In-SRAM Computing for Neural Network Acceleration

by

Charles S. Eckert

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2022

Doctoral Committee:

Associate Professor Reetuparna Das, Chair
Professor Trevor N. Mudge
Professor Dennis Sylvester
Professor Lingjia Tang

Charles S. Eckert

eckertch@umich.edu

ORCID iD: 0000-0002-8839-9890

© Charles S. Eckert 2022

ACKNOWLEDGMENTS

I would like to deeply thank my advisor Processor Reetuparna Das. Reetu has been instrumental during my PhD, always providing me with advice and support. This PhD wouldn't have been possible without her support.

I would like to thank Professor Trevor Mudge, Professor Lingjia Tang, and Professor Dennis Sylvester for agreeing to be a part of my committee and for the helpful feedback they provided throughout the process.

I am also thankful for my many collaborators during my Ph.D. In addition to being on my committee, Professor Sylvester along with Professor David Blaauw were very valuable contributors from the ECE department. I also frequently collaborated with Ravi Iyer and Charles Augustine from Intel who were able to provide an industry perspective to my research. Additionally, I worked with fellow Ph.D. students Arun Subramaniyan, Xiaowei Wang, and Jingcheng Wang who were all invaluable resources for questions and brainstorming with.

I would like to acknowledge my generous funding support from Applications Driving Architectures (ADA) Research Center and my NDSEG fellowship. NDSEG is a prestigious fellowship DoD National Defense Science and Engineering Graduate fellowship that provided me funding for the final three years of my Ph.D. In addition to funding, the fellowship gave me mentorship opportunities with two Air Force Research Labtoroies researchers, Stanley Wenndt and Andrew Noga as well as the opportunity to network at the NDSEG conference. I would like to thank Stanley and Andrew for their mentorship and support. I would like to thank Professor Blaauw, Ravi Iyer, and my advisor Professor Das for writing my recommendation letters for my fellowship.

My fellow labmates have all been wonderful people to work with, share an office with, and

hang out with outside of work; Daichi, Xiaowei, Vidushi, Arun, Tim, Hari, Yufeng, Ali, Yichen, Jack, Kush, and Yuwei.

I would also like to thank my family, my friends from undergrad, Daelin, Bar, Naz, Bryan, Zuzanna, and Hayley, and hometown friends; Ben, Bobby, Brandon, and Pat who all provided me with support. A shout out to Daelin in particular who demanded a shout-out in exchange for helping proofread my dissertation, Zuzanna and Julie instead simply helped out of the kindness of their hearts. I am also thankful for my climbing partners: Tim, Amrit, and Marisa. I was only injured twice while Tim was belaying me! I also made many wonderful friends at the university and in the area who I would like to thank; Reid, Tan, Alextia, Gabby, Tara, Andrew, Barrett, Will, Miles, Christian, and Julie.

TABLE OF CONTENTS

Acknowledgments . . . . **ii**

List of Figures . . . . **vii**

List of Tables . . . . **ix**

Abstract . . . . **x**

Chapter

1 Introduction **1**

2 Background **8**

- 2.1 In-SRAM Computing 8
- 2.2 Deep Neural Networks 11
- 2.3 Accelerating Neural Networks 16

3 Prior Works **20**

- 3.1 DNN Accelerators 20
- 3.2 Processing in Memory, In-Memory, and Near-Memory Computing 25

4 Neural Cache **32**

- 4.1 Introduction 32
- 4.2 Neural Cache Organization 35
- 4.3 Neural Cache Arithmetic 36
  - 4.3.1 Bit-Serial Arithmetic 36
  - 4.3.2 Addition 38
  - 4.3.3 Multiplication 40
  - 4.3.4 Reduction 41
  - 4.3.5 SRAM Array Peripherals 41
  - 4.3.6 Transpose Gateway Units 42
- 4.4 Neural Cache Architecture 44
  - 4.4.1 Data Layout 44
  - 4.4.2 Data Parallel Convolutions 47
  - 4.4.3 Orchestrating Data Movement 48
  - 4.4.4 Supporting Functions 50
  - 4.4.5 Batching 52

4.4.6 ISA support and Execution Model . . . 52
4.5 Evaluation Methodology . . . 53
4.6 Results . . . 55
4.6.1 Latency . . . 55
4.6.2 Batching . . . 57
4.6.3 Power and Energy . . . 58
4.6.4 Scaling with Cache Capacity . . . 60
4.7 Summary . . . 60

5 Eidetic . . . 61

5.1 Introduction . . . 61
5.2 Eidetic Abstractions and Execution Model . . . 63
5.2.1 Matrix Multiplication Systolic Arrays . . . 63
5.2.2 Dataflow Architecture . . . 65
5.2.3 Dependencies and Layers . . . 65
5.2.4 Concurrent Execution of Pipelined Layers . . . 66
5.3 Architecture . . . 68
5.3.1 PE Unit . . . 68
5.3.2 PE Memory Addressing . . . 70
5.3.3 Column Block . . . 71
5.3.4 Central Control . . . 73
5.4 Data Mapping and Eidetic Instructions . . . 75
5.4.1 Data Mapping . . . 75
5.4.2 Instructions . . . 78
5.4.3 Programmability Example . . . 80
5.5 Putting It Together . . . 81
5.6 Evaluation methodology . . . 82
5.7 Results and Analysis . . . 84
5.7.1 Performance . . . 84
5.7.2 Energy and Area . . . 86
5.7.3 Throughput comparison with other accelerators . . . 87
5.7.4 Versatile Precision . . . 88
5.8 Summary . . . 89

