INVESTIGATING OPPORTUNITIES AND CHALLENGES IN MODELING AND DESIGNING SCALE-OUT DNN ACCELERATORS

A Dissertation
Presented to
The Academic Faculty

By

Vineet Nadella

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2020

Copyright © Vineet Nadella 2020

INVESTIGATING OPPORTUNITIES AND CHALLENGES IN MODELING AND DESIGNING SCALE-OUT DNN ACCELERATORS

Approved by:

Dr. Tushar Krishna, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Saibal Mukhopadhyay
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Alexandros Daglis
School of Computer Science
Georgia Institute of Technology

Date Approved: April 24, 2020

Do not go where the path may lead, go instead where there is no path and leave a trail.

Ralph Waldo Emerson

This work is dedicated to my parents, my grandparents, and my brother.

ACKNOWLEDGEMENTS

I have gained so much knowledge and have experienced so many wonders of technology over the last 5 years. Firstly, I would like to thank Dr. Tushar Krishna who has inspired me to pursue interesting research topics and for advising me on my journey as a researcher. Even while being one of the prominent Computer Architecture researchers in the world, Dr. Krishna always finds the time to answer questions and provide valuable mentoring. Furthermore, I am grateful for the leadership of Ananda Samajdar, a doctoral student at Georgia Tech for his willingness to teach and give insight on complex subjects even during his busy schedule. He has published multiple paper over the last 3 years earning him status as a top researcher. I wish to see more success from him as he transitions into completing his degree. Additionally, I want to express my gratitude for the opportunity to work with Eric Qin on his groundbreaking publication of SIGMA. He was extremely crucial as I started my journey as a researcher. I would also like to thank Amrita Mathuriya from Kepler Computing for her guidance on my work to develop NoC modeling in SCALE-Sim. She has a strong reputation as a leader and researcher in the industry and has provided me with valuable support. Finally, I would like to thank my family for instilling in me the importance of education and to value learning at a young age. Many of the lesson I learned from my family throughout my life have helped me mature into a strong researcher, engineer, and leader. Even after I graduate, the selfless support presented to me will continue to shape my path to a promising future.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS v

LIST OF TABLES ix

LIST OF FIGURES x

SUMMARY xiii

CHAPTER 1: INTRODUCTION 1

1.1 Definition of Deep Learning, Accelerator, and NoC Terms: 3

1.2 History of Deep Neural Network Accelerators 5

1.3 The Problem 7

CHAPTER 2: BACKGROUND WORK AND DESIGN CONSIDERATIONS 8

2.1 Systolic Array 8

2.2 Dataflows 9

2.3 Convolution Translation 11

2.4 Memory Policies 14

2.5 DNN Layers 17

2.6 Partitioning 20

2.7 NoC Considerations 22

CHAPTER 3: SCALE-SIM-V2: CNN ACCELERATOR SIMULATOR 24

3.1 SCALE-Sim 24

3.2 Simulator Environment 25

3.3 Software Organization 25

3.4 SCALE-Sim Inputs 26

3.5 Operand Matrix Creation 28

3.5.1 Calculated Hyperparameters 28

3.5.2 Address Mapping 29

3.5.3 Batching 30

3.5.4 MNK Operands 30

3.6 SRAM Trace Generation 31

3.6.1 Output Stationary Trace 32

3.6.2 Weight Stationary Trace 34

3.6.3 Input Stationary Trace 36

3.6.4 Dataflow Comparisons 37

3.7 DRAM Trace Generation 38

3.8 NoC Model 39

3.8.1 NoC Modeling Framework 39

3.9 Experiments 42

3.9.1 GPT2 & ResNet-50 Performance Evaluations using NMF 43

3.9.2 GPT2 & ResNet-50 NoC Congestion Evaluation 49

CHAPTER 4: SIGMA BUILDING BLOCKS 52

4.1 Multiplier 53

4.2 Adder 53

4.3 Local Buffer 54

4.4 Control Unit 54

4.5 Flexible Interconnects 55

CHAPTER 5: CONCLUSION AND FUTURE WORK 57

5.1 Future Work 58

5.1.1 C++ Syntax Porting 58

5.1.2 Versatile Compute Architecture Support 58

REFERENCES 59

LIST OF TABLES

Table 2.1: Reuse characteristics of frequently used DNN accelerator dataflows 10

Table 3.1: Example architecture parameters in configuration file 26

Table 3.2: AlexNet CNN with 5 CONV layers 27

Table 3.3: Example network topology 32

Table 3.4: *MESH* Adjacency Matrix 41

LIST OF FIGURES

Figure 1.1: Architecture of systolic array based DNN accelerator [38] 6

Figure 2.1: High-level architecture for compute and memory stack 8

Figure 2.2: Read operand elements pulled from left and top of systolic array and write operand elements pushed out from bottom 9

Figure 2.3: Read and write operands after convolution translation 12

Figure 2.4: *Ifmap* operand after skewing (rotated 90° right) 13

Figure 2.5: Initial Prefetch before process execution.. 15

Figure 2.6: Partial flush and replenish of active buffer.. 17

Figure 2.7: Network of systolic arrays each with a perfect interconnect to external memory (DRAM) [21] 21

Figure 3.1: Schematic depicting the inputs needed and the outputs generated by SCALE-Sim [2]. 25

Figure 3.2: High level code organization and flow in SCALE-Sim simulator environment. Arrows provide a rough view of the sequential flow of the tool and do not necessarily represent I/O movement.. 26

Figure 3.3: $(ifmapheight = 4, ifmapwidth = 4)$ & $(filterheight = 3, filterwidth = 3, filternum = 4)$ with $stride = 2$ 29

Figure 3.4: An example activation map with addresses prioritized based on the [*Channel, Height, Width*] scheme... 30

Figure 3.5: Operands compatible with GEMM operations [6] 30

Figure 3.6: 1-fold SRAM trace for output stationary (*OS*) dataflow. Header represents the division in elements from each operand. By convention, *ifmap* and filter elements are read

from SRAM and *ofmap* elements are written to SRAM. Reference Figure 3.7 and 3.8 for comparison. 34

Figure 3.7: 1-fold SRAM trace for weight stationary (*WS*) dataflow. Reference Figure 3.6 and 3.8 for comparison. 36

Figure 3.8: 1-fold SRAM trace for input stationary (*IS*) dataflow. Reference Figure 3.6 and Figure 3.7 for comparison. 37

Figure 3.9: Direct *mesh* NoC topology with 16 PEs each acting as a router in the network. 40

Figure 3.10: NoC Memory Hierarchy 42

Figure 3.11: GPT2 Partitioning Evaluation over BW Sweep 44

Figure 3.12: GPT2 ABP Evaluation over BW Sweep 46

Figure 3.13: GPT2 ABP evaluation over partitioning for a *mesh* and *fully connected* topology with remote bandwidth of 1000 bytes 47

Figure 3.14: Resnet-50 Layers evaluation over partitioned configuration for execution on a *mesh* 48

Figure 3.15: Resnet-50 Layers evaluation over partitioned configuration for execution on a *fully connected* topology 49

Figure 3.16: Resnet congestion evaluation using heat map 50

Figure 3.17: Link load distribution for GPT2 and Resnet50 layers 51

Figure 4.1: bfloat16 bit layout with most significant bit (MSB) on left representing the decimal value $5.0 = -1signbit * 2exponentvalue - 127 * 1 + mantissavalue128; signbit = 0, exponentvalue = 129, \& mantissavalue = 32$ .. 54

Figure 4.2: FAN topology with 32 multipliers, 31 adders, and flexible interconnects for data forwarding in a $32-sized$ Flex-DPE.. 56

SUMMARY

The rapid growth of deep learning used in practical applications such as speech recognition, computer vision, natural language processing, robotics, any many other fields has opened the gate to new technology possibilities [1]. Unfortunately, traditional hardware systems are being stretched to the maximum to accommodate the intense workloads presented by state-of-the-art deep learning processes in a time when transistor technology is not scaling. To serve the demand for better computational power and more specialized computations, specialized hardware needs to be developed that provides better latency and bandwidth specifications for various demanding applications.

The trend in the semi-conductor industry is to move towards heterogenous System-On-Chip (SoC) thereby choosing application specific performance vs. generality seen in most CPU architectures today. In most situations, hardware engineers are left to construct systems that serve the needs of various applications, often needing to predict the use-cases of the system. As with any field, the ability to predict and act on the future innovation trends of the industry is the difference between success and failure.

A novel simulator for the design of convolutional neural network accelerators is presented and described in detail named SCALE-Sim (Systolic CNN Accelerator Simulator). The simulator is available as an open-sourced repository and has 2 primary use-cases in which computer architects can extract significant results. The first use-case is for system designers who would like to integrate an existing DNN accelerator architecture into a larger SoC and would be interested in system-level characterization results. The second use-case is for an accelerator architect who would like to use the tool to explore the accelerator design space by sweeping through design parameters [2].

CHAPTER 1

INTRODUCTION

Deep learning is a fast-growing field of study that has potential for application in many fields such as AI robotics, natural language processing, computer vision [1, 3, 4, 5, 6]. Unfortunately, the present suite of general-purpose processors used in servers and client computing do not have the hardware resources to build complex deep learning networks that achieve near flawless accuracy. Effectively, this means that hardware is the bottleneck in this growing field. To serve the demand for better computational power and more specialized computations, specialized hardware needs to be developed that provides better latency and bandwidth requirements for various applications.

General matrix to matrix multiplication (GEMM) operations are at the essence of neural network processing [7, 8]. Though GPUs have been found to be well-suited for GEMM operations, the regular dataflows pushed to the processor by a deep neural network (DNN) introduces the idea of using specialized hardware. A custom hardware chip used in DNN processing is known as a DNN accelerator. To further discuss the characteristics of a DNN accelerator, it is important to note that custom hardware is achieved using primarily MAC (Multiply and Accumulate) units which are extremely popular for GEMM operations. Moreover, these MAC units are used to exploit algorithmic parallelism and achieve high throughput while performing inference [7].

The question left to analyze for a DNN accelerator is how an architect can organize the compute and memory components on-chip and off-chip to fully take advantage of the specific networks used in inference. Unfortunately, this is a non-trivial question and involves many workload and architecture specific parameters making it much too difficult

to answer without a tool that can analyze the suite of variables before delivering performance results.

Convolution Neural Networks (CNN) have been found to be extremely useful in image classification and analysis, natural language processing, recommender systems, financial time series, and many more applications [9]. A single convolutional layer in a larger CNN has three primary components in execution. Two of the components are read into execution: *ifmap* (input feature map) and the filter. The filter is used to convolve the *ifmap* in a series of sliding GEMM operations to produce the *ofmap* which is either the final output of the CNN or the intermediate resulting *ifmap* for the next layer. Since CNNs are arguably the most popular and most compute and memory intensive neural network, focus of the team efforts were placed in solving the question posed above specifically for this subset of DNNs.

1.1 Definition of Deep Learning, Accelerator, and NoC Terms:

Many terms and acronyms are referenced in the detailed discussion throughout this document. This section can be referenced for clarity on specific technical term and acronym definitions.

- Deep Neural Networks (*DNN*): A class of neural network techniques with multiple hidden layers between input and output layers in the field of deep learning (*DL*) which is part of the larger field of machine learning (*ML*).
- Convolutional Neural Network (*CNN*): A subset of deep neural networks which have been found useful in visual data learning and inference. CNNs have multiple layers consisting of convolutional (*CONV*) layers, activation layers (*RELU, POOL*) and fully connected (*FC*) layers [10].
- Fully Connected Layer (*FC*): An execution layer used in DNNs whose output size corresponds to the number of classification labels. This is usually the last layer in a classification task.
- Multiply-Accumulate (*MAC*): A combination of multiplication operations followed by the accumulation of the multiplication products into a single sum which is the basis of matrix multiplication [1].
- General Matrix-Matrix Multiplication (*GEMM*): Common algorithm in machine learning that can be executed on *MAC* units.
- Single Instruction Multiple Data (*SIMD*): A class of parallel computers performing the same operation on multiple data elements.
- Arithmetic Logic Unit (*ALU*): A microprocessor component used for the implementation of arithmetic and logic operations [11].

- Field Programming Gate Array (*FPGA*): An integrated circuit designed to be configured by a hardware designer after the manufacturing stage [12].
- Input Feature Map (*Ifmap*): A set of structured 2-D maps or channels consisting of input activations of a layer [1].
- Output Feature Map (*Ofmap*): A set of structured 2-D maps or channels consisting of output activations of a layer [1].
- Filter: A structured 3-D map consisting of weights of a layer with one or more channels of activations [1].
- Weight Stationary (*WS*): A dataflow designed to minimize energy consumption of reading weights by maximizing weight reuse [1].
- Input Stationary (*IS*): A dataflow designed to minimize energy consumption of reading input activations [1].

  Output Stationary (*OS*): A dataflow designed to minimize the energy consumption of reading and writing partial sums [1].
- Non-Uniform Memory Access (*NUMA*): Phenomenon that memory at various points in the address space of the processor have different performance characteristics [13].
- Network-on-Chip (*NoC*): A network-based data communication subsystem between on-chip nodes [14].
- Packet & Flit: Packet is a data container containing a header and payload used for data sharing in computer networks. Packets can be broken down into link-level messages called flits [15].

1.2 History of Deep Neural Network Accelerators

The idea to create highly efficient systems intended to accelerate deep neural network processes has been relevant since the 1990s. One of the first DNN accelerators was presented in 1991 named ANNA. The chip used mixed analog/digital computation techniques to speed up ALU computations while still retaining the advantages of digital interfacing to various components. Even with an archaic CMOS technology capable of about 100 times less transistor power compared to a present-day chip, the advantages of creating a specialized chip to attack the problem of neural network processing outweighed computational power to researchers. As transistor technology continued to scale up exponentially and the neural network computing required stayed virtually even, many of the advantages of specialized hardware started to fade while general CPUs and FPGA platforms gained interest. One example of an early attempt was for the implementation of Hofield neural neural network processing in FPGAs in 1996. Many architectures have been presented in the 2000s, leveraging the programmability of FPGAs with the speed and throughput available in more specialized systems.

