] Liu, Y. et al. "A compact low-power 3D I/O in 45nm CMOS". *2012 IEEE International Solid-State Circuits Conference*. IEEE. 2012, pp. 142–144.
- [31] Luo, T. et al. "DaDianNao: A Neural Network Supercomputer". *IEEE Transactions on Computers* **66**.1 (2017), pp. 73–88.
- [32] Maas, A. L. et al. "Rectifier nonlinearities improve neural network acoustic models". *Proc. ICML*. Vol. 30. 1. 2013.
- [33] Maddison, C. J. et al. "Move evaluation in go using deep convolutional neural networks". *arXiv preprint arXiv:1412.6564* (2014).
- [34] Nielsen, M. *NNs and DL*. <http://www.neuralnetworksanddeeplearning.com/index.html>. Accessed: 2018-01-02.
- [35] Nvidia®. *NVidia Tesla P100 Datasheet*. <http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-datasheet.pdf>. Accessed: 2017-12-29.
- [36] Patti, R. "2.5D and 3D Integration Technology Update". *Additional Conferences (Device Packaging, HiTEC, HiTEN, & CICMT) 2014*. DPC (2014), pp. 1–35. eprint: <https://doi.org/10.4071/2014DPC-ta12>.
- [37] Paugam-Moisy, H. & Bohte, S. "Computing with spiking neuron networks". *Handbook of natural computing*. Springer, 2012, pp. 335–376.
- [38] Qiu, Q. et al. "A parallel neuromorphic text recognition system and its implementation on a heterogeneous high-performance computing cluster". *IEEE Transactions on Computers* **62**.5 (2013), pp. 886–899.
- [39] Schabel, J. et al. "Processor-in-memory support for artificial neural networks". *Rebooting Computing (ICRC), IEEE International Conference on*. IEEE. 2016, pp. 1–8.
- [40] Schabel, J. C. et al. *Predictive energy-Per-Op scaling for exploring the design space*. Tech. rep. Box 7911, Raleigh, NC 27695-7911, 2014.
- [41] Standard, JEDEC. "DDR3 SDRAM Standard". *JESD79-3, Jun* (2007).
- [42] Taigman, Y. et al. "DeepFace: Closing the Gap to Human-Level Performance in Face Verification". *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2014.
- [43] Various. *Backpropagation*. <https://en.wikipedia.org/wiki/Backpropagation>. Accessed: 2018-01-02.

- [44] Various. *Neuron*. <https://en.wikipedia.org/wiki/Neuron>. Accessed: 2018-01-02.
- [45] Various. *Neuron*. https://en.wikipedia.org/wiki/Deep_learning. Accessed: 2018-01-02.
- [46] Various. *Pooling Overview*. <http://ufldl.stanford.edu/tutorial/supervised/Pooling/>. Accessed: 2018-02-16.
- [47] Various. *Softmax function*. https://en.wikipedia.org/wiki/Softmax_function. Accessed: 2018-01-02.
- [48] Various. *Softmax regression*. <http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/>. Accessed: 2018-02-16.
- [49] Various. *Stochastic gradient descent*. https://en.wikipedia.org/wiki/Stochastic_gradient_descent. Accessed: 2018-01-02.

APPENDICES

APPENDIX

A

SOFTMAX IMPLEMENTATION

In classifiers, the last layer is often implemented using a softmax [47][48] function. The outputs of this layer represent probabilities of each class/output. As seen in Figure A.1, the classifier ANe state calculation involves a MAC of the pre-synaptic ANe states followed by an exponent. The final ANe state is the exponent of individual ANes divided by the sum of all ANes exponent value.

This work calculates the classifier ANe states by separating the classifier into multiple layers as shown in Figure A.2. The single classifier layer is implemented using three layers. The first layer determines the exponent of each ANe. This is accomplished using the StOp MAC operation followed by the SIMD ReLu and exponent SFUs. The ANe states of this layer are sent back to main memory. The next layer is a single ANe calculation, as described in Section 3.3 with the pre-synaptic weights set to unity. The resulting calculation is an accumulation of the ANe exponent values. The SIMD divider SFU is then used to perform a reciprocal and the result is placed in a SIMD register. The final layer is calculated using the StOp to perform a multiplication of each pre-synaptic ANe state with the

reciprocal value held in the SIMD register, effectively performing the division in the softmax function.

Performing the softmax function is relatively inefficient, however the time taken is masked by the time taken to perform the initial classifier layer ANe state calculation. An example calculation sequence is shown in Figure A.4 using the baseline ANN which employs one thousand classifiers with four thousand pre-synaptic ANEs.