6 Comparison of Cache vs Custom Architecture . . . 90

6.1 Introduction . . . 90
6.2 Data Mapping . . . 92
6.3 Results and Analysis . . . 94
6.3.1 Eidetic comparison to *Neural Cache* . . . 94
6.3.2 Custom Interconnect and Dataflow Architecture . . . 95
6.3.3 Increase Storage for Weight Pinning and In-Place Computation . . . 96
6.3.4 Efficient MACs . . . 97
6.3.5 Mapping Granularity . . . 98
6.4 Summary . . . 98

7 Conclusion . . . 100

Bibliography **103**

LIST OF FIGURES

Figures

2.1 Prototype test chip [1]. 10
2.2 SRAM circuit for in-place operations. Two rows (WL_i and WL_j) are simultaneously activated. An *AND* operation is performed by sensing bit-line (BL). A *NOR* operation is performed by sensing bit-line complement (BLB). 10
2.3 Computation of a convolution layer. 13
2.4 Typical LSTM Cell showing micro-operations performed 14
2.5 GNMT Encoder Layout 15

3.1 Google TPU with its 2D Systolic Array 21

4.1 Neural Cache overview. (a) Multi-core processor with 8-24 Last Level Cache (LLC) slices. (b) Cache geometry of a single 2.5MB LLC cache slice with 80 32KB banks. Each 32KB bank has two 16KB sub-arrays. (c) One 16KB sub-array composed of two 8KB SRAM arrays. (d) One 8KB SRAM array re-purposed to store data and do bit line computation on operand stored in rows (A and B), (e) Peripherals of the SRAM array to support computation. 36
4.2 Addition operation 37
4.3 Reduction Operation 38
4.4 Multiplication Operation, each step is shown with four different sets of operands . . . 39
4.5 Bitline peripheral design 42
4.6 Transpose Memory Unit (TMU) 43
4.7 Neural Cache Data Layout for one LLC Cache Slice. 43
4.8 Array Data Layout (a) MACs (b) Reduction. 46
4.9 Partitioning of Convolutions between Slices. 47
4.10 SRAM Array Layout. 53
4.11 Inference latency by Layer of Inception v3. 56
4.12 Inference Latency Breakdown. 56
4.13 Total Latency on Inception v3 Inference. 58
4.14 Throughput with Varying Batching Sizes. 59

5.1 (a) Systolic Array (b) Optimized Systolic Array 64
5.2 Eidetic Overview 67
5.3 SRAM-based *Eidetic* PE Unit 71
5.4 Column Block Logic Overview 72

5.5 Control Flow of Central Control Unit. Bold text between units indicates the passed information was created in one of the units. 74
5.6 Mapping of weights to a systolic array 76
5.7 Putting it together 81
5.8 Throughput of TPU Batch Size 1024 vs *Eidetic* Input Batch Size 16 85
5.9 Latency of TPU Batch Size: 8 vs *Eidetic* Input Batch Size: 16 85
5.10 *Eidetic* throughput by input batch size 86
5.11 Power Breakdown 87
5.12 Avg. and Max Cycles for MAC – different precision 89

6.1 Mapping of *Eidetic* on *Neural Cache* 92
6.2 Breakdown of *Neural Cache*'s Throughput 96
6.3 Breakdown of *Neural Cache*'s Energy 96

LIST OF TABLES

Tables

4.1 Parameters of the Layers of Inception V3. 53
4.2 Baseline CPU & GPU Configuration. 54
4.3 Energy Consumption and Average Power. 59
4.4 Scaling with Cache Capacity (Batch Size=1). 59

5.1 Area breakdown of *Eidetic* by component. There are 54 Systolic Column Blocks each with 2.375MB of compute SRAM 86
5.2 Area and power normalized throughput comparison for different accelerators. For comparison, the metrics in the last five rows are normalized to 22nm using DeepScaleTool [2]. For TPUv3 and Hanguang 800, we conservatively assume reported numbers are in 14nm for normalization to 22nm. 87
5.3 ML Microbenchmarks Eidetic Mapping and TPU V2 and Eidetic Performance. 88

6.1 Comparison of Eidetic and *Neural Cache* Latency and Throughput 95
6.2 Comparison of Eidetic and *Neural Cache* Energy and Power 95

ABSTRACT

For decades, the computing paradigm has been composed of separate memory and compute units. Processing-in-Memory(PIM) has often been proposed as a solution to break past the memory wall. With PIM, compute logic is moved near the memory, which can reduce the data movement. In-memory computing expands on PIM by morphing the memory into hybrid memory compute units, where data can be stored and computed on in-place. Recent work has modified SRAM arrays to allow logical operations to be performed directly inside the arrays. Our work extends basic logical operations and additionally adds support for arithmetic operations.

Coinciding with the rise of increasing memory on-chip and more focus on near and in-memory computing is the ascendance of neural networks. Neural networks are highly data-parallel applications that are challenging to accelerate due to being data-bound, compute-bound, or both. In-memory computation reduces on-chip data movement and will increase the amount of compute available as well as the amount of storage in a custom chip. These factors can greatly alleviate compute and data bottlenecks.

First, this thesis observes that SRAM memory has increasingly dominated the on-chip area for general-purpose processors. This area comes at the cost of compute potential and can be repurposed to function as a dual storage and compute unit. The benefits of such repurposing are greatly expanding the parallel compute capability of the chip while also reducing the on-chip data movement, all with minimal area increase. When SRAM is repurposed, the storage area can be reclaimed with minimal overhead. Modifications to the SRAM arrays are presented that allow the SRAM to function as hybrid compute/storage units capable of arithmetic operations. Additionally,

this work presents a mapping strategy for supporting CNNs in the hybrid SRAM storage compute arrays.