Interestingly, in the late 2000s and in the bulk of 2010s, the applications for which modern neural network processes could be applied to greatly increased in number. Previously thought of as a method to learn simple processes, deep neural networks were proven to learn complex functions many times without any real context to the function other than ample amount of training data to learn from. With the increased demand in complex neural network processing, the same FPGA and CPU systems popular early on did not have the latency and throughput requirements needed to perform learning and

inference tasks in real-time. Parallel computing on SIMD computers became increasingly popular, arguably the most popular being GPU by Nvidia.

Today SIMD processing is the mainstay for most DNN processing; however, convolutional layers in CNNs have been shown to require exponentially more computing power than provided on even the best GPU designs. Systems focusing on convolution inference acceleration have been the topic of interest among DNN acceleration researchers. DianNao was introduced in 2014 as a highly efficient, small footprint DNN accelerator capable of performing convolutional inference at the edge [16]. Industry leaders such as Google and Xilinx have taped-out DNN accelerators named Google TPU and Xilinx FPGA overlays xNN, respectively, using the compute architecture known as a systolic array for convolutional inference [17, 8].

As more advanced designs are the focus of industry and academia, the demand to speed up the process of innovation is robust. The future sections will describe the tool (SCALE-Sim) developed to speed up the process of accelerator design as well as the results found from preliminary simulator use.

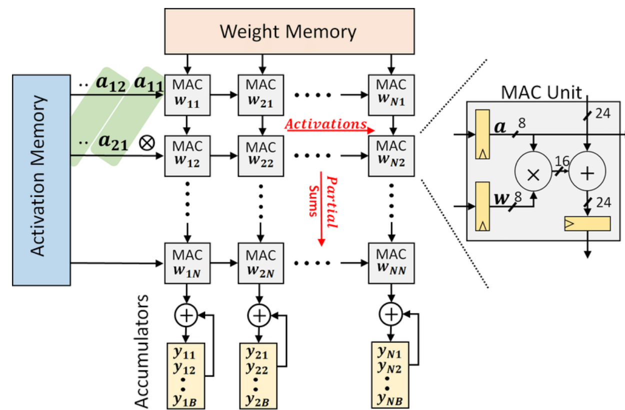


Figure 1.1: Architecture of systolic array based DNN accelerator [38]

1.3 The Problem

As described in the introduction, a key aim of researchers and computer architects is designing architectures that optimize for high performance at low cost. For architectures involving only a few design parameters, an empirical procedure can be used to conceive a strong solution; however, as the number of variables is scaled up, this becomes a difficult problem to solve empirically.

In computer architecture, parameters such as memory placement, memory sizing, processor design, instruction-set-architecture are modified depending on architectural constraints and workload specifications [18]. Specifically, for deep learning workloads, optimization of architectural parameters is crucial for an effective design because of high memory bandwidth and compute requirements [19]. Unfortunately, an open-sourced tool for optimization does not exist for the deep learning community to model workloads on accelerator architectures. This predicament is one of the reasons for difficulty in accelerator developments in a time of high demand.

The beforementioned problem is the reason for the introduction of SCALE-Sim. As Chapter 2 describes, the process of creating the tool required a substantial amount of background work and simulator tool considerations before leading to a developed product.

CHAPTER 2

BACKGROUND WORK AND DESIGN CONSIDERATIONS

The goal of SCALE-Sim is to provide as much valuable information to a computer designer as possible. For this objective to be realized, much background work needed to be done by the entire team that worked on the project to understand the true nature of DNN workloads and how workloads are translated to execute an architecture. Furthermore, as illustrated in Figure 2.1, all computer architectures require the ability to efficiently read and write from memory leading to considerations of memory policies and memory hierarchy during workload execution. These previously mentioned topics and various other technical considerations are vital to the strength of SCALE-Sim and to user experience (*UX*) design. This section will cover the detailed background work performed to create the scalable, modular simulator.

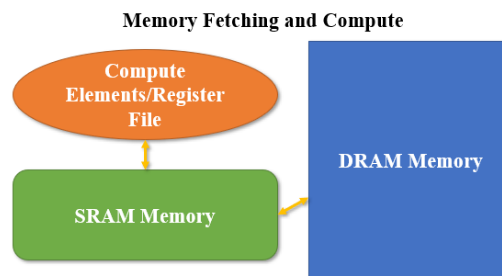


Figure 2.1: High-level architecture for compute and memory stack

2.1 Systolic Array

Systolic array is the architecture of concern for this simulator. A systolic array is a collection of processing engines (*PEs*) with each element connected via a *mesh* topology. Each PE reads data from its neighbors, computes a mathematical function and stores the result in its local memory [20]. In our applications, the PEs are MAC units are tightly

coupled with store and forward units to accommodate data passing. One advantage of systolic arrays lies in the simplicity of design making it relatively easier to build by microprocessor designers over competing compute architectures.

Though modeling a systolic array might seem simple in practice, the highly configurable nature of the array dimensions or whether the model accommodates for output planes that can retrieve output values from middle MAC units instead of waiting for the result to forward to the bottom of the array creates some modeling challenges. Figure 2.2 illustrates a method to read in the input operands from top and left and write out the output operands to the bottom assuming no output planes.

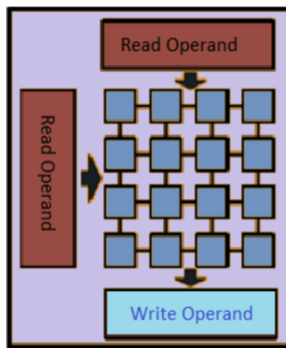


Figure 2.2: Read operand elements pulled from left and top of systolic array and write operand elements pushed out from bottom

2.2 Dataflows

In many systems running state-of-the-art DNNs, energy and performance challenges arise from the large amount of activations and weights required for operations that are fetched from far-away memory banks [3]. These highly expensive fetches lead to orders of magnitude higher latency and energy requirements compared to local fetches, not to mention the added bandwidth challenges associated with greater memory accesses [21, 22]. Fortunately, the translation of input/output operands for a convolutional layer into

input/output operands for a systolic array architecture is programmable, and there exists an optimal mapping for the best energy efficiency, which depends on the shape configuration of the DNN and the constraints on hardware resources such as the number of PEs and the size of memory in the hierarchy [1]. For this reason, dataflow approaches with data-reuse are preferred to give us optimal efficiency. The most widely used subset of these approaches are illustrated in Table 2.1. The five forms that are investigated involve the reuse of elements in a convolution operation. The first approach is labeled as input stationary (*IS*) which maximizes reuse of the input activations or input feature map. The second dataflow is weight stationary (*IS*) reusing the weights in the filter. The third approach is output stationary (*OS*) dataflow which reuses the partial sums that are accumulated to create the *ofmap*.

The fourth dataflow is row stationary which is a novel approach concerned on maximizing reuse on all the above-mentioned components. The premise of the dataflow is to keep the most recently used input and output data in the register file of the ALU maximizing nearby accesses [23]. This approach would essentially lead to maximum reuse from the most local memory location (RF) and would minimize SRAM and DRAM accesses [4]. The disadvantage of this dataflow is that a systolic array architecture cannot support a row by row computation required by a single PE. Finally, the last and most obvious approach would be to choose no local reuse leading to limited energy expended in the preprocessing stage to run workloads. Still, the disadvantages of constantly reading and writing values from higher level memory provides little optimism for this approach in a general convolution layer workload based on earlier discussion.

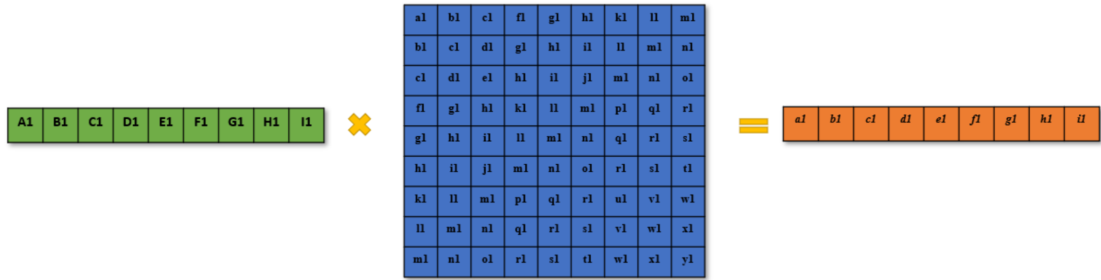
Table 2.1: Reuse characteristics of frequently used DNN accelerator dataflows

Dataflows	Input Stationary	Weight Stationary	Output Stationary	Row Stationary	No Local Reuse
Ifmap Reuse					
Weights Reuse					
Partial Sums Reuse					

2.3 Convolution Translation

Although a convolution operation does not resemble a matrix multiplication operation at first glance, the operands are translated into MAC operations friendly format to be executed on a compute array. The process of convolution translation involves all 3 operands in a convolutional layer: input activations (*ifmap*), weights, and output activations (*ofmap*) as shown in Figure 2.3 into a Toeplitz matrix, a specific matrix in which each descending diagonal from left to right is constant [1].

a. Convolution operands



b. Matrix Multiplication operands (Toeplitz matrix)

Figure 2.3: Read and write operands after convolution translation

As illustrated in Figure 2.3, the translation of the operands in a convolution operation does not preserve the ordering of data elements or the dimensions of operands. In this case, the weight matrix changes from $(3, 3)$ to $(9, 1)$ while the *ifmap* matrix transforms from $(5, 5)$ to $(9, 9)$. At first glance, it might seem that the total elements are being preserved after translation, but this is not the case. The translation does not guarantee unique elements in the resulting matrices thereby creating reuse opportunities.

After translation, the operand matrices are in a format compatible with dot product matrix multiplication: $O = I \cdot W^T$. This can be proven by testing that the number of columns in the *ifmap* operand and the number of rows in the filter operand are equal after translation. Therefore, the *ofmap (rows, columns)* dimensions are calculated as $(9, 1)$.

To add more complexity to translation, each input operand may need to be skewed before reading into a compute array to achieve proper timing in the dataflow. By the nature of the dataflows analyzed in Section 2.2, input skews also result in output skews. Therefore, the final input activation operand looks like the illustration in Figure 2.4. The filter and *ofmap* from Figure 2.3 are not presented since skewing would have no effect on a 1-dimensional matrix.

Figure 2.4: *Ifmap* operand after skewing (rotated 90° right)

The process of skewing does not alter the number of elements in the operand and changes only 1 dimension of the operand. To prove these statements, a comparison is made between Figure 2.3 operands after translation and Figure 2.4 after skewing. Since the process of skewing is no more than a matrix manipulation, software tricks enable us to model this effect.

The two steps of translation and skewing in order is performed for all operands to satisfy the condition of dataflow over a systolic array for the *OS* dataflow; however, other dataflows such as *IS* and *WS* do not skew the *ifmap* and the weight matrix, respectively. This is because the stationary operand is first positioned into the compute array before the non-stationary operand is fed into the MACs in *IS* and *WS* dataflows. It is important to keep in mind that this still results in a skewed output since the *ofmap* is still computed with a skew. The specific nature of the workload translations and how this affects the reads and writes from SRAM is extremely critical for accurate modeling; for more in depth analysis, Section 3.6 describes the specific nature of the dataflows using a read/write trace.

2.4 Memory Policies

Once the process of translating operand matrices to the hardware is finalized, the memory policy used to fetch data elements from off-chip memory into local memory need to be analyzed. Assumption is made that no intermediate memory between off-chip and local memory and that off-chip memory is held in DRAM and local memory is in SRAM.

In an ideal case, the SRAM would be big enough to hold exactly enough elements needed to perform the computation for a DNN layer without needing fetches from off-chip during compute processing. Unfortunately, memory is extremely expensive especially on edge computing (e.g., IoT or mobile) leading to considerations of memory policies at a hardware level [1]. This challenge is one that is at the crux of all modern NUMA systems and will be important to model [24].

Many caching or memory policies exist within modern processors to decrease latency of misses in local memory. Three policies stand out as most important to model considering the nature of each in the context of the memory problem. In each scheme, a double buffer SRAM allocation procedure is used. In other words, the SRAM is separated into an inactive and active buffer based on a ratio defined by the architect that can be defined as active buffer percentage (ABP). The active buffer contains elements that are accessed by compute elements at any point in the execution of the workload while the inactive buffer is used for fetching elements from off-chip DRAM. For simplicity, each buffer is assumed to be the same size meaning the true size of the accessible SRAM by the compute region is 50% of the entire SRAM capacity if $ABP = 0.5$. Next, each memory policy is described in detail.

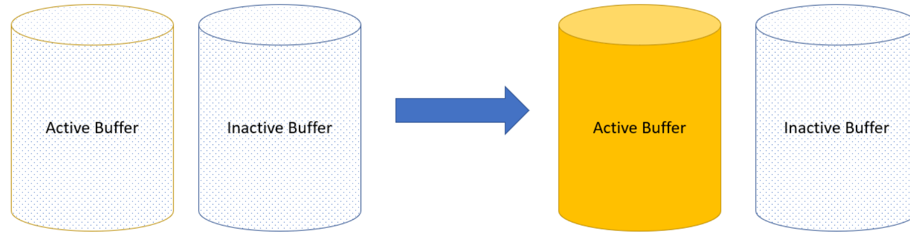


Figure 2.5: Initial Prefetch before process execution

In the demand fetch scheme, the SRAM active buffer is initially filled fully to contain the elements for the first set of accesses from the compute region. This process illustrated in Figure 2.5 is performed to ensure no stalls are needed to start execution of a workload. Once the first miss occurs in the active buffer during an access, the active buffer is flushed out and designated inactive while the current inactive buffer is filled with fetches from off-chip DRAM and designated as the active buffer. This process of flushing and fetching consumes stall cycles in computing due to latency and bandwidth constraints. Unfortunately, only once the inactive buffer is filled and designated as active, compute execution resumes. This process of flushing and fetching is repeated each time a miss occurs in the active buffer. As can be assumed, the stall cycles from fetching can add substantial delays in execution due to the lack of a preemptive fetching mechanism or prefetching. Furthermore, the inactive buffer in this setting is sitting idle for most of execution other than the time in which fetching occurs. One way to make use of the capacity of the SRAM is to reserve use the entire buffer for accesses using a single buffer scheme which is equivalent to using the entire SRAM space as the active buffer ($ABP = 1.0$). The inevitable stall cycles in this method is empirically calculated using the equation below with $C = SRAM\ memory\ capacity$, $BW = off - chip\ storage\ bandwidth$, and $L = latency\ for\ single\ off - chip\ access$.

$$Data\ Elements = C \times ABP\ ;\quad Stall\ Cycles = \left(\frac{Data\ Elements}{BW}\right) + L$$

Due to the severe limitations of the demand fetch scheme especially when the workload memory accesses requirement is much larger than the size of the local SRAM, a prefetch scheme is analyzed in which DRAM bandwidth is predefined [5]. In this scheme, the active buffer is prefilled to the maximum capacity before execution begins according to the predefined bandwidth requirements. During the accesses of the active buffer by the compute region, the inactive buffer prefetches elements from DRAM to serve future requests. Once the first miss occurs in the active buffer, the active buffer is flushed out while the inactive buffer is treated as the new active buffer. In an ideal case, the active buffer is instantly refilled with new elements to service the SRAM miss while the inactive buffer is emptied for future prefetches. Therefore, this process prevents stall cycles leading from SRAM misses. Unfortunately, prefetching is not always perfect since the active buffer flush and refill only occurs once the inactive buffer is filled fully by elements from DRAM. The condition exists in which an active buffer miss occurs before the inactive buffer is fully filled. In this case, stall cycles are incurred until the inactive buffer is finished filling. Only once filling is complete, the inactive buffer is treated as the new active buffer and execution resumes. This edge condition is the primary reason for stall cycles in this scheme.