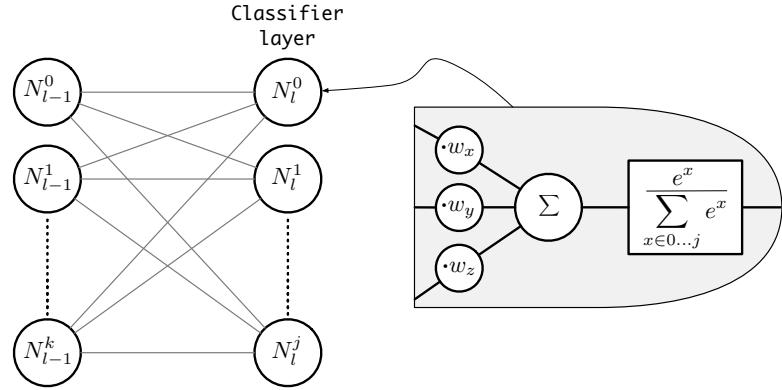


Figure A.1 Classifier layer

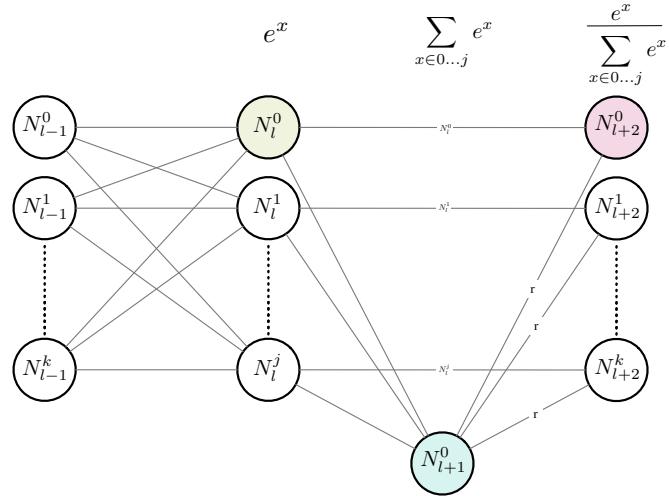


Figure A.2 Classifier additional implementation layers

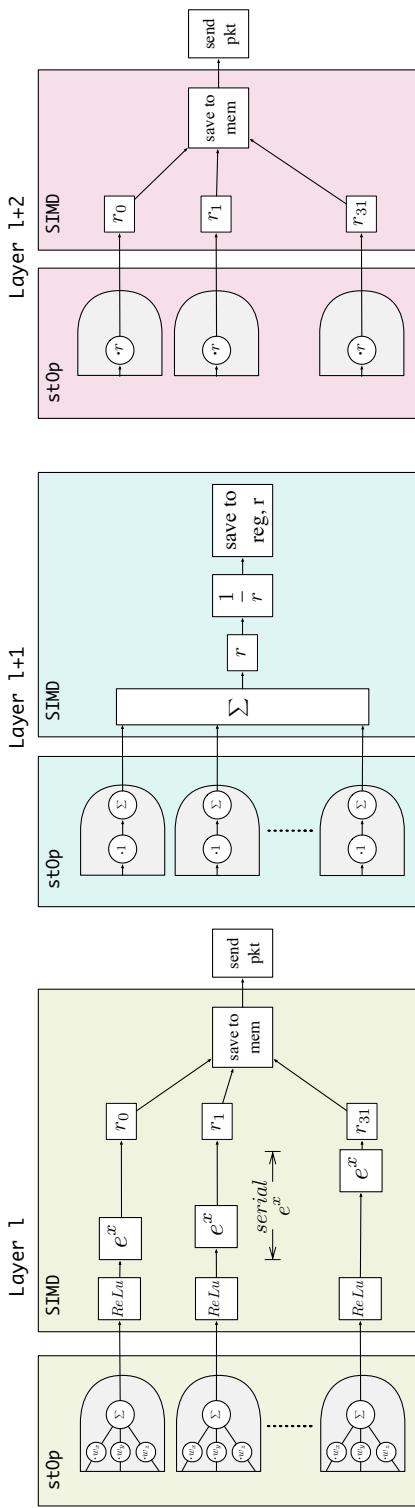


Figure A.3 Classifier layer stOp/SIMD implementation

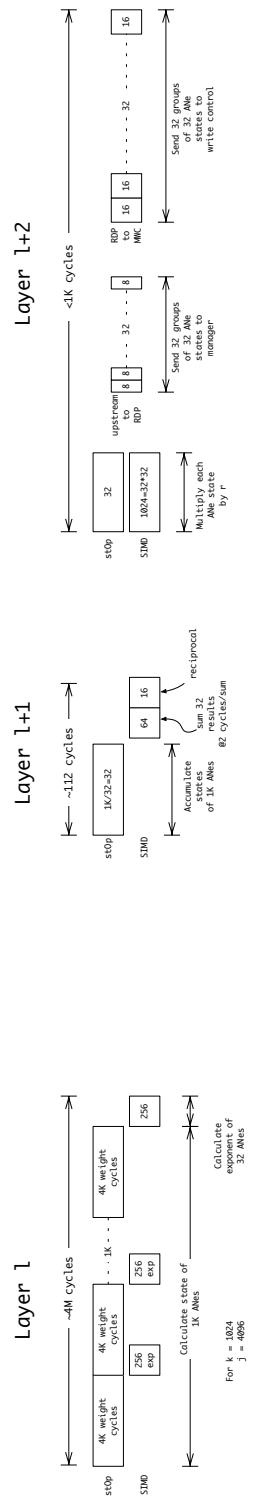


Figure A.4 Classifier layer stOp/SIMD sequence timing

APPENDIX

B

MAX POOLING IMPLEMENTATION

In DNNs, there are “pooling”[46] layers which perform an operation on a region of a feature layer. The common pooling operation is max-pooling (see Figure 1.12) where a group of feature ANes are compared and the ANe with the maximum state value is passed to the next layer. The feature ANes compared are a 2x2 region as shown in Figure B.1.

This work calculates the “max pooling” layer by sequencing the ANes calculations such that the pooling ROI group of features are cascaded. A maximum comparison operation is performed on the current and previous ANe state and the maximum value retained. After the final comparison, the result is the next layer’s ANe state. This sequencing is shown in Figure B.2.