Second, this thesis proposes a custom ASIC called Eidetic that utilizes hybrid compute/storage SRAM arrays as both its primary storage and compute units. Repurposing a processor's SRAM is hamstrung by maintaining the cache's original functions and area footprint. Too many modifications to the cache would render the solution undesirable to chip designers. By further customizing the SRAM we can create more efficient PE units. Additionally, the increased SRAM storage allows more weights to be stored on-chip. Finally, the custom ASIC allows for a control logic that supports a graph-based programming model that further reduces off-chip data movement. These customizations allow Eidetic to target data-bound applications such as RNNs and MLPs.

Third, we propose a detailed comparison between the in-cache and ASIC approaches to ML acceleration. Between repurposing the cache and creating an SRAM-based custom ASIC, in-SRAM computing offers multiple viable approaches. In-Cache computing is cheaper but comes with limitations, while an ASIC design is more expensive due to the total cost of ownership (TCO).

We compare the performance and energy efficiency of our repurposed cache with a server-class GPU and the baseline CPU. We similarly evaluate our custom ASIC to other state-of-the-art ASIC DNN accelerators. For both the repurposed cache and the ASIC, we develop cycle-accurate simulators to determine the performance.

CHAPTER 1

Introduction

Over the last two decades compute architecture has been increasingly limited by the gap between memory and compute known as the memory wall. The performance of compute on-chip is increasing at a much faster rate than the improvements in off-chip (DRAM) memory technology. The difference in improvement rates is leading to an imbalance in the speedups of compute and memory. As Amdhals's law dictates, speedups for one component of execution (compute) will result in continually decreased performance gains as the other components (memory) dominate execution time. Thus, the current paradigm of computer architecture has led us to a memory wall where memory bandwidth and memory energy have increasingly dominated execution time and energy. Another trend in computer architecture is the increase in the number of cores on-chip. Previously, the frequency could be increased from one generation to another without vastly increasing the amount of power used. However, since the end of Dennard scaling, this is no longer true, thus motivating increasing the number of cores on-chip to increase compute performance.

Overcoming the memory wall problem is a complicated task with several suggested solutions. One option is to address this gap by adding/increasing the amount of intermediate storage between the DRAM and compute. Static random-access memory (SRAM) is on-chip memory that is commonly found on most chips, particularly as the cache. The cache is typically organized in a multi-level hierarchy of SRAM, where from level to level, the amount of SRAM in that group is increased, but so is the time and energy to fetch data. By increasing the amount of on-chip storage, more data can be reused without going off-chip. However, increasing the amount of SRAM

comes with costs. More on-chip SRAM creates the need for a deeper and wider hierarchy, thus increasing the time and energy of on-chip data access. This is still less than the cost of accessing data from off-chip. Additionally, any area dedicated to on-chip storage means a reduction in the number of compute units as well as an increase in static/leakage power from the SRAM. Since memory access is the larger bottleneck, sacrificing some compute for storage will yield speedups. However, sacrificing one component of the execution for another will limit the possible speedups compared to a solution where neither is sacrificed. Thus increasing the on-chip area for storage is an imperfect solution.

Another solution involves rethinking the memory/compute paradigm and how they are separated on-chip. It is possible to move the compute closer to the memory. Moving the compute closer has been referred to as processing-in-memory (PIM) or near-memory computing. PIM has long been suggested as a solution to the memory wall. PIM helps alleviate issues with on-chip data movement with the compute units close to the memory units. However, the compute can be moved even closer. Instead of using separate compute and storage units, the two can be morphed together as a single unit where data is both stored and computed on. This style of architecture termed *in-memory computing* can offer intrinsically more efficiency compared to the traditional computing paradigm or even near-memory computing. Additionally, hybrid compute memory units can offer a lower overhead than separate compute and memory units.

The industry has largely responded with the first solution, with on-chip SRAM increasingly growing over the last two decades. Chips today have SRAM exceeding 3/4 of the total die area. However, these two solutions do not necessarily have to be separate. SRAM can be modified to support in-memory computing with minimal cost. This is a very attractive solution as on-chip memory would no longer come at sacrificing compute area. Additionally, SRAM is an existing technology that can be found on almost any chip. SRAM has ongoing research efforts to scale and reduce power, as well as an established manufacturing protocol. Less established technologies would require significant investments to implement and manufacture. And since it is found everywhere, incorporating in-SRAM computing into existing architectures is a more viable solution.

Further exasperating the memory wall bottleneck is the rise of data-intensive applications, such as neural networks. These data-intensive applications have created a data bottleneck from general-purpose CPUs to accelerators. Neural networks in particular suffer from both computation and data bottlenecks, making it difficult to balance both the amount of compute and storage on-chip. In-memory computing with SRAMs offers a way to achieve this balance.

Previous work for in-SRAM computing has provided the basis for logical operations but lacks support for arithmetic operations. By increasing the capabilities of the SRAM compute arrays to perform arithmetic operations, neural networks can be supported.

This dissertation offers two solutions for accelerating DNNs using in-memory computing to push back against Moore's slowdown and the memory wall. By using in-SRAM computing, we can transform existing processors and build custom chips that will see significant performance improvements. By offering a solution for both general purpose and customs chips, we show the vast potential of in-SRAM computing.

Firstly, our work in *Neural Cache* allows us to take an existing processor that is designed for general-purpose computing and expand its capabilities to parallel computing for neural network acceleration. The last level cache of the Xeon can be repurposed from a storage unit to a hybrid compute-storage unit capable of bit-serial arithmetic. This is accomplished primarily by augmenting logic to the peripherals of the SRAM array, resulting in a very small area penalty. The Intel Xeon E5-2697 v3 35MB cache can be transformed into 1,146,880 bit-serial ALUs. Additionally, *Neural Cache* provides a mapping for CNNs and data movement strategies that allow these bit-serial ALUs to take advantage of the inherent parallelism of DNNs for massive performance gains. All of this is accomplished without the need for a new chip or sacrificing the CPU's general-purpose computing capabilities. With the rise in demand for parallel processing, the ability to support both general-purpose and parallel processing is a huge advantage that cannot be overstated.