Since the stall cycles in the previous prefetching scheme are attributed to the limited predefined bandwidth of a SRAM to DRAM link, a new scheme is considered that does not define a bandwidth. In this scheme, the logic of prefetching and flushing is maintained to achieve the same functionality of the bandwidth-fixed prefetch scheme; however, the bandwidth is shrunken or enlarged each cycle to accommodate the exact fetching requirement over the SRAM to DRAM link. This logic would always ensure a perfectly

stall-free process. As evident from the description of shrinking and enlarging bandwidth, a bandwidth-variable scheme is not practical in an actual architecture; instead, the bandwidth of a link is a design-time metric. The primary advantage of this scheme would be simulator analysis of bandwidth requirements before designing and taping-out the architecture.

In the above memory policies, the inactive and active buffer are assumed to be equal in size; however, this should not be a fixed parameter since buffer allocation strategies make big differences in the execution of a workload. Instead the *ABP* parameter should be available to an architect in the range of $[0.5, 1.0]$ to achieve peak performance. If the inactive and active buffer sizes are not even as shown in Figure 2.6, the flushing process would only result in a partially flushed active buffer to replenish new elements from the inactive buffer. In this example, active buffer is 75% of the total buffer meaning only 50% of the active buffer is flushed every time a flush is performed. Interesting results are possible with variable buffer ratios which is important to model.

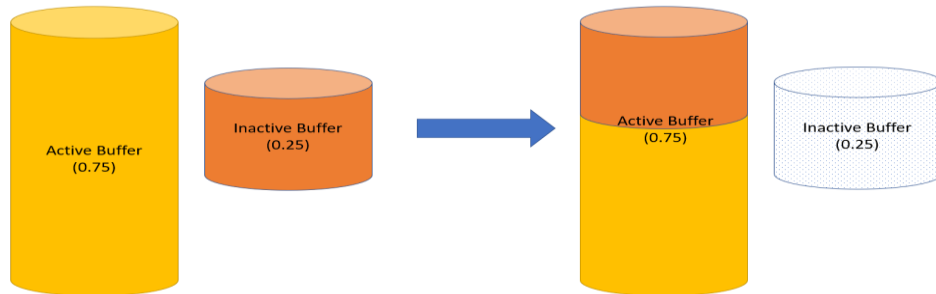


Figure 2.6: Partial flush and replenish of active buffer

2.5 DNN Layers

DNNs are built using a series of input, hidden, and output layers. A convolutional layer involves input activations and weights which are read and output activations which are

written by the convolution. Convolutional layers (CONV) are often interleaved by pooling (POOL) and other activation layer such as rectifier logic unit (RELU) [3]. These operations are interesting at a deep learning level since the layers change the spatial characteristics or the element values of the output activations which affects the subsequent layers; however, the memory demand for each of these layers is zero and the compute demand is not very high relative to CONV and FC layers. For this reason, activation layers are not vital to model for performance measures. It should be noted that activation layers execution is often handled by a post processing unit (PPU) in a DL accelerator architectures which takes the SRAM output activation buffer elements as inputs and outputs elements back into the input SRAM buffers reserved for input activation elements and weights [3].

Finally, a DNN uses a fully connected layer (FC) to compute the likelihood of each inference label. Modeling a FC layer in a compute array is virtually the same as a CONV layer since each layer involves the same inputs and outputs. The most significant difference is the size of the input and output elements since the output activations of a FC layer must dimensionally match the number of labels, a requirement which does apply for a CONV layer.

Focusing on CONV layers, the convolution process is standard for an *ifmap* with a single channel and a filter with a single channel since the *ofmap* will always be a single channel as well. The variances occur when the number of channels is > 1. In this situation, 2 types of convolution are be performed. Depth-wise convolution is a technique in which each channel is broken into separate elements. Then, the convolution is performed independently for each sub-element (*ifmap* and filter) pair. Once the independent *ofmap* are computed for each sub-element pair, the *ofmaps* are concatenated to form the final

ofmap which ensures the number of output channels is equal to the number of input channels. The issue with depth wise convolution is the restriction against multiple filters in the CONV layer. This problem is not obvious at first glance; however, it can be rationalized by the fact that multiple filters would lead to concatenation the *ofmap* results, thereby repeating the concatenation process for two separate logical purposes. To solve this problem, pointwise convolution is used. In this convolution operation, the number of channels in the *ifmap* and filter are not preserved meaning the *ofmap* is always a single channel map in the result of execution of a single *ifmap* and single filter convolution operation. Pointwise convolution can be thought of as a convolution across the entire depth of the *ifmap*. Since concatenation is not required in the *ofmap* of a resulting convolution with only a single filter, concatenation of the result from convolution of each of N_f filters is performed without loss of generalization. Because of this advantage, point-wise convolution is more prevalent in practice.

Another form of convolution is depth-wise separable convolution ($DWSC$). This method involves a $2-stage$ process in which a depth-wise convolution is first performed followed by a pointwise convolution. Using this method, depth-wise convolution logic is mixed with the advantage of allowing multiple filters. Furthermore, the total computational complexity of this method is lower than the complexity of performing a pointwise convolution directly [25]. To prove this claim, let us assume 512 (N_f) kernels of dimensions ($F_h = 5, F_w = 5, F_d = 3$), *ifmap* of dimensions ($I_h = 12, I_w = 12,\ I_d = 3$), resulting in *ofmap* of height and width dimensions ($O_h = 8, O_w = 8$). Total multiplication operations (M_{pw}) for a pointwise convolution given these convolution parameters:

$$M_{pw} = N_f F_h F_w F_d O_h O_w = 512 * 5 * 5 * 3 * 8 * 8 = 5{,}529{,}600$$

If the same convolution parameters are used for $DWSC$, total multiplications (M_{dw}):

$$M_{dw} = F_h F_w F_d O_h O_w + N_f F_d O_h O_w = 5 * 5 * 3 * 8 * 8 + 512 * 3 * 8 * 8 = 103{,}104$$

As evident from the calculations in the example above, the number of operations in convolution drops significantly for DWSC compared to pointwise convolution.

2.6 Partitioning

In previous discussions, a single systolic array of MAC unit is assumed to be used for compute by the entire workload. Therefore, the dimension of the array constrains the number of MMs at any given time by a workload. For example, if the compute array had dimensions of $(8, 8)$, the maximum matrix multiplications in one cycle is 64. Using a tiling or partitioning scheme, the workload is split into partitions each independent in computation. Therefore, each partition is executed in parallel assuming enough compute arrays to support compute requirements. One major benefit of a partitioned scheme is to achieve better utilization of compute elements. For example, in the case in which the per filter elements are much greater than the number of filters ($[F_w * F_h * F_d] \gg N_f$), the filter operand is dimensionally biased after translation. This leads to low utilization of a square shaped compute array for sequential execution. By partitioning the workload over independent compute arrays that match operand dimensions, utilization improves thereby improving energy and execution time [21]. The idea of partitions can be translated to hardware systems as illustrated in Figure 2.7. Each node in the graphic consists of a systolic

array with SRAM buffer and a shared link to DRAM executing independent partitions.

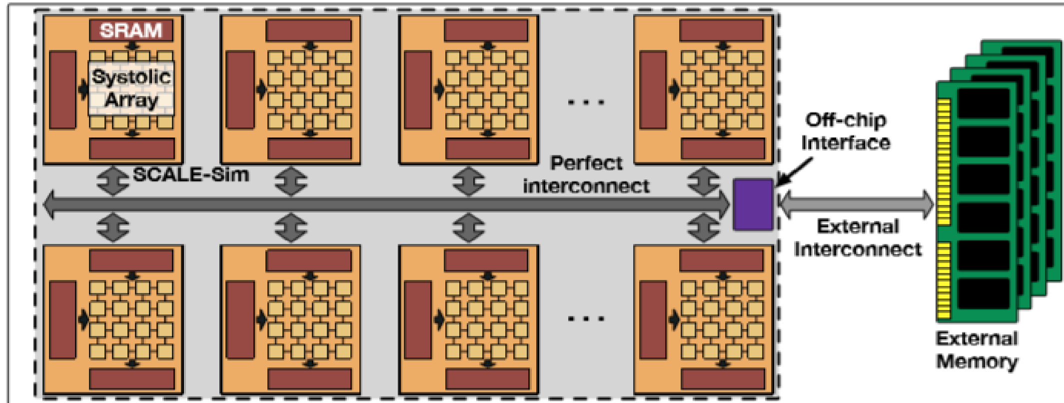


Figure 2.7: Network of systolic arrays each with a perfect interconnect to external memory (DRAM) [21]

Partitioning the workload can be performed using three approaches: input-parallel, filter-parallel, and input-on-filter-parallel. As the names suggest, partitioning can be attributed to the operands partitioned against. In the input-parallel scheme, the *ifmap* operand is divided in a uniform manner for up to the number of partitions requested while keeping the filter static. The reason the requested partitions may not be possible is because keeping the filter static requires specific partitioning to ensure convolution correctness. In the filter-parallel scheme, the filter operand is divided in a uniform manner while keeping the *ifmap* static also leading to less than the requested partitions since keeping the *ifmap* static requires a specific distribution of filter. To counter-act the less than ideal nature of the previous two schemes, input-on-filter parallel scheme can be used to flexibly partition against the *ifmap* and filter. This leads to greater partitioning granularity and evenly partitioned workloads. It should be kept in mind that the *ofmap* operand is dependent on the partitioning of the input operands resulting in *ofmap* partitioning for each of the schemes listed above.

2.7 NoC Considerations

In the partitioned scheme described above, independent nodes are responsible for execution of a single partition in a larger workload. In this configuration, each node is a separate compute element in a larger network. The main disadvantage of this design is the assumption that each PE has a perfect interconnect with external memory leading to ideal bandwidth and latency. In larger networks, each node cannot satisfy the requirement of a perfect interconnect leading to enormous differences in bandwidth, latency, and energy between on-chip PE communication [3]. Therefore, another approach is examined in which L2 SRAM buffers are allocated within the network to be used for data sharing. This design gives rise to the idea of Network-on-Chip (NoC) communication systems, widely used in Chip Multi-Processors (CMP). The primary advantage of a NoC is the ability to scale to support large-scale inference. This is due to the energy efficiency and latency improvement for on-chip accesses to L1 or local or remote L2 SRAM compared to off-chip accesses to DRAM [22]. This added performance in latency, energy, and throughput measures makes NoC architectures the de facto fabric for application specific SoCs [26].

Many DL platforms today are built by interconnecting multiple accelerators together such as Google's TPU that uses multiple TPUs interconnected in a 3D Torus [27]. For this reason, it is important that SCALE-Sim can be wrapped around framework that supports NoC architecture modeling. Specifically, the framework needs to support NoC parameters such as topology, link bandwidth, L2 SRAM memory mapping, and additional parameters discussed below.

Supporting versatile NoC topologies would include *mesh, torus, ring, fully connected* graph or any other direct topology as an input parameter with variable number of nodes in

the NoC [28]. Additionally, indirect topologies such as memory-centric networks should be supported [29]. To model the constrained nature of data sharing that occurs within a NoC, the modeling framework should have an idea of link and/or port bandwidth which can model real-time congestion and bandwidth requirements anywhere in the NoC. Finally, the allocation of data elements within the NoC should be modeled leading to interesting considerations of mapping. For example, every node in the NoC could contain a L2 SRAM bank, or instead a specific number could represent the data sources. Moreover, each L2 SRAM bank could contain a non-uniform amount of data elements specified by a memory map.

Chapter 3 presents the final design choices and implementation work performed to develop SCALE-Sim and the NoC modeling framework as an extension to the base simulator.

CHAPTER 3

SCALE-SIM-V2: CNN ACCELERATOR SIMULATOR

3.1 SCALE-Sim

The background research and consideration effort performed in Chapter 2 gave rise to the development of SCALE-Sim, the configurable systolic array-based cycle accurate CNN accelerator simulator [2]. The tool was developed as illustrated in Figure 3.1 to perform DNN inference on systolic arrays and to generate on-chip memory access, runtime, and DRAM bandwidth requirements for a given workload [2]. The tool performed fundamental operand matrix creation, followed by SRAM trace generation, and ending with a DRAM memory policy that calculates the bandwidth requirement. The key logic contributions of the effort in SCALE-Sim-v2 can be broken down into three phases illustrated in Figure 3.2. Support for depth-wise convolution, batching, and MNK operands was added during operand matrix creation. Further key additions include support for trace generation without an output plane on the systolic array, and separate traces for all three operands during SRAM trace generation. Lastly, contributions to DRAM trace generation include support for a fixed bandwidth prefetch memory policy, support for a NoC model memory policy, and the ability to generate multiple traces for each memory source in a NoC. Encompassing the entire simulator framework for SCALE-Sim, much of the end-to-end logic was not organized in a format consistent with efficient debugging and extended development. For this reason, all current logic was revamped and modularized before the additions above were incorporated to allow for continued development efforts from the design community. This chapter provides specific insights on the overhauled framework and the last section

of this chapter focuses on experiments conducted using SCALE-Sim-v2 and possible novel experiments for future studies.