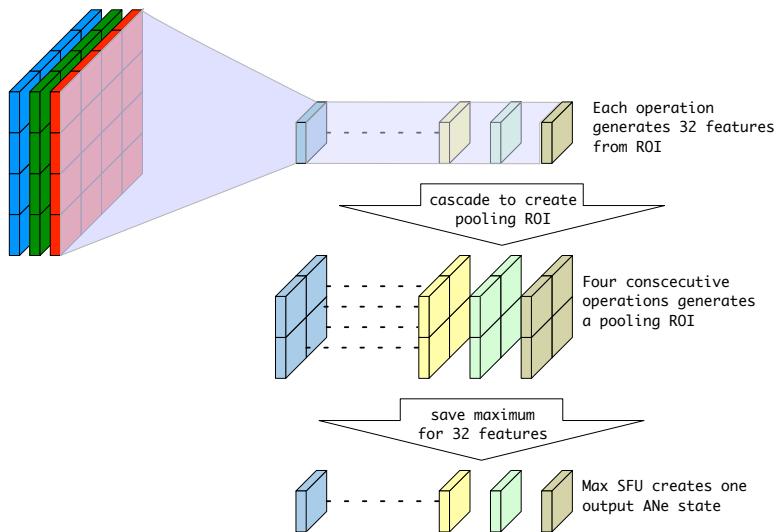


Figure B.1 Classifier additional implementation layers

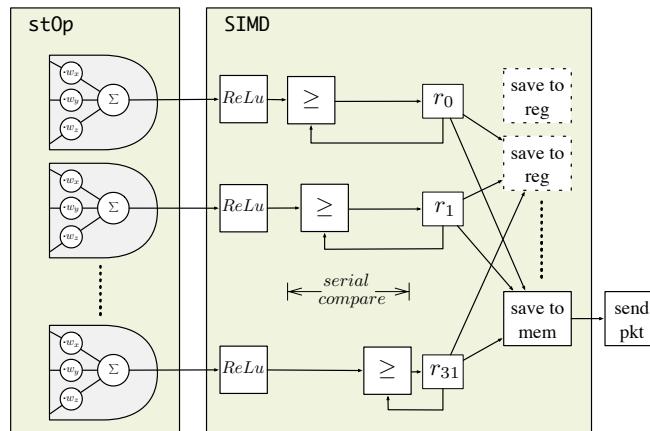


Figure B.2 Pooling implementation

APPENDIX

C

DIRAM4 CHARACTERISTICS

Feature	Value	Comment
Capacity	64 Gbit	1 Gbit/port
Number of ports	64	Each port provides access to an individual memory
Maximum clock frequency	1 GHz	
Number of banks per port	64	
Pages per bank	4096	
Bits per page	4096	
Cycles per read	2	Burst of 2
Cycles per write	2	Burst of 2
Timing		
Name	Value	Comment
t_{POPO}	15 ns	Page open to page open
t_{POCRA}	3 ns	Page open to cache read, aligned to word address
t_{POCR}	9 ns	Page open to cache read
t_{POCW}	9 ns	Page open to cache write
t_{POPC}	9 ns	Page open to page close
t_{PCPO}	10 ns	Page close to page open
t_{CRL}	5 ns	cache read latency
t_{CWL}	-1 ns	cache write latency

Table C.1 DiRAM4 characteristics [14]