The work in *Neural Cache* presents bit-serial computing in SRAM arrays. With modifications to the bitline peripherals, we can perform addition, subtraction, multiplication, division, and

comparison on any two operands stored along a bitline, without moving them outside the array. Additionally, the computation inside the array is driven by an FSM rather than fixed logic. This allows us to support configurable precision without adding additional hardware to the SRAM arrays. Adding compute support to the SRAM arrays only incurs a 2% area overhead across the processor chip while adding 1,146,880 bit-serial ALUs.

In order to facilitate bit-serial computation, we first need to transpose our data. This can be done in software, but we instead developed hardware to transpose our data on the fly. Our transpose memory units (TMU) are a custom 8T SRAM array used solely to transpose data. The additional transistors are designed to add an extra set of bitlines and wordlines that are perpendicular to the originals. This allows data to be written horizontally into the array as usual, but then the perpendicular wordline can be activated to read data out vertically, as if along the bitline. As weights can be pre-transposed, only a few TMUs in the cbox for each slice are needed to transpose the input.

Additionally, this work provides a mapping for CNNs onto our newly compute-enabled cache. The filters and inputs can be unrolled into 2D matrices, with one dimension being the height and width of the filter, and the other the input and output channels. Along an SRAMs bitline, the unrolled filters and inputs can be stored. This allows us to perform multiple multiplications and accumulate the outputs along the bitline. Across the bitlines in the arrays, other channels of the output activation are computed. The reduction will be performed inside the SRAM arrays. The remaining input channels, followed by the output channels, will be mapped across the SRAM arrays in a bank, the banks in a way, the ways in a slice, and the slices on-chip. Small layers can be replicated to allow more output activations to be computed in parallel.

This data mapping is designed to exploit the parallelism in CNNs and reuse the weights as much as possible. Every multiply in the convolution is independent of each other; however, the multiplications within a filter window and across the filter channels will be accumulated. The next sliding window of inputs can be loaded into the array to reuse the weights that are stored in the SRAM arrays. This mapping allows accumulation to stay local while also reusing the weights.

Weight replication reduces the weight reuse, but will improve the parallelism that can be exploited, and therefore throughput.

Secondly in *Eidetic* we look to push the power of in-SRAM computing further by designing a custom architecture that uses in-SRAM computing as its fundamental building block. By creating a custom architecture instead of repurposing an existing cache, *Eidetic* is able to break past the limitations of *Neural Cache*. *Eidetic* can support many more compute units than *Neural Cache* without the CPU logic or cache overhead. *Eidetic* contains 3,538,944 bit-serial ALUs while remaining effectively area neutral with *Neural Cache*. Additionally, the architecture can be designed to more efficiently handle the data-bound problems in DNNs with a focus on pinning weights, facilitating data movements, and handling data dependencies. Finally, the SRAM PEs can be redesigned to improve their efficiency. These modifications give *Eidetic* significantly more compute power, but also ensure the availability of data to compute on.

By customizing the SRAM arrays and their organization, we can optimize our SRAM architecture to balance power, storage density, area, and compute throughput. The SRAM PEs can be organized into a systolic array allowing accumulation to be performed through more efficient data propagation rather than reduction across bitlines. A systolic array is a common dataflow architecture for neural networks that focuses on data reuse and data movement. Further, the compute operations themselves can be optimized with the addition of an input latch outside the SRAM arrays.

The vast increase in the number of SRAM arrays coupled with a custom array structure allows for new possibilities such as pinning weights, computing in place on the pinned weights, and executing multiple layers in parallel. *Eidetic* custom control logic allows on-demand processing of inputs for a network with complex dependencies and inputs of varying lengths. The custom control logic creates a second tier to the dataflow architecture, where the outputs of a layer are automatically routed to the input of the next layer. These optimizations are made to overcome potential data bottlenecks so that the vast number of compute units can be kept occupied.

Finally, in this dissertation, we discuss the usage of both approaches, including the cost and

benefits of each. Providing both of these solutions and an analysis of the two allows us to make a case for in-SRAM computing to take over the computing paradigm.

In summary, this dissertation offers the following contributions.

- Neural Cache, an architecture that repurposes the last level cache to support the inference of DNNs. This repurposing maintains the original cache structure and results in minimal increases in total area.

- SRAM modifications for Neural Cache that allow for arithmetic operations such as multiplication, division, subtraction, and addition. Each SRAM array is transformed into 256 vector bit-serial ALU.

- A data layout for Neural Cache to optimize the mapping of DNNs. The parallelism of DNNs is exploited across the cache structure of the SRAM arrays.

- Eidetic, an architecture that uses in-SRAM computing to accelerate neural networks. The ASIC design allows for more efficient neural network inference.

- Customization of the SRAM PEs to support scalar vector multiplication, resulting in simplified systolic arrays. The customized SRAM allows for reduced data movement and more efficient operations.

- A dataflow architecture capable of mapping neural networks based on high-level graphs. The graph-based programming reduces the amount of off-chip data movement.

- A comparison and a discussion between the two approaches. This discussion provides optimal use cases and makes the case for the range of in-SRAM computing.