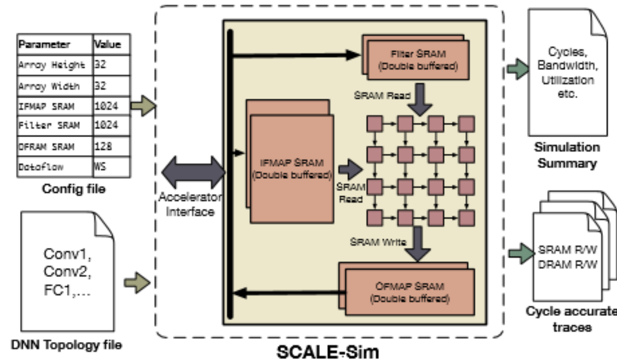


Figure 3.1: Schematic depicting the inputs needed and the outputs generated by SCALE-Sim [2].

3.2 Simulator Environment

The simulator logic is written in Python3 for the rich suite libraries for simulator design, quicker time to development, and faster time to usage for users. The primary python dependencies used are numpy, configparser, math for simulator logic and tqdm for debugging. The top-level directory is SCALE-Sim with subdirectories: configs, scale_sim_simulator, topologies.

3.3 Software Organization

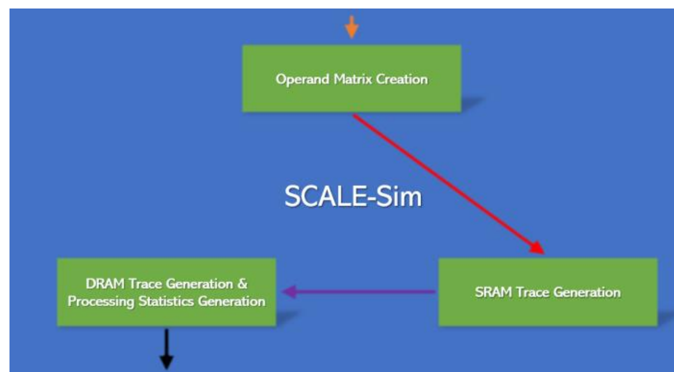


Figure 3.2: High level code organization and flow in SCALE-Sim simulator environment. Arrows provide a rough view of the sequential flow of the tool and do not necessarily represent I/O movement

SCALE-Sim is broken down into 3 main modules illustrated in Figure 3.2. 1st module is Operand Matrix Creation which handles the conversion of convolutional layer operands into operands to be mapped onto a compute array of MAC units. 2nd module is SRAM Trace Generation which creates a trace representing the reads and writes from SRAM for the specific dataflow selected. 3rd module is DRAM Trace Generation which creates a trace representing the reads and writes from DRAM as well as the cycle accurate adjustment of the SRAM trace. The 3rd module also handles statistics such as average SRAM and DRAM bandwidth as well as more detailed analysis. The top-level file for the simulator is scale.py. This file takes in the inputs listed in Section 3.4 and parses these values into parameters of logical consequence.

3.4 SCALE-Sim Inputs

The architecture parameters are presented in a .cfg file included in the configs directory. The 10 architecture parameters are listed under the section: architecture_presets. The 10 parameters are shown in Table 3.1.

Table 3.1: Example architecture parameters in configuration file

Array Height	**Array Width**	**Ifmap SRAM Size**	**Filter SRAM Size**	**Ofmap SRAM Size**	**Ifmap Offset**	**Filter Offset**	**Ofmap Offset**	**Bandwidth**	**Dataflow**
32	*32*	*524,288*	*524,288*	*524,288*	*0*	*10000000*	*20000000*	*1,000*	*OS*

Array height (A_H) and array width (A_W) represent the row and column dimensions of the systolic array PEs. The *ifmap*, filter and *ofmap* SRAM size represent the available on-chip memory in bytes for each operand. The offset for each operand represents the first address in the address space for each operand. The bandwidth value is the available bandwidth per cycle in a unidirectional link between the on-chip SRAM and off-chip DRAM. Finally, the dataflow represents the mapping of elements onto the compute array.

Convolutional and/or fully connected layers are presented in a csv file included in the topology directory. Each file has a header line with the data labels followed by lines representing each layer and its characteristics. For any CONV or FC layer, 7 parameters are needed to describe the inputs and convolution process: *ifmap* height and width, filter height and width, depth of *ifmap* and filter given as channels, number of filters, and the stride of convolution. An example of a csv file representing AlexNet CNN, most popular for its accuracy in image prediction during the ImageNet Challenge in 2012 leading to the revolution of CNN adoption, is presented in Table 3.2. AlexNet has 8 layers: 5 convolution layers and 3 fully connected layers; only the 5 CONV layers are shown here with the layer characteristics [30].

Table 3.2: AlexNet CNN with 5 CONV layers

Layer Name	**IFMAP Height**	**IFMAP Width**	**Filter Height**	**Filter Width**	**Channels**	**Number of Filters**	**Strides**
Conv1	224	224	11	11	3	96	4
Conv2	207	207	5	5	96	256	1
Conv3	13	13	3	3	256	384	1
Conv4	13	13	3	3	384	384	1

Conv5	13	13	3	3	384	256	1

3.5 Operand Matrix Creation

Operand matrix creation is the process of translating convolutional layer operands into compute array operands. This process is described in Section 2.5 for *ifmap*, filter, and *ofmap* operands.

3.5.1 Calculated Hyperparameters

Because of the deterministic nature of the *ofmap* operand, the dimensions are calculated using the formula listed below:

$$O_h = ceil[(I_h - F_h + S_h)/S_h];\ O_w = ceil[(I_w - F_w + S_w)/S_w]$$

O_h and O_w represent the *ofmap* height and width, respectively. I_h and I_w represent the *ifmap* height and width, respectively. F_h and F_w represent the filter height and width, respectively. S_h and S_w represent the stride height and width, respectively.

Oftentimes, it can be assumed that the height and width of the layer parameters are equal. Therefore, the *ofmap* height and width will also be equal. For example, Conv1 in AlexNet satisfies this condition; therefore, O_h and O_w are each 55. The reason for using the ceiling function is because non-integer real number dimensions are not allowed and because taking the result of the floor function would result in loss of information in convolutions involving stride values > 1. The ceiling function is necessary to ensure a symmetry in convolution by adding padding in the *ifmap* operand. An example of this

scenario is presented in Figure 3.3. The red portion of the operand represents the padding required to ensure a symmetric convolution given $stride = 2$.

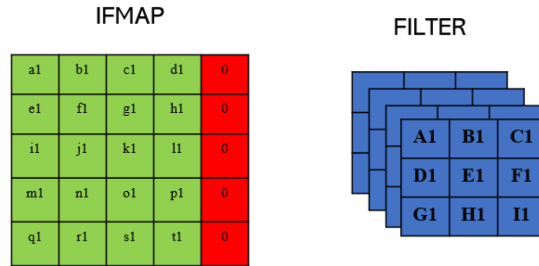


Figure 3.3: $(ifmap_{height} = 4, ifmap_{width} = 4)$ & $(filter_{height} = 3, filter_{width} = 3, filter_{num} = 4)$ with $stride = 2$

3.5.2 Address Mapping

During the process of operand matrix creation, mapping of addresses to the operand dimensions is important to consider. There are 4 logical mapping algorithms that can be deployed in operand matrix creation (each placed in order of sequential priority): [*Channel, Height, Width*], [*Height, Channel, Width*], [*Channel, Width, Height*], [*Width, Channel, Height*]. After considering the various the most widely used priority structures in practice, [*Channel, Height, Width*] is the preferred solution [1, 3]. An example of an *ifmap* implemented with the address mapping scheme is shown in Figure 3.4.

Address Mapping (Channel, Width, Height)

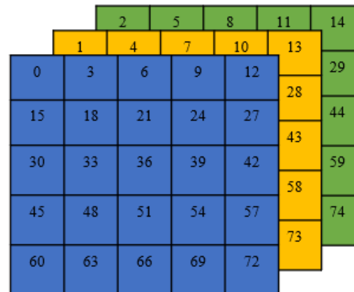


Figure 3.4: An example activation map with addresses prioritized based on the [*Channel, Height, Width*] scheme

3.5.3 Batching

In practical applications, a technique called batching is deployed to continuously execute multiple *ifmap* convolutions over a filter [1]. In order to set up the batch for convolution, the “batch” of *ifmaps* is first translated independently into *ifmap* operand matrices and row concatenated. This process ensures the batch is executed in the same dataflow execution resulting in a single *ofmap* operand matrix. The matrix is then translated back into individual *ofmap* by separating the “batch” of *ofmaps* based on the calculated dimensions of the final output.

3.5.4 MNK Operands

Operand matrix creation is not performed if DNN operands are given in an MNK format as illustrated in Figure 3.5 since the operands are already GEMM compatible. This consistency ensures all types of DNN layers involving matrix multiplications can be used as input topologies to SCALE-Sim including the layers in a multilayer perceptron (MLP), the recursive layers in a long short-term memory network (LSTM), or any other generic DNN with input, hidden, and output layers.

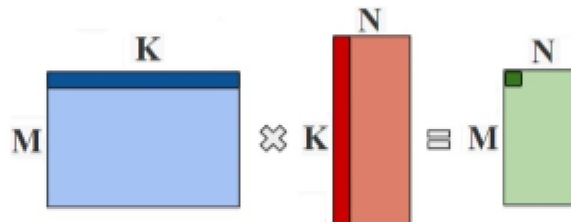


Figure 3.5: Operands compatible with GEMM operations [6]

3.6 SRAM Trace Generation

In the best case, the SRAM trace represents a cycle accurate trace in which there are no stall cycles in computation. In a general case, the SRAM trace is a non-cycle accurate representation of the dataflow in/out of the compute array. The process of generating the SRAM trace requires knowledge of the dataflow used to map elements to MAC units. Out of the 5 dataflows discussed in the background Section 2.2, 4 dataflows are achievable using a systolic array architecture while only 3 are worth considering due to their reuse capabilities eliminating no local reuse dataflow. The 3 dataflows are output stationary (*OS*), weight stationary (*WS*), and input stationary (*IS*), one of which is provided by the user in the input config file. The top-level file for this module is trace_per_layer.py which takes the operands at the output of operand matrix creation, the dataflow, and compute array dimensions as inputs. The logic within this file chooses which trace generation file to call based on the dataflow: os_trace_per_fold.py, ws_trace_per_fold.py, or is_trace_per_fold.py each of which handles its dataflow-specific trace logic. Each dataflow, first, separates the operand matrices into folds. This process is crucial if the operands are larger than the compute array dimensions since entire operands cannot be executed in one process. Then, each fold is skewed and executed on the systolic array independently over parallel compute arrays or sequentially over one. In the analysis and examples provided, a single compute array is assumed for workload execution eliminating workload parallelization possibilities.

Before each dataflow trace is analyzed in depth, a few terms need to be clarified: $ifmap_{rows}$ and $ifmap_{cols}$ are the input feature map operand dimensions, $filter_{rows}$ and $filter_{cols}$ are the filter operand dimensions, and $ofmap_{rows}$ and $ofmap_{cols}$ are the output feature map operand dimensions. By definition, $ifmap_{cols} = filter_{rows}$, $ofmap_{rows} = ifmap_{rows}$, and $ofmap_{cols} = filter_{cols}$. Additionally, *folding* is the idea that the full execution of a network may not be possible in one iteration on a compute array as can be demonstrated with a layer that requires a $64 * 45$ physical PE array; however, the compute used contains a $32 * 32$ PE array. In this case, the execution of the layer can be broken up into $N_{folds} = 2$ of $32 * 32$ and $32 * 13$ [1].

For the examples illustrated in Figures 3.6, 3.7, and 3.8, the network topology shown in Table 3.3 is executed on a $(A_H = 4, A_W = 4)$ compute array.

Table 3.3: Example network topology

Layer Name	**IFMAP Height**	**IFMAP Width**	**Filter Height**	**Filter Width**	**Channels**	**Number of Filters**	**Strides**
BASE1	5	5	3	3	1	4	1

3.6.1 Output Stationary Trace

In the output stationary dataflow, $N_{folds} = Ceil\left(\frac{ifmap_{rows}}{SA_{rows}}\right) * Ceil(\frac{filter_{cols}}{SA_{cols}})$. The first term ($v_folds$) represents the folds in the *ifmap* and the second term (v_folds) represents the number of folds in the filter. The dataflow is processed by iterating over v_folds and h_folds in no specific priority. For each iteration, the fold specific operands are extracted from the larger operands by indexing the $ifmap_{rows}$, $filter_{cols}$, $ofmap_{rows}$, $ofmap_{cols}$

for each fold by $v_{folds}, h_{folds}, v_{folds}$, and h_{folds}, respectively. The $ifmap_{cols}$, $filter_{rows}$ retain the same characteristics as their larger operand counterparts regardless of fold. The fold specific operands *ifmap*, filter, *ofmap*, are skewed independently before being pushed into the compute array. The skew appears from left to right in the trace as illustrated in Figure 3.6.

A notable trace characteristic of the *OS* dataflow is that writing of the sums to the *ofmap* does not occur until at least the cycle after all reads are made into the systolic array. This is due to the delay in forwarding the sums down towards the bottom MAC unit to escape the compute array since this configuration does not assume an output plane. Therefore, even if the sums at the top row of the compute array are calculated by cycle α, the sum can only be forwarded down to the bottom of the compute array in β cycles at the minimum, a term that represents the total time to forward from top to bottom of the compute array. In the best case, forwarding would happen instantly after calculated the sum resulting in writing to SRAM in the $\alpha + \beta$ cycle assuming no stalls; however, this case assumes no sums are calculated lower down the compute array. In the worst case, sums are calculated lower down the compute array and would take forwarding precedent over the top row sums meaning writing to SRAM would occur in the $\alpha + 2\beta - 1$ cycle assuming no stalls. As evident from the worst-case scenario, the taller the compute array, the worse the delay is for writing. Fortunately, this delay is a direct result of the minimal or zero delay in the lower sums meaning the average delay for each sum converges to $\alpha + \beta$ cycles for a general sum anywhere in the compute array.