Our experimental results show that the proposed architecture can improve inference latency by 18.3× over state-of-art multi-core CPU (Xeon E5), and 7.7× over server-class GPU (Titan Xp), for Inception v3 model. *Neural Cache* improves inference throughput by 12.4× over CPU (2.2× over GPU), while reducing power consumption by 50% over CPU (53% over GPU). We evaluate *Eidetic*

on Google's Neural Machine Translation System (GNMT) encoder and demonstrate a 17.20× increase in throughput and 7.77× reduction in average latency over a single TPUv2 chip.

The remainder of this dissertation is as follows. We discuss the background and related research work in Chapter 2. Following that, we describe our proposed research in transforming a cache into a neural network accelerator in Chapter 4 and our SRAM-based ASIC for neural network acceleration in Chapter 5. In Chapter 6 we discuss the two approaches of using cache vs a custom SRAM-based architecture for neural network acceleration. Finally, we conclude the dissertation in Chapter 7.

CHAPTER 2

Background

In this chapter, we will first give a background on the design and usage of SRAM in modern architectures. Then we will discuss how SRAM can be modified to support computation. Finally, we will give an overview of the types of DNNs, discussing their usages, characteristics, and features.

2.1 In-SRAM Computing

Random-access memory (RAM) is a category of memory that is extremely common in on-chip caches and off-chip main memory. RAM is a type of volatile memory that requires constant power to retain the stored data. The two main types are static (SRAM) and dynamic (DRAM). DRAM's memory cells typically consist of a capacitor and transistor and require frequent refreshes to retain the data/charge. The memory cell for SRAM is generally composed of 6 transistors (6T) and does not require a refresh. SRAM is less dense than DRAM, but is more energy efficient and provides for faster memory reads and writes. For instance, SRAM arrays are capable of operating at high frequencies such as 4.0GHz in Intel's Xeons. Most modern processors utilize SRAM for both register files and on-chip caches. Two sets of two transistors store the bit and inverted bit value within the memory cell. The other two transistors are used for access control. The SRAM memory cells have a set of complement bitlines, BL and BLB, and a wordline that is used for reading and writing to the cells. During the read access, the set of bitlines is pre-charged, and the wordline is activated. This connects the stored bit and the inverted bit to the pair of bitlines and will cause the voltage on one of the bitlines to drop. A sense amp reads the difference between these bitlines to

determine if a one or zero is stored. To write a value, the charge and the inverted charge are applied to the corresponding bitline and inverted bitline. The target wordline is also activated during the write. These components are shown in Figure 2.2.

SRAM cells are typically organized in a 2D array, with cells along a column sharing a bitline and cells across a row sharing a wordline to share the needed peripherals, decreasing the area overhead. This structure allows reading and writing to the SRAM array in parallel across a single row only. An SRAM array has a row decoder used to select a wordline. Typically multiple sets of bitlines will share a single sense amp. This reduces the parallel access to a row by the amount of sense amp sharing or column muxing. Modern processors store and compute data in a bit parallel format. Processors have a fixed data width of a "word" that contains some multiple of 8 bits. Data is read and written in the SRAM arrays in this granularity, with multiple words occupying a cache line.

This dissertation builds on previous work on SRAM bitline circuit technology [1, 3, 4] shown in Figure 2.2. To enable computation on data stored along a bitline, a second row decoder is added so that two wordlines can be activated simultaneously. Additionally, the differential sense amp is replaced with two single-ended sense amps. When multiple wordlines are activated, these two single-ended sense amps can be configured to detect if the activated cells along the bitline contain all ones or zeros. This effectively allows in-place computation of (`and` and `nor`) along a SRAM bitlines.

Data corruption due to multi-row access is prevented by lowering the wordline voltage to bias against the write of the SRAM array. Measurements across 20 fabricated 28 nm test chips (Figure 2.1) demonstrate that data corruption does not occur even when 64 wordlines are simultaneously activated during such an in-place computation, which is more than adequate as compute cache requires only two. Monte Carlo simulations also show a stability of more than six sigma robustness, which is considered the industry standard for robustness against process variations. The robustness comes at the cost of an increase in delay during compute operations. Conventional array read/write accesses remain unaffected. The increased delay is more than compensated by

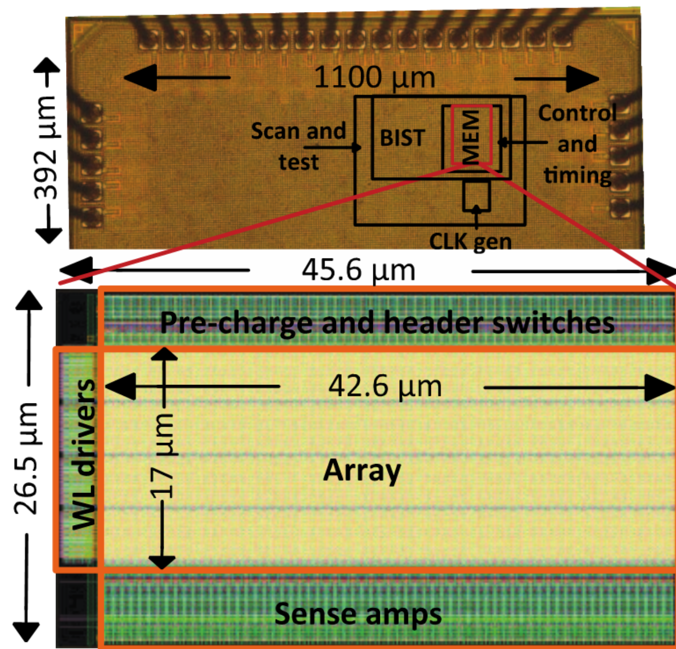

Figure 2.1: Prototype test chip [1].