In total, 3 folds are needed to complete execution using this dataflow using the N_{folds} equation listed above.

IFMAP				FILTER				OFMAP			
a1	NULL	NULL	NULL	A1	NULL	NULL	NULL	NULL	NULL	NULL	NULL
b1	b1	NULL	NULL	B1	A2	NULL	NULL	NULL	NULL	NULL	NULL
c1	c1	c1	NULL	C1	B2	A3	NULL	NULL	NULL	NULL	NULL
f1	d1	d1	f1	D1	C2	B3	A4	NULL	NULL	NULL	NULL
g1	g1	e1	g1	E1	D2	C3	B4	NULL	NULL	NULL	NULL
h1	h1	h1	h1	F1	E2	D3	C4	NULL	NULL	NULL	NULL
k1	i1	i1	k1	G1	F2	E3	D4	NULL	NULL	NULL	NULL
l1	l1	j1	l1	H1	G2	F3	E4	NULL	NULL	NULL	NULL
m1	m1	m1	m1	I1	H2	G3	F4	NULL	NULL	NULL	NULL
NULL	n1	n1	p1	NULL	I2	H3	G4	NULL	NULL	NULL	NULL
NULL	NULL	o1	q1	NULL	NULL	I3	H4	NULL	NULL	NULL	NULL
NULL	NULL	NULL	r1	NULL	NULL	NULL	I4	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	*d1*	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	*c1*	*d2*	NULL	NULL
⋮				⋮				⋮			
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	*a3*	*b4*
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	*a4*

Figure 3.6: 1-fold SRAM trace for output stationary (*OS*) dataflow. Header represents the division in elements from each operand. By convention, *ifmap* and filter elements are read from SRAM and *ofmap* elements are written to SRAM. Reference Figure 3.7 and 3.8 for comparison.

Adding an output plane could be experimented with leading to an offset of the *ofmap* portion of the trace in Figure 3.6 by $-\beta$ rows. An output plane could also allow a flip on input activation and filter values so that address a is written earlier than b, address b is written earlier than c and so on.

3.6.2 Weight Stationary Trace

In the weight stationary dataflow, $N_{folds} = Ceil\left(\frac{filter_{rows}}{SA_{rows}}\right) * Ceil(\frac{filter_{cols}}{SA_{cols}})$. The first term is denoted as v_{folds} and the second term as h_{folds}. Specific to the *WS* dataflow, folds are dependent on both dimensions of the filter operand. For each fold, the fold specific operands are extracted from the larger operands by indexing the $ifmap_{cols}$, $ofmap_{cols}$, $filter_{rows}$, and $filter_{cols}$ for each fold by v_{folds} , h_{folds}, v_{folds}, and h_{folds}, respectively. $ifmap_{rows}$, $ofmap_{rows}$ retain the same characteristics regardless of fold. In this dataflow,

only the *ifmap* and *ofmap* operands are skewed since the filter is pushed into the compute array to start each fold. The specific nature of this dataflow is illustrated in Figure 3.7.

A notable observation from the *WS* dataflow in the example below is that the first output element is pushed out 4 cycles after the cycle in which the first *ifmap* element is read in. This is contrary to the behavior in *OS* dataflow since output elements are pushed out before all input elements are read in in the *WS* dataflow even without an output plane. This is due to the nature of the dataflow in which the MAC units residing in the higher rows forward down partial sums for accumulation with the output at the succeeding MAC units until the bottom of the compute array is reached leading to the first output element calculated in theA_H cycle, at a minimum. Note that the output elements do not necessarily constitute the final *ofmap* elements since the entire filter needs to be read into the compute array for the final sum of the partial sums to be calculated. This is evident from Figure 3.7 since elements $[\mathrm{E}, \dots, \mathrm{I}]$ are missing from the filter trace.

In total, 3 folds are needed to complete execution using this dataflow using the N_{folds} equation listed above.

IFMAP				FILTER				OFMAP			
NULL	NULL	NULL	NULL	D1	D2	D3	D4	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	C1	C2	C3	C4	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	B1	B2	B3	B4	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	A1	A2	A3	A4	NULL	NULL	NULL	NULL
a1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
b1	b1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
c1	c1	c1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
f1	d1	d1	f1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
g1	g1	e1	g1	NULL	NULL	NULL	NULL	*a1*	NULL	NULL	NULL
h1	h1	h1	h1	NULL	NULL	NULL	NULL	*b1*	*a2*	NULL	NULL
k1	i1	i1	k1	NULL	NULL	NULL	NULL	*c1*	*b2*	*a3*	NULL
l1	l1	j1	l1	NULL	NULL	NULL	NULL	*d1*	*c2*	*b3*	*a4*
m1	m1	m1	m1	NULL	NULL	NULL	NULL	*e1*	*d2*	*c3*	*b4*
NULL	n1	n1	p1	NULL	NULL	NULL	NULL	*f1*	*e2*	*d3*	*c4*
	⋮				⋮					⋮	
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	*i3*	*h4*
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	*i4*

Figure 3.7: 1-fold SRAM trace for weight stationary (*WS*) dataflow. Reference Figure 3.6 and 3.8 for comparison.

3.6.3 Input Stationary Trace

In the input stationary dataflow, $N_{folds} = Ceil\left(\frac{ifmap_{cols}}{SA_{rows}}\right) * Ceil(\frac{ifmap_{rows}}{SA_{cols}})$, the first term is designated as v_{folds} and the second term as h_{folds}. In contrast to the *OS* dataflow, folds are dependent on the two dimensions of the *ifmap* operand in the *IS* dataflow. For each fold, the fold specific operands are extracted from the larger operands by indexing $ifmap_{rows}$, $ifmap_{cols}$, $ofmap_{rows}$, $filter_{rows}$ for each fold by v_{folds}, h_{folds}, v_{folds}, and h_{folds}, respectively. Only $ofmap_{cols}$ and $filter_{cols}$ retains the same characteristics regardless of fold. In this dataflow, only the filter and *ofmap* operands are skewed since the *ifmap* is pushed into the compute array to start each fold. The specific nature of this dataflow is illustrated in Figure 3.8.

IFMAP				FILTER				OFMAP			
f1	g1	h1	k1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
c1	d1	e1	h1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
b1	c1	d1	g1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
a1	b1	c1	f1	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	A1	NULL	NULL	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	A2	B1	NULL	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	A3	B2	C1	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	A4	B3	C2	D1	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	B4	C3	D2	*a1*	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	C4	D3	*b1*	*a2*	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	D4	*c1*	*b2*	*a3*	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	*d1*	*c2*	*b3*	*a4*
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	*d2*	*c3*	*b4*
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	*d3*	*c4*
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	*d4*

Figure 3.8: 1-fold SRAM trace for input stationary (IS) dataflow. Reference Figure 3.6 and Figure 3.7 for comparison.

Interestingly, the fold shown above in Figure 3.8 takes a minimum of 15 $cycles$ to execute while the folds shown in Figure 3.6 and Figure 3.7 take a minimum of 19 $cycles$ and 20 $cycles$, respectively. This result could lead architects to prefer *IS* dataflow over the field; however, a more complete discussion is required to converge on a stronger result.

3.6.4 Dataflow Comparisons

Though each fold is executed based on the same network layer, the difference in dataflow is what leads to this difference. As is calculated using the N_{folds} equation above, the *IS* dataflow leads to 9 $folds$ much greater than the 3 $folds$ for each OS and WS dataflows. Using a 1st order (semi-accurate) comparison tool to compare the minimum runtime for each of the 3 dataflows, it is obvious that *IS* dataflow ($135 = 9 * 15$ cycles) leads to by far the worst performance followed by WS ($57 = 3 * 19$ cycles) and OS dataflow ($60 = 3 * 20$ cycles).

Unfortunately, greater depth comparisons are still needed to gain a true idea of the effectiveness of each dataflow for a workload. Even though an analysis into total runtime might lead to preference for an *OS* dataflow over the field, implementing a stall free *OS* dataflow architecture is difficult due to the larger SRAM buffer requirements. On the contrary, *WS* and *IS* dataflows require half the amount of SRAM buffer for square operand arrays [2]. Therefore, it is vital to model the complete system with considerations to higher level memory accesses and the latency and bandwidth requirements associated with these accesses to effectively compare various dataflows. This serves as the motivation for DRAM and NoC modeling described in Section 3.7 and Section 3.8, respectively.

3.7 DRAM Trace Generation

As detailed in Section 2.4, the data required to process a convolution layer in real-time is too high to store in local memory. Even with data reuse techniques, the ability to prefetch data into on-chip SRAM buffers before execution eliminates the additional latency due to off-chip accesses to DRAM. A cycle-accurate trace of prefetch requests for reads and writes to DRAM is created to analyze the data movement required for a layer execution based on the memory policy, fixed bandwidth prefetch or variable bandwidth prefetch.

In each memory policy, user parameters for active buffer percentage (*ABP*), DRAM bandwidth, DRAM access round-trip latency, bytes per data element can be modified depending on user preference leading to variable DRAM trace characteristics.

3.8 NoC Model

The base version of SCALE-Sim models a single PE with memory fetching from on-chip SRAM and off-chip DRAM. In the Section 2.6, partitioning is introduced as a mechanism to achieve parallel execution of workloads by multiple PEs. In Section 2.7, the popularity of NoC architectures is presented as a reason to model NoC compute and memory. The two concepts are integrated to give rise to a NoC modeling framework with partitioned execution across various PEs in the NoC. This framework is implemented on top of the SCALE-Sim infrastructure and has the advantage of providing the flexibility to model a suite of NoC architectures depending on the configuration the architect is interested in.

3.8.1 NoC Modeling Framework

The framework to model NoC architectures was developed to handle data requests from PEs in the NoC. The framework has 4 primary parameters that are specified as inputs by the user: NoC topology, remote bandwidth, number of partitions, active buffer percentage.

The NoC topology is the number of nodes in a network as well as the intra-node links between the nodes. The user creates an adjacency matrix using the convention illustrated in Table 3.4 to create a $4-ary\ 2-cube\ mesh$ shown in Figure 3.9 and store the values in a .csv format. In this convention, the left-most column and the top-most row indicate the nodes in the NoC. A value of 1 in the table defines a link between the two nodes and a value of $-$ is equivalent to no link between the two nodes. It should be noted that the bottom left elements and the top right elements are identical in the adjacency matrix. Therefore, a value in the adjacency matrix corresponding to the pair of nodes $(a,\ b)$ will always be the same as the value for the pair $(b,\ a)$. The file path is then given as an input to the simulator

before execution. The simulator parses the matrix at run-time and determines the number of hops between any pair of nodes to calculate the latency of accesses for each source, destination pair communication. Because of the flexible nature of network creation using the adjacency matrix format, any and every irregular and symmetric network can be created and be supported. 3 topologies files were created for use in experiments representing *mesh*, *flattened butterfly*, and *fully connected* topologies [31].

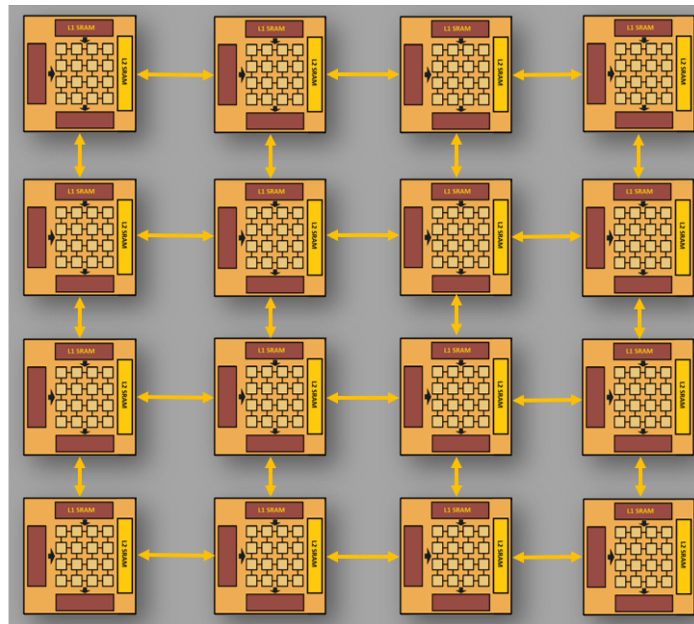


Figure 3.9: Direct *mesh* NoC topology with 16 PEs each acting as a router in the network.

Remote bandwidth is specified as an input parameter by the user and is used to determine the maximum allowed packets to be requested from remote L2 caches each cycle. The remote bandwidth specified acts as the total bandwidth accepted each cycle by the remote port of a node on the network. This is independent from the local bandwidth which is reserved for accesses from/to local L2 cache. Assumptions are made in this model including no off-chip DRAM accesses, an ideal data sharing network with no contention, no real-time NoC issues such as deadlocks and flit blocking. Furthermore, an assumption

is made that bandwidth is not specific to links but rather attached to the remote and local port of the node. So, in effect, infinite packets can traverse the NoC each cycle but only the specified remote bandwidth can enter/exit the node from the remote port illustrated in Figure 3.10.