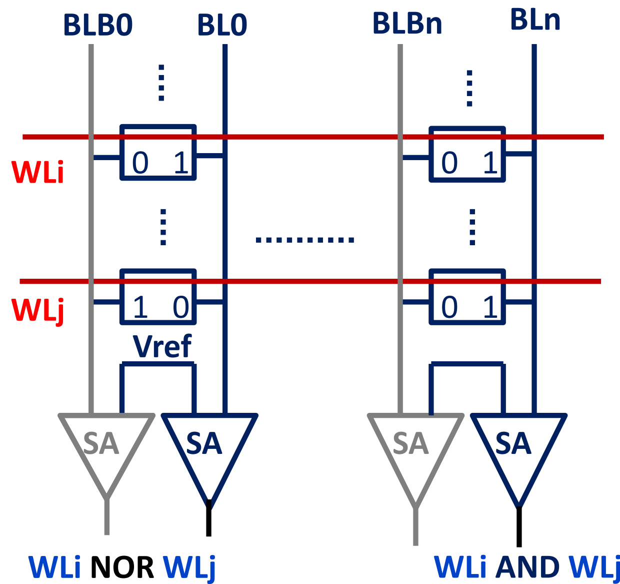

Figure 2.2: SRAM circuit for in-place operations. Two rows (WL_i and WL_j) are simultaneously activated. An *AND* operation is performed by sensing bit-line (BL). A *NOR* operation is performed by sensing bit-line complement (BLB).

massive parallelism exploited with in-SRAM computing.

Compute caches [5] further extends the in-SRAM bitline computing framework to support additional operations: `copy`, bulk zeroing, `xor`, equality comparison, and search. With this, they make a case for using a processor's cache for processing data-centric operations by addressing various architectural problems: operand locality, managing parallelism across various cache levels and banks, coherency, consistency, and reliability. Performing computation along the bitline with data stored in a bit-parallel format prevents more complicated operations, such as addition, because there is no way to move data across the bitlines without additional hardware. Addition would require a carry to be propagated across the bitlines.

Cache Automaton [6] introduced the concept of sense amp cycling. If an SRAM array has multiple columns that share a sense amplifier, sense amp cycling can be used to increase the throughput of in-SRAM compute. Most of the compute cycle is spent pre-charging the bitlines with a smaller amount of time devoted to sensing. Thus all bitlines can be precharged, and the sense amp can cycle through each shared bitline without the pre-charging overhead.

2.2 Deep Neural Networks

The field of Deep Neural Networks (DNNs) and Machine Learning (ML) was reinvigorated by Alexnet in 2012. Alexnet achieved significant improvements at ImageNet 2012. Alexnet showed the importance of using multiple layers in a network and GPUs' potential to make training these deep neural networks feasible. DNNs are a key component in the field of Machine learning with applications in bioinformatics, speech recognition, autonomous vehicles, economics, fraud detection speech translation, and more.

Artificial Neural Networks (ANNs) are attempts to mimic biological neural networks by creating a layer of nodes with a weight or learned value that interacts with inputs to produce an output. Deep neural networks are ANNs with multiple layers between the input and output layers. Neurons can be implemented in a variety of different ways, but typically invoke a matrix operation such

as matrix-matrix multiplication, vector-matrix multiplication, and convolution. Common types of DNNs are multi-layer perceptrons (MLPs), Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and Transformers. While CNNs utilize convolutions, the convolution is often converted into matrix-matrix multiplication. The nodes in RNN, MLP, and Transformer all heavily utilize matrix multiplication. RNNs introduce a recurrent data dependency and transformers invoke a self-attention mechanism. The two most important operations in a neural network are convolution and matrix multiplication. However, since a convolution can be converted to matrix multiplication, accelerators can be optimized strictly for matrix multiplication operations.

DNNs have two important stages; training and inference. Training uses a set of test data to determine optimal weight values for all the nodes or neurons in the network. The network will be trained for specific applications or tasks. Training data consists of input data and the desired output/classification for each input. Additionally, a separate set of test data is used to evaluate the accuracy of the output without the network being specifically trained on it. Most commonly an algorithm called gradient descent is used to update the weights based on the error from the expected outputs and the realized outputs. Inference is running the network with inputs after the network has been trained for the task.

MLPs are usually defined as a relatively generic feed-forward ANN. A basic MLPs layer consists of matrix multiplication usually followed by a nonlinear activation such as sigmoid or tanh.

CNNs as their name suggests contains convolutional layers. In addition to convolutional layers, CNNs will typically have pooling and fully connected layers mixed in as well. Rectified Linear Unit (ReLU) is typically used as the activation function for each layer. Most CNNs spend the majority of the compute during the convolution cycles. A convolution layer uses the weights as a filter and applies the filter to the input image. Filters have four dimensions, height (R), width (S), channels (C), and batches of 3D filters (M). Inputs have three dimensions with a height (H), width (W), and channel (C). The filter is overlaid on the inputs, and each pixel of the input is multiplied by the corresponding filter pixel and repeated across the M dimension. The results across the $R \times S$, i.e. the height and width, are accumulated together. Further, the channels are also reduced