Table 3.4: *MESH* Adjacency Matrix

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	1	1	-	-	1	-	-	-	-	-	-	-	-	-	-	-
1	1	1	1	-	-	1	-	-	-	-	-	-	-	-	-	-
2	-	1	1	1	-	-	1	-	-	-	-	-	-	-	-	-
3	-	-	1	1		-	-	1	-	-	-	-	-	-	-	-
4	1	-	-	-	1	1	-	-	1	-	-	-	-	-	-	-
5	-	1	-	-	1	1	1	-	-	1	-	-	-	-	-	-
6	-	-	1	-	-	1	1	1	-	-	1	-	-	-	-	-
7	-	-	-	1	-	-	1	1	-	-	-	1	-	-	-	-
8	-	-	-	-	1	-	-	-	1	1	-	-	1	-	-	-
9	-	-	-	-	-	1	-	-	1	1	1	-	-	1	-	-
10	-	-	-	-	-	-	1	-	-	1	1	1	-	-	1	-
11	-	-	-	-	-	-	-	1	-	-	1	1	-	-	-	1
12	-	-	-	-	-	-	-	-	1	-	-	-	1	1	-	-
13	-	-	-	-	-	-	-	-	-	1	-	-	1	1	1	-
14	-	-	-	-	-	-	-	-	-	-	1	-	-	1	1	1
15	-	-	-	-	-	-	-	-	-	-	-	1	-	-	1	1

Number of partitions is specified as an input in the config file to SCALE-Sim to divide the execution of DNN layers. Each partition is executed on a distinct PE in the network, and in the scenario of a L1 SRAM buffer miss, a request for data read and/or write is placed to any of the nodes in the network containing L2 SRAM buffers depending on the memory map. Note that number of partitions do not have to be the same as the number of nodes in the NoC. In the case of a $8 - ary\ 2 - cube\ mesh$ topology, partitioning on 30 PEs would lead to only the first *30* PEs $(0^{th} -\ 29^{th}\ PE)$ executing the DNN layers while all 64 nodes act as L2 SRAM buffer sources.

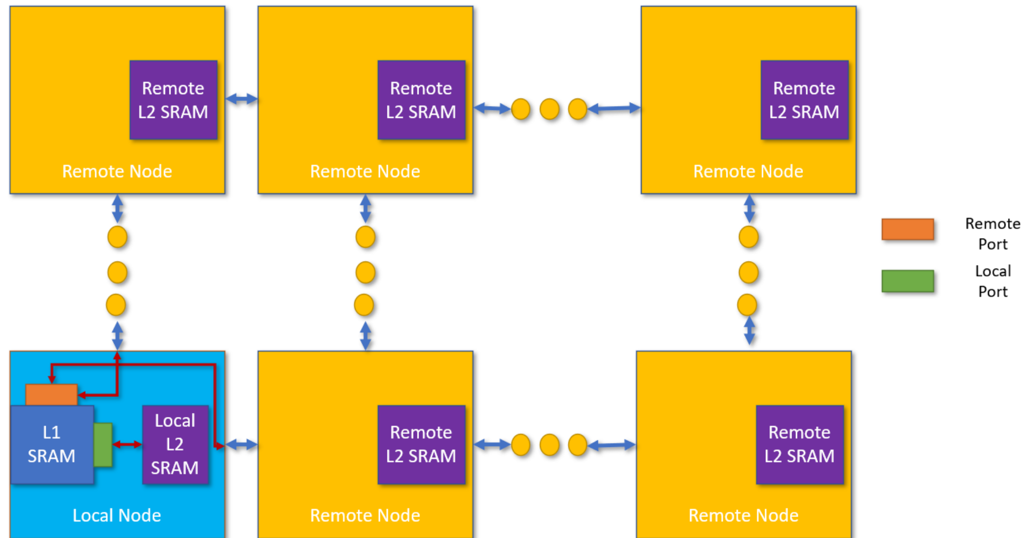


Figure 3.10: NoC Memory Hierarchy

Active buffer percentage (*ABP*) is a parameter that affects the prefetching policy into L1 local SRAM by controlling the ratio of active and inactive buffer. *ABP* can be anywhere from a $50\% - 50\%$ even ratio to a biased $99\% - 1\%$ active buffer-inactive buffer ratio. By sweeping through the possible ratios, interesting evaluations can be made to compare architectural parameters.

3.9 Experiments

The vast array of interesting evaluations of SCALE-Sim have yet to be observed since the tool adoption is still in its infancy. Papers have already been written on experiments performed by the developers of SCALE-Sim detailing the ability to test sweeps of architecture configurations on a suite of workloads [2, 21]. One of the most interesting results observed so far is that at scale, if the compute array is fully utilized, the memory bandwidth remains the prime bottleneck to performance irrespective of how much on-chip memory is allocated [21]. This bottleneck is most obvious when running large loads on the

hardware which demonstrates that the choice of interface bandwidth from memory is the most likely indicator of throughput no matter the efficiency in compute such as element reuse and/or high utilization.

For evaluations with respect to NoC designs, NoC Modeling Framework (NMF) is used to create a network of nodes containing L2 SRAM data banks and/or systolic arrays running independent partitions each acting as a PE in the NoC. In the experiments presented below, differences in performance metrics for popular DNNs used for DL inference are compared using various configurations to gain insight into the importance of certain architectural and controller parameters.

3.9.1 GPT2 & ResNet-50 Performance Evaluations using NMF

GPT2 is a large transformer-based language model with 1.5 billion parameters trained on a dataset of 8 million web pages [32]. The model contains 6 CONV layers labeled Linear1, QKT, QKTV, Linear2, PW-FF-L1, and PW-FF-L2. Furthermore, ResNet-50 a residual DNN that rose to fame after winning the ImageNet challenge in 2015 for its ability to effectively train a large number of layers [33]. ResNet-50 consists of 5 stages, conv1, conv2_x, conv3_x, conv4_x, conv5_x, and a fully connected layer. Each of these two models can be used as input DNNs to SCALE-Sim in conjunction with NMF to gather performance loss due to stall cycles.

To provide motivation for experiments involving these two DNNs, Linear1 from GPT2 has a base requirement of $7{,}680{,}094$ cycles without partitioning for an output stationary dataflow on a $32x32$ systolic array compute architecture. Therefore, the greatest optimization possibility is 43,894 stall cycles which is $.568\%$ of the total runtime. Meanwhile, conv2a_1 has a base requirement of $65{,}630$ cycles with the same

configurations. In this case, the greatest optimization possibility is 495,543 stall cycles which is 88.3% of the total runtime. The substantial optimization possibility differences indicate an underlying pattern in NoCs that is analyzed in Section 3.9.2 while this section provides experiments evaluating configuration parameters such as active buffer percentage (ABP) and partitioning (P).

An experiment is performed to gain insight on the effect of P on a GPT2 workload executed in a NoC. Six independent configurations are swept across various bandwidths assuming BW_{loc} for a PE is 10000 $bytes/cycle$ and the sum of remote BW accesses is $BW_{remote} = (n-1) * BW_{loc}$. Each of the 6 configurations is a combination of NoC topology: $8-ary\ 2-cube\ mesh$, $8-ary\ 4-flat\ flattened\ butterfly$, or $64\ node\ fully-connected\ network$, and partitions: $P = 1$ or $P = 10$. The 6 configurations are executed independently on SCALE-Sim and the results for Linear1 are presented in Figure 3.11.

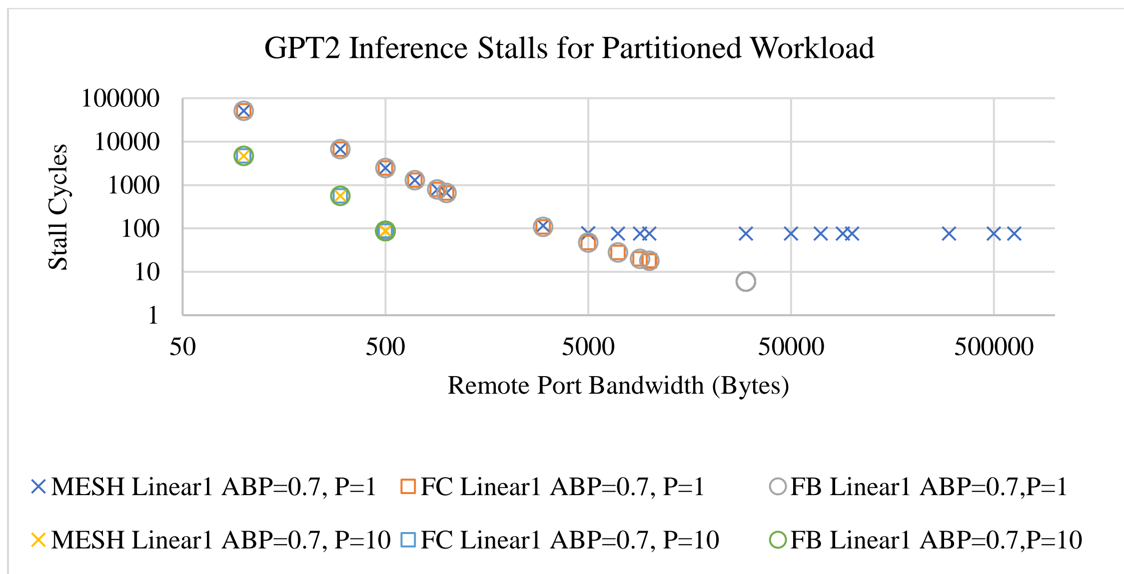


Figure 3.11: GPT2 Partitioning Evaluation over BW Sweep

For the configurations with $P = 10$, no differences are found among each topology, and performance loss due to stalls falls to 0 with remote bandwidth greater than 500. The same cannot be observed for $P = 1$ since *mesh* continues to incur stall cycles even at high bandwidths. This would lead us to surmise that the latency of a *mesh* traversal is much too great to recover from even while using large bandwidths. Meanwhile, the results for *flattened butterfly* and *fully connected* are almost identical with a net difference of 6 stall cycles added across bandwidths tested. Based on these differences, a possible claim is that partitioning removes the need for high bisection bandwidth and low diameter in a NoC. By evenly distributing the workload among the NoC and using a communication aware data placement, the need for robust NoC designs can be eliminated [3].

The results above showcase the importance of partitioning in a NoC, however, in many cases, partitioning is not possible because of the added overhead and lack of resources in the architecture. In the next experiment, the differences between active buffer percentages: $ABP = 0.7$ and $ABP = 0.5$ are compared. For consistency in experiments, same topologies from the last experiment are used to gain insight into differences the double buffer memory policy configuration can make on performance.

Figure 3.12 illustrates the resulting stall cycles for the experiment conducted over the sweep of remote port bandwidths. The overarching take-away is that a more balanced buffer scheme has real benefits to counter-act performance loss due to remote fetches. The results for $ABP = 0.5$ were always at least equal and in many cases better than the results for $ABP = 0.7$. Furthermore, the performance results for various topologies matched with $ABP = 0.5$ configuration. Based on this result, the claim can be made that the active and

inactive buffer ratio allocation is a consequential architectural parameter for which performance gains are possible.

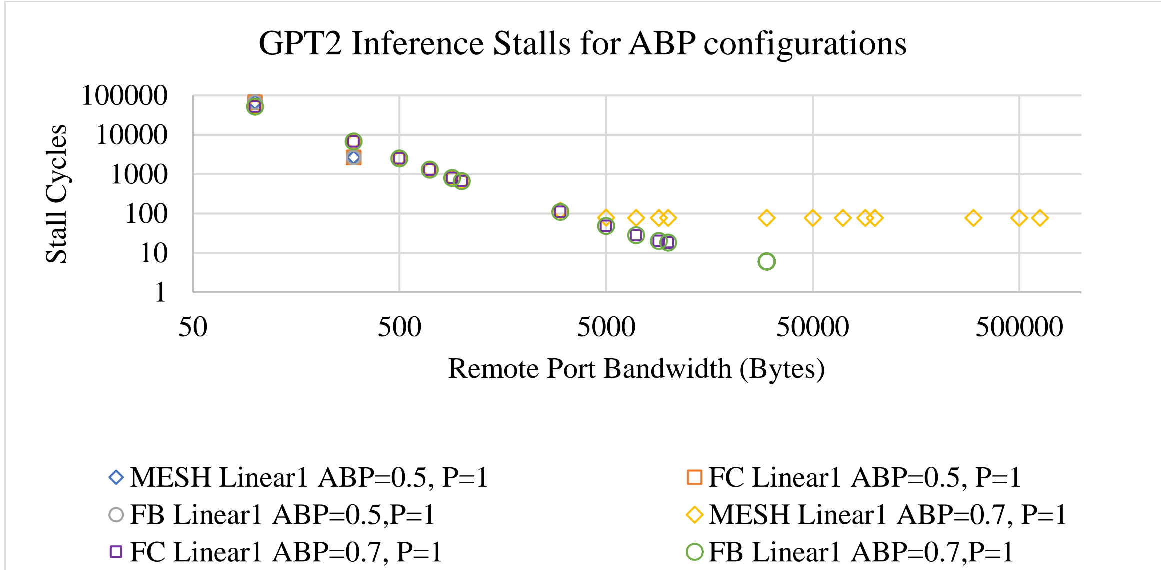


Figure 3.12: GPT2 ABP Evaluation over BW Sweep

To follow up previous evaluations, an experiment is performed to reveal insights as to the effect an optimized set of active buffer percentage (*ABP*) and partitioning *(P)* can have on performance. A sweep is performed across ABP from $0.5 - 0.9$ for one of 2 configurations with $P = 1$ or $P = 10$ over NoC topologies: $8 - ary\ 2 - cube\ mesh$, $64\ node\ fully - connected\ network$ illustrated in Figure 3.13.

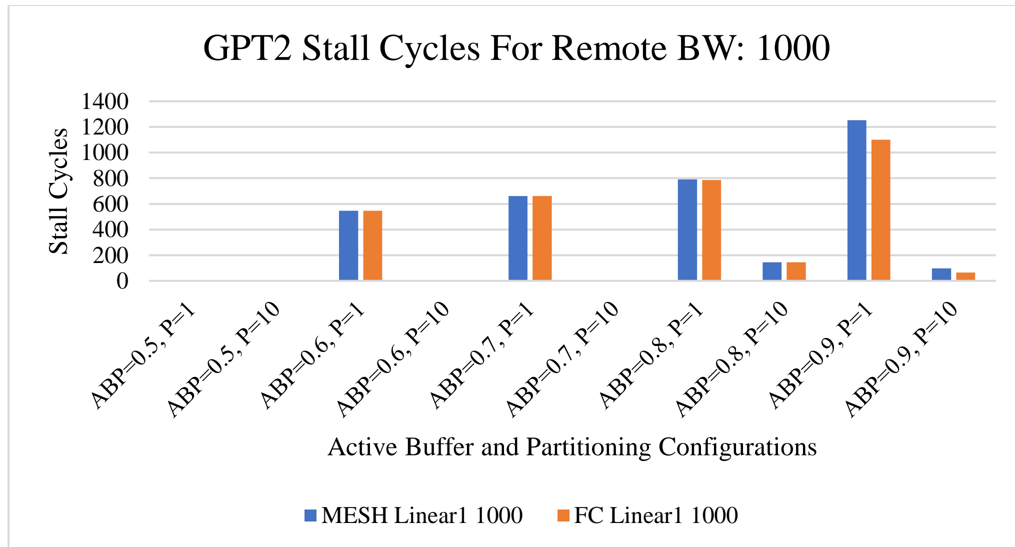


Figure 3.13: GPT2 ABP evaluation over partitioning for a *mesh* and *fully connected* topology with remote bandwidth of 1000 bytes

As previously expected, an even *ABP* leads to lesser sensitivity to prefetch latency. $ABP = 0.5, 0.6, \& \ 0.7$ lead to no stalls for partitioned workloads. Using no partitions, $ABP = 0.5$ is the only configuration without stall cycles. Furthermore, the differences in NoC topology are more pronounced at more biased *ABP* ratios while more even ratios mask the latency of the network. For $ABP < 0.8$, average variance between topologies is less than 0.01; however, for $ABP \geq 0.8$, variance between topologies is 797.78 cycles. Based on these results, a claim can be made that an optimized set of both parameters *ABP* and *P* leads to strong performance and that priority in design among the two parameters should be given to partitioning workloads. This claim can be strengthened with experiments covering a suite of workloads which are not present in these evaluations.

Next, an experiment is performed using Resnet-50 to distinguish the difference in performance between a partitioned vs. a non-partitioned workload for various layers. Using NMF with a *mesh* and *fully connected* topology, 2 sets of charts are generated evaluating

topologies individually. *Mesh* topology results are shown in Figure 3.14 and *fully connected* topology results are shown in Figure 3.15.

The resulting chart for *mesh* topology shows the substantial increase in timing performance for a partitioned workload versus a sequential workload without partitioning. 9 layers incurred no stall cycles before partitioning including the FC layer which are not included in comparison. Averaging the performance increase for the consequential layers, 99.24% stall cycles reduction is observed.

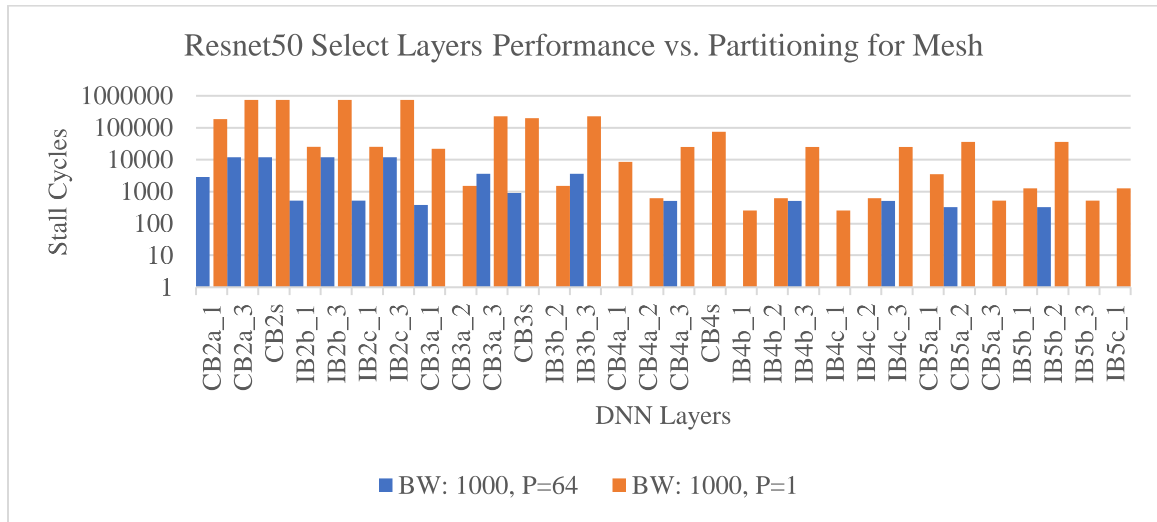


Figure 3.14: Resnet-50 Layers evaluation over partitioned configuration for execution on a *mesh*

Results for a *fully connected* topology show the substantial increase in timing performance for a partitioned workload versus a sequential workload without partitioning. Averaging the performance increase for the consequential layers, 99.28% stall cycles reduction is observed.

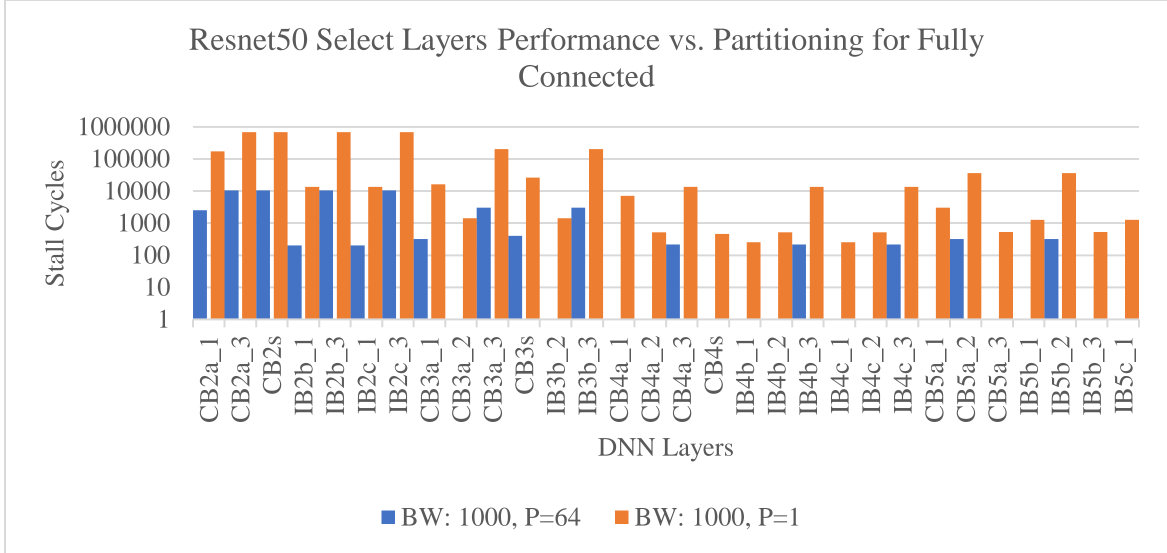


Figure 3.15: Resnet-50 Layers evaluation over partitioned configuration for execution on a *fully connected* topology

The base performance increase for $P = 1$ and remote bandwidth of 1000 $bytes$ was 13.99% for a *fully connected* topology over a *mesh* topology executing ResNet-50. Nevertheless, the similarity in performance increase between the two topologies indicates a possible correlation into the effects of partitioning on various layers regardless of differences in base performance. Another interesting observation is that stage 2 layers had the smallest improvement with 98.49% loss reduction for both topologies while stage 5 layers had the greatest improvement with 99.7% loss reduction for both topologies. This is likely due to the differences in network congestion which is investigated in Section 3.9.2.

3.9.2 GPT2 & ResNet-50 NoC Congestion Evaluation

Since NMF has real disadvantages due to its granularity in modeling an entire NoC during run-time experiencing contention and deadlock possibilities, experiments are performed to gain insight on the complete network bandwidth sensitivity to various workloads using

various NoC configurations [34]. ResNet-50 is the subject of the initial evaluations before moving to link load analysis comparing ResNet-50 and GPT2.

In this experiment, (*source, destination*) pairs are generated for the input operand communications throughout a $8 - ary\ 2 - cube\ mesh$ NoC and the resulting average injection rates are map over the total execution of each layer. Here we assume each data transfer packet is 1 byte. A visual representation of the evaluation is presented using heat maps illustrated in Figure 3.16. Each layer of each stage provides unique and interesting congestion results; however, only layers in stage 2 and stage 5 are analyzed, for brevity. Here the NoC congestion for layer conv2a_1 (*layer 1)* in stage 2 and conv5a_1 (*layer 43*) in stage 5 are compared.

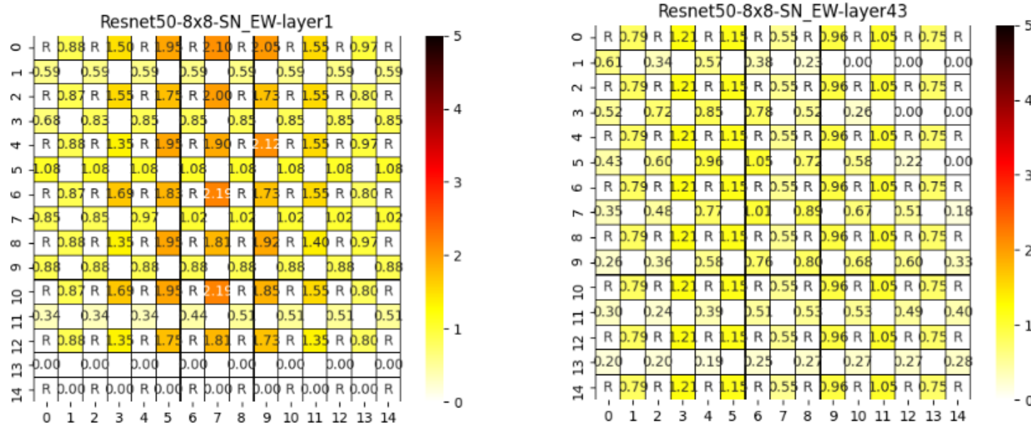


Figure 3.16: Resnet congestion evaluation using heat map

The heat map comparison indicates a much higher potential for congestion in conv2a_1 over conv5a_1. The greatest congestion link running conv2a_1 has a link load of 2.19 bytes while link loads in conv5_a are maxed at 1.21 bytes. The likely explanation for this result is advantageous data placement for conv5_a which results in closer communication pairs while conv2_a suffers from similar challenges as uniform random traffic which serves uniform communication for all (*source, destination*) pair combinations throughout the

network leading to bottlenecks around the bisection lines. Tracing the underlying reason for the difference in congestion, the difference can be attributed the differences in parameters between the two layers. Parameters are set as $stride = 2$ and $channels = 1024$ for conv5a_1 which reduces the number of accesses throughout the address space of the *ifmap* and prioritizes sequential addresses along the depth of a channel while conv2a_1 does not have benefit from advantageous parameters with $stride = 1$ and $channels = 64$.

A potential solution to congestion would be communication-aware data placement using an iterative algorithm similar to the proposed ideas in Simba architecture [3]. Another approach would be to design a NoC that purposefully sizes link bandwidths to serve specific workload traffic, thereby restricting optimal performance to a few workloads.

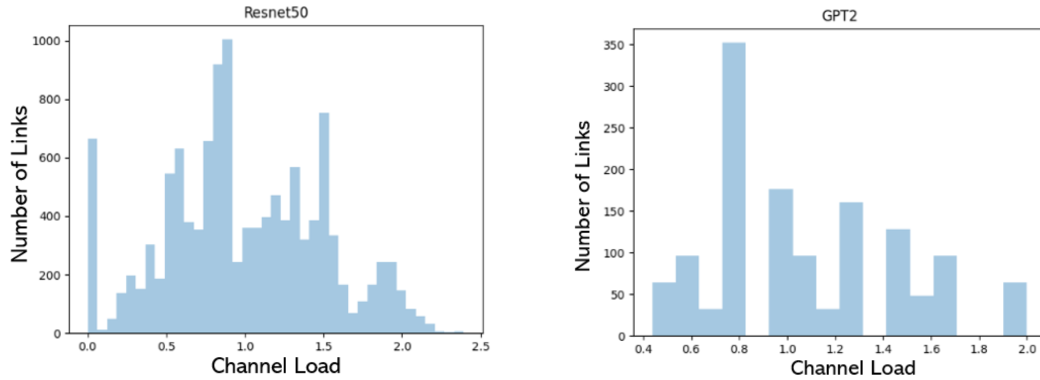


Figure 3.17: Link load distribution for GPT2 and Resnet50 layers

Taking a closer look at the distribution of channel load for Resnet-50 and GPT2 in Figure 3.17 reveals a generally random distribution of link loads with a mode of about 0.8. For both DNNs, the potential for congested links exists with link loads reaching up to 2.37 for some layers in Resnet-50. This evaluation gives added motivation to design custom heterogenous NoCs in chip-multiprocessors specific to application requirements.

CHAPTER 4

SIGMA BUILDING BLOCKS

Chapter 2 and 3 describes SCALE-Sim, a tool used to model scale-out DNN inference on systolic arrays. Unfortunately, emerging GEMMs in DL are highly irregular and sparse, which lead to poor data mappings on systolic architectures. A microarchitecture of a flexible and scalable GEMM accelerator is proposed that can handle arbitrary amounts of sparsity, arbitrary irregularity in GEMM dimensions, while guaranteeing close to full compute utilization named SIGMA. SIGMA performs 5.7 × better than systolic array architectures for irregular sparse matrices and roughly 3 × better than state-of-the-art sparse accelerators [6].

The fundamental building block and key novelty of SIGMA's compute fabric is a processor named Flexible Dot Product Engine (*Flex-DPE*) that can map GEMMs of arbitrary shapes and sparsity distributions via rich interconnect fabric. Within each Flex-DPE includes a novel reduction tree microarchitecture named Forwarding Adder Network (FAN) and a distribution network supporting flexible dataflows into the architecture. A k-sized Flex-DPE consists of k multipliers, k-1 adders, local buffers, a control unit, and flexible interconnects [6]. The design for the Flex-DPE was composed in Verilog RTL, synthesized using Synopsys Design Compiler on a 28 nm process, and place & routed using Cadence Innovus.

The next sections describe the logic design of the Flex-DPE which was conceived and developed in collaboration with *Eric Qin.* Individually, my contribution to this work was Verilog implementation of the Multiplier, Adder, Multiplier Local Buffer, Control Unit, and Flexible Interconnects in the FAN.