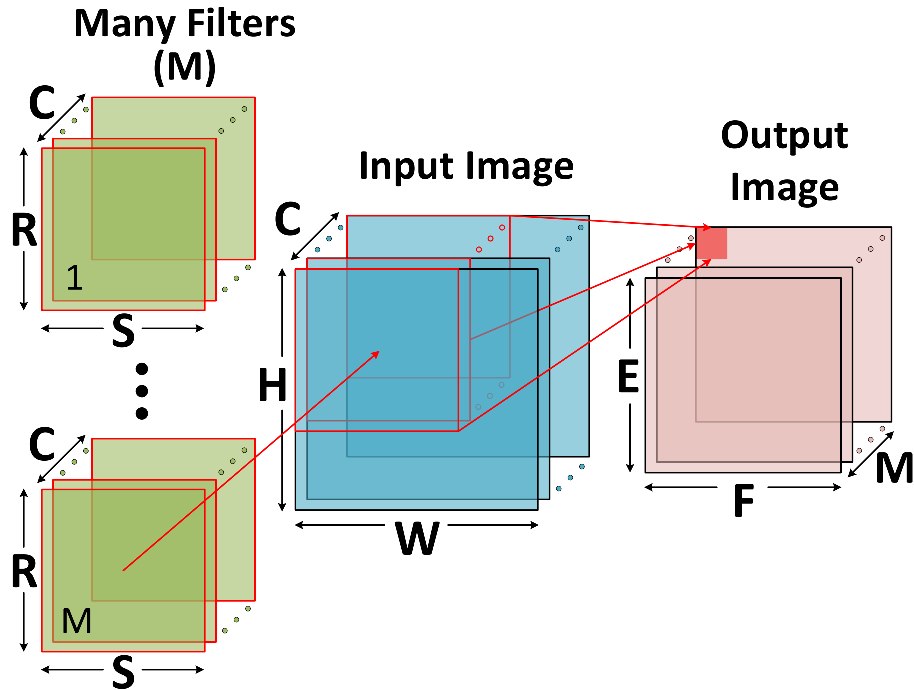

Figure 2.3: Computation of a convolution layer.

into a single element. Thus each window gives us an output of $1 \times 1 \times M$. The filter window is slid across the input to produce multiple output activations. The window is slid using a stride (U), increasing the stride will decrease the number of output elements computed. The output's channel size is equivalent to the M dimension of the filter. The output image, after an activation function that varies across networks and layers, is fed as the input to the next layer. CNNs vary greatly in size and number of layers. AlexNet, for instance, has 5 convolutional layers and 3 fully connected layers whereas Inception v3 has 94 convolutional 'sub layers'. The convolutional layers at the start of a network tend to have an input with a large height and width, whereas the channel size and depth of the filters are small. The layers at the end feature inputs with smaller heights and widths but have a much larger depth and number of channels. This results in the first layers having smaller weight matrices but a larger amount of compute, and the later layers having larger weight matrices but a smaller amount of compute.

RNNs introduce a recurrent element through a temporal dependency. The output of a node is not only passed to the next layer but is reused in the same node for a future input. This is extremely

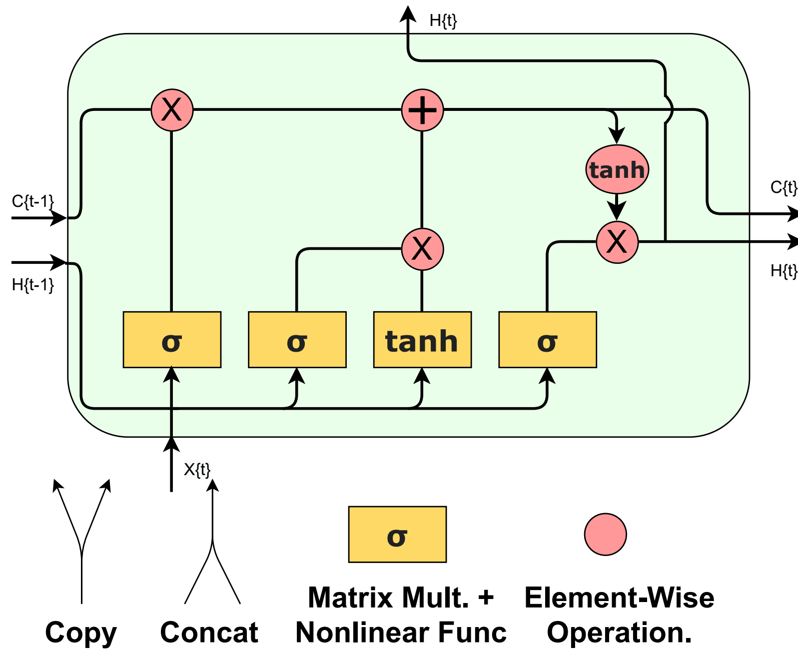

Figure 2.4: Typical LSTM Cell showing micro-operations performed

useful for speech translation, where words in a sentence are separate components of the inputs and the meaning depends on past and future words. Incorporating the previous word's output into the input of the node allows past information to be retained. Additionally, a network can have a bidirectional layer where the input is reversed allowing for translation to be based on the past and future words in the sentence.

Long short-term memory (LSTM) is a popular subcategory of RNNs. An LSTM layer is typically referred to as an LSTM cell. Each cell takes in and concatenates an input from a previous layer and its own output from a past or future time step. Figure 2.4 shows an LSTM cell that consists of various micro-operations. The yellow box represents a matrix multiplication operation (`matmul`) with hidden weights, followed by a non-linear activation function. The outputs of these non-linear functions are then involved in a variety of element-wise operations. The outputs of the cell can become inputs to the next cells and the same cell at a different time step. GNMT is a widely used LSTM, a subclass of RNN, that features a bidirectional layer. Figure 2.5 shows how the different LSTM cells connect across layers and time steps in the GNMT encoder.