4.1 Multiplier

The multiplier was designed in Verilog to support bfloat16, a numerical format being adopted industrywide for neural networks [35]. In this format, 16 bits are used to represent a floating-point value. Bit 15 is the sign of the value, bit $14 - 7$ is the exponent value, and bit $6 - 0$ is the fraction or mantissa value illustrated in Figure 4.1. This format is a truncated version of the 32-bit binary32 format losing precision in the fraction. To ensure the output of the multiplier is consistent with the bfloat16 format, a multiplication normalizer is used to ensure accuracy while removing the extra precision mantissa bits reducing to 7 bits. To preserve timing integrity, a stand-alone multiplication computation always occurs in a single cycle.

4.2 Adder

The adder was designed in Verilog as a float32 data format adder. This format is consistent with the binary32 format which preserves extra precision by adding 16 extra bits to the bfloat16 mantissa in Figure 4.1. An addition normalizer is used to ensure the output of the adder is consistent with the bfloat32 format. The addition is computed in a single cycle to preserve system integrity. The output of the multiplier is concatenated with 16 bits of 0's to form the 32-bit input to the adder. The extra bits ensure greater precision from the adder operations since the chain of adder operations can cause exponential precision loss

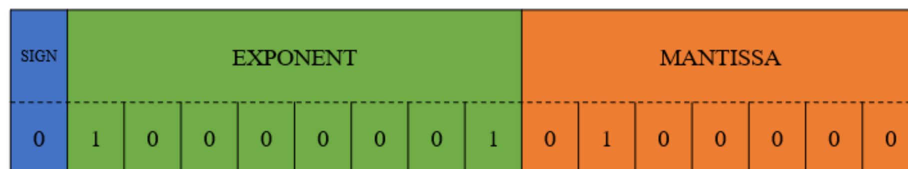


Figure 4.1: bfloat16 bit layout with most significant bit (MSB) on left representing the decimal value 5.0 = $(-1)^{signbit} * 2^{(exponent_{value}-127)} * \left(1 + \frac{mantissa_{value}}{128}\right); signbit = 0, exponent_{value} = 129, \& \ mantissa_{value} = 32$

4.3 Local Buffer

Each multiplier has a local buffer associated with it which stores a stationary value. This buffer is used to maximize data reuse for a stationary element (input activation or weight) while the non-stationary element is streamed in. If the input stationary bit is toggled on, the stationary buffer value is used for computation. If the reset bit is turned on, the local buffer element is cleared, and the new input value is stored acting as the new stationary value. This process would lead to an invalid multiplier output which is handled by turning off the output valid bit. The management of the reset bit is performed by the control unit described in Section 4.4

4.4 Control Unit

A control unit is used to determine how the multiplier or adder input and/or outputs are forwarded down the reduction tree. Each compute block: bfp16_mult.v and fp32_adder.v has a switch framework named mult_switch.v and adder_switch.v, respectively, that interfaces with the control unit.

The control unit toggles the valid bit of the input and/or the reset bit of the local buffer. Using the input valid bits, the multiplier switch toggles the valid bit of the output of the multiplier. If the valid signal is switched off, the adders down the path line will continue to invalidate the output valid bits as the values are streamed through the reduction tree.

The reduction tree incorporates the concept of a virtual neuron (VN) which allows for output buffers at each stage to be filled and pipelined down the tree. The control unit

controls the forwarding logic in the adder switch and the VN completion logic by selecting out of 5 options for non-edge adders described in the pseudo-code below:

Algorithm A pseudo-code for adder switch logic for non-edge adders

```
if input_valid then
   case (ctr_cmd == 1) then //forward both inputs
         fwd_output = [left_input, right_input]
         vn_output = [NULL, NULL]
         vn_valid = [False, False]
   case (ctr_cmd == 2) then //compute addition and forward to both paths
         fwd_output = [adder_output, adder_output]
         vn_output = [NULL, NULL]
         vn_valid = [False, False]
   case (ctr_cmd == 3) then //forward right input, set left input as VN output
         fwd_output = [left_input]
         vn_output = [right_input, NULL]
         vn_valid = True, False
   case (ctr_cmd == 4) then //forward left input, set right input as VN output
         fwd_output = [right_input]
         vn_output = [left_input, NULL]
         vn_valid = [True, False]
   case (ctr_cmd == 5) then //set both inputs as VN output
         vn_output = [left_input, right_input]
         vn_valid = [True, True]
end if
```

4.5 Flexible Interconnects

Each adder which is not located in the final level and is not an edge adder, has two interconnects in the FAN for data forwarding illustrated in Figure 4.2. The control unit manages the data forwarding pattern ensuring a spatial reduction requiring $O(\log_2 m)$ cycles for a m-sized dot product [6].

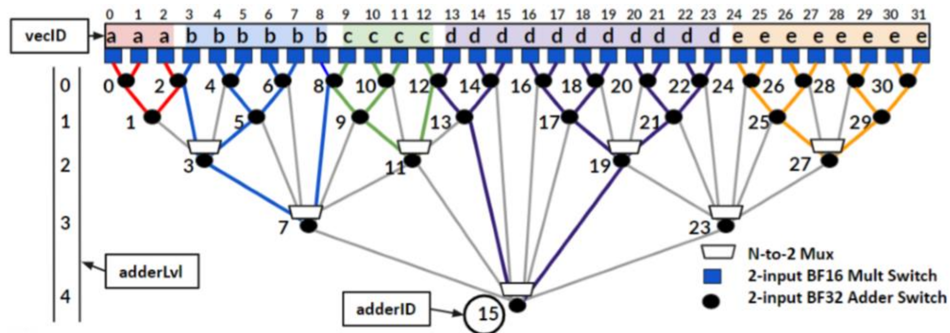


Figure 4.2: FAN topology with 32 multipliers, 31 adders, and flexible interconnects for data forwarding in a $32 - sized$ Flex-DPE

CHAPTER 5

CONCLUSION AND FUTURE WORK

Current innovation in deep learning accelerator development is hindered by the lack of open-sourced resources to the design and research community. SCALE-Sim-v2 gives us the ability to explore novel accelerator designs on systolic array substrates in a fast and convenient manner. Furthermore, the SCALE-Sim-v2 interface is highly modular allowing for greater tool developments from the open-sourced development community.

Based on current experiments exploring partitioning, double buffered prefetch memory policies, and remote bandwidth using the NoC Modeling Framework, real insights are gained as to the performance bottlenecks in accelerators and Chip Multi-Processors. Based on experimental results, architectural parameters targeting increased partitioning, partitioning scheme, and balanced double buffer ratios are arguably more vital to high performance than using a NoC topology with a high bisection bandwidth and low diameter. Furthermore, in the complete NoC link load evaluation, experimental results show that congestion due to high channel loads is dependent on the workload. Therefore, two approaches can be used to improve performance. One solution is to use communication-aware data placement to restrict high diameter communication. An alternative solution is creating custom NoC designs with adjusted bandwidths on links suiting application requirements.

5.1 Future Work

5.1.1 C++ Syntax Porting

The current simulator repository is restricted to Python 3 compatibility. For future integration with various architecture simulator tools and to speed up execution of the base simulator, a C++ compatible syntax would be required with parallel execution capabilities provided by CUDA and Nvidia GPUs or with C++ compatible libraries such as PASL [36].

5.1.2 Versatile Compute Architecture Support

Systolic array is presently the most widely explored design for compute architectures concerned with DNN inference because of the simplicity of design and easily translatable dataflows leading to low overhead. The disadvantage of this approach is the poor mapping of highly irregular and sparse operands. For this reason, architectures such as SIGMA have been proposed that use a flexible architecture offering higher utilization and better performance than systolic array architectures [6].

SCALE-Sim should be able to support modeling various types of compute architectures and the full-suite of dataflows mapped on the compute architectures such as row stationary dataflow on Eyeriss architecture [4].

REFERENCES

[1] V. Sze, Y.-H. Chen, T. Yang and J. S. Emer., "Efficient processing of deep neural networks: a tutorial and survey," in *IEEE-HPCA*, 2017.

[2] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina and T. Krishna, "SCALE-Sim: Systolic CNN Accelerator Simulator," *arXiv: 1811.02883 [cs.DC],* 2018.

[3] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik and N. Jiang, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *MICRO*, 2019.

[4] Y. Chen, T. Krishna, J. S. Emer and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *ISSCC*, 2016.

[5] H. Kwon, A. Samajdar and T. Krishna, "Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *ASPLOS*, 2018.

[6] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul and T. Krishna, "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training," in *IEEE-HPCA*, 2020.

[7] *https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/,* 2015.

[8] Xilinx, "FPGA Acceleration of Matrix Multiplication for Neural Networks," 27 February 2020. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1332-neural-networks.pdf. [Accessed 25 March 2020].

[9] H. H. Aghdam and E. J. Heravi, Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification, Cham: Spring International Publishing AG, 2017.

[10] S. Jothilakshmi and V. N. Gudivada, "H," *Handbook of Statistics,* vol. 35, pp. 301-340, 2016.

[11] G.-M. Tang, D.-R. Fan, X.-C. Ye and P.-Y. Qu, "Logic Design of a 16-bit Bit-Slice Arithmetic Logic Unit for 32-/64 bit RSFQ Microprocessors," *IEEE Transactions on Applied Superconductivity,* vol. 28, no. 4, pp. 1-5, 2018.

[12] C. Zhuang, H. Su, Q. Yang, J. Shen, M. Wen and C. Zhang, "P4 to FPGA-A Fast Approach for Generating Efficient Network Processors," *IEEE Access,* vol. 8, pp. 23440-23456, 2020.

[13] C. Lameter, "An Overview of Non-Uniform Memory Access," *Communications of the ACM,* vol. 56, no. 9, pp. 59-65, 2013.

[14] N. E. Jerger, T. Krishna and L.-S. Peh, On-Chip Networks: Second Edition, San Rafael: Morgan & Claypool, 2017.

[15] J. Lee, C. Nicopoulos, S. J. Park, M. Swaminathan and J. Kim, "Do we need wide flits in Networks-on-Chip?," in *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Natal, 2013.

[16] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen and O. Temam, "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," in *ASPLOS '14*, Salt Lake City, 2014.

[17] "Cloud tpu," 2019. [Online]. Available: https://cloud.google.com/tpu.

[18] C. Abzug, "Computer Architecture," *AccessScience,* 2019.

[19] F. Schuiki, M. Schaffner, F. K. Gürkaynak and L. Benini, "A scalable near-memory architecure for training deep neural networks on large in-memory datasets," *IEEE Trans. Comput.,* vol. 68, no. 4, pp. 484-497, 2019.

[20] K. Cho, I. Lee, H. Lim and S. Kang, "Efficient Systolic-Array Redundancy Architecture for Offline/Online Repair," *Electronics,* vol. 9, no. 2, p. 338, 2020.

[21] A. Samajdar, A. Himanshu, N. Raina, V. Nadella, A. Mathuriya, S. Maniputruni and T. Krishna, "Technology and Architecture Opportunities for Breaking the Bandwidth Ceiling to Accelerate Billion Operand AI Inference," in *DAC*, 2020.

[22] M. McKeown, A. Lavrov, M. Shahrad, P. J. Jackson, Y. Fu, J. Balkind, M. Nguyen, K. Lim, Y. Zhou and D. Wentzlaff, "Power and energy characterization of an open source 25-core manycore processor," *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA),* pp. 762-775, 2018.

[23] L. Peh, T. Krishna and W. Kwon, "Locality-oblivious cache organization leveraging single-cycle multi-hop NoCs," *Architectural support for programming languages and operating systems (ASPLOS 14),* vol. 49, no. 4, pp. 715-728, 2014.

[24] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize and M. Roth, "Challenges of Memory Management on Modern NUMA Systems," *Communications of the ACM,* vol. 58, no. 12, pp. 59-66, 2015.

[25] *http://cs231n.github.io/convolutional-networks/,* 2015.

[26] R. S. Ramanujam, V. Soteriou, B. Lin and L.-S. Pen, "Extending the Effective Throughput of NoCs With Distributed Shared-Buffer Routers," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 30, no. 4, pp. 548-561, April 2011.

[27] S. Rashidi, S. Sridharan, S. Srinivasan and T. Krishna, "ASTRA-SIM: Enabling SW/HW Co-Design Expoloration for Distributed DL Training Platforms," in *In Proc of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Boston, 2020.

[28] S. Chattopadhyay and S. Kundu, Network-on-Chip: The Next Generation of System-on-Chip Integration, Boca Raton: CRC Press, 2018.

[29] G. Kim, J. Kim, J. H. Ahn and J. Kim, "Memory-centric system interconnect design with Hybring Memory Cubes," in *22nd International Conference on Parallel Architectures and Compilation Techniques*, Edinburgh, 2013.

[30] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Communication of the ACM,* vol. 60, no. 6, pp. 84-90, 2017.

[31] J. Kim, W. J. Dally and D. Abts, "Flattened Butterfly: A Cost Efficient Topology for High-Radix Networks," in *The 34th International Symposium on Computer Architecture*, San Diego, 2007.

[32] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, "Language models are unsupervised multitask learners," *Technical Report,* 2018.

[33] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, 2016.

[34] A. K. Mishra, O. Mutlu and C. R. Das, "A Heterogeneous Multiple Network-On-Chip Design: A Application-Aware Approach," in *DAC '13*, Austin, 2013.

[35] C. Kloss, "Intel Nervana Neural Network Processor: Architecture Update," 03 March 2020. [Online]. Available: https://www.intel.com/content/www/us/en/artificial-

intelligence/posts/intel-nervana-neural-network-processor-architecture-update.html. [Accessed 10 April 2020].

[36] U. A. Acar, A. Charguerand and M. Rainey, "Parallel Computing in C++ with PASL," April 2014. [Online]. Available: http://www.cs.cmu.edu/afs/cs/academic/class/15210-s15/www/lectures/pasl.html#_preface. [Accessed 11 April 2020].

[37] N. Agarwal, T. Krishna, L.-S. Peh and N. K. Jka, "Garnet: A detailed on-chip network model inside a full-system simulator," *ISPASS,* 2009.

[38] J. Z, K. Rangineni, Z. Ghodsi and S. Garg, Thundervolt: enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators, 2018.

[39] P. Lotfi-Kamran, B. Grot and B. Falsafi, "NOC-Out: Microarchitecting a Scale-Out Processor," in *45th Annual IEEE/ACM International Symposium on Microarchitecture*, Vancouver, 2012.