Matrix-Matrix multiplication can be described with an input matrix of [n x k] and a weight

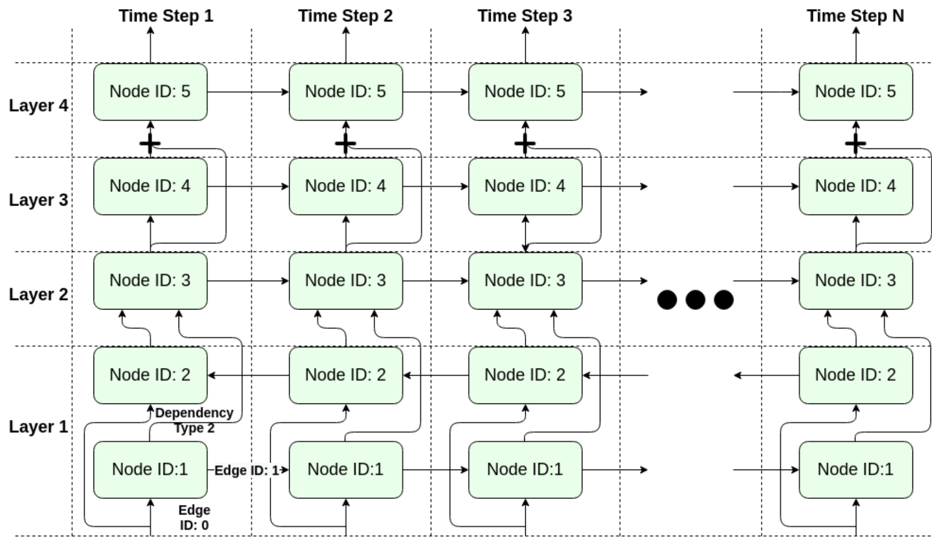

Figure 2.5: GNMT Encoder Layout

matrix of [k x m]. Note that the number of columns in the input matrix and the number of rows in the weight matrix match. A convolution can be converted into a matrix multiplication by performing an Im2col transform on the input image. This creates an input matrix of dimensions [E*F x R*S*C] with a weight matrix of dimensions [R*S*C x M]. A column of elements from the input matrix is multiplied by the corresponding elements of a row of the weight matrix with the products being accumulated together. This multiplication and accumulation happen for each input row and weight column pair. Thus the output matrix will be of dimension [n x m]. The number of multiply and accumulations needed will be n*k*m.

DNNs have a wide range of network and layer sizes based on their application, power, latency, and throughput budget. A larger network tends to have better accuracy but also increases the amount of compute and energy needed. For instance, EfficientNet is an image classification network designed for edge computing with 5.3M parameters. Bert Large, on the other hand, is used in data centers for Natural Language Processing (NLP) and contains 345M parameters.

2.3 Accelerating Neural Networks

Matrix multiplication is a highly parallelizable operation, as every multiplication is independent of each other and thus can be parallelized. However, the weight matrices in networks are generally too large to parallelize every multiplication. Additionally, since the products produced need to be accumulated with other products from the same row in the weight matrix, it is common for DNN accelerator architectures to design PEs capable of a single fused multiply and accumulate (MAC). Products that are accumulated together can be created sequentially in the MAC units, eliminating the need for a separate accumulation.

Matrix multiplication acceleration can suffer from being compute-bound or data-bound, and sometimes both across multiple layers. Compute-bound refers to when computation is limited by the number of compute units. If a bottleneck is created because there are more available operations to perform than compute units, the application is said to be compute-bound. Data-bound can refer to when there are available compute units, but not enough operations are available to execute due to lack of data. The lack of data can come from on-chip data movement and off-chip data movement as well as from data dependencies. TPUv1 detailed the breakdown of compute cycles and data stalls for a pair of each CNNs, MLPs, and LSTMs. The compute units in MLPs and LSTMs were only active 8.2% to 12.7% of the execution time. Instead loading weights from memory and moving weights on-chip resulted in stalls for between 57.6% and 79.2% of the cycles for the 4 MLPs and LSTMs. The two CNNs had the compute units active 78.2% and 46.2% of the cycles, with weight stalls and shifts between 0.0% to 35.1%. These number help show the extreme range that DNNs can have for being compute-bound or data-bound.

Critical factors for accelerating DNNs consist of supporting optimal precision(s), optimizing data reuse, facilitating optimal dataflow, and having a large amount of compute units available. Additionally, these factors affect throughput, latency, and power budgets which need to be balanced. The type of precision(s) supported affects not only the accuracy of the network but the ability to do training, the compute units' speed, the power of the compute units, the area of the compute units as well as the available software support. Data reuse is also a key factor, as every off-chip

memory access and on-chip storage access adds an energy and delay penalty. Efficient caching can prevent multiple off-chip memory access for the same data. Further data can be reused in registers. Both reducing off-chip and on-chip accesses are achieved by creating an architecture and data mapping designed for the parallel computation and data reuse of neural networks. Facilitating an efficient dataflow can be key to making reuse available without stalls. Accumulations, dependencies between layers, and replications of weights or inputs for parallelization all have dataflow considerations. Throughput and latency will often have an inverse relationship when inputs are batched together to increase the reuse of the weights.

DNNs are most commonly designed using single-precision floating-point format (FP32) data format. FP32 is used to represent a dynamic range of decimal numbers. IEEE FP32 contains a sign bit, 8 exponent bits, and 23 fraction or mantissa bits. With this dynamic range, FP32 can represent both $1.4012984643 \times 10^{-45}$ and $3.4028234664 \times 10^{38}$. Floating point is used for training neural networks due to its increased flexibility in range and precision. Training with floating point results in significantly better accuracy than with integers.

Other reduced precision floating point formats can also be used. IEEE half-precision floating-point format (FP16) uses 5 exponent bits and 10 mantissa bits. Bfloat16 instead uses a truncated version of IEEE FP32, maintaining 8 bits for the exponent but reducing the mantissa bits to 7. Nvidia's TensorFloat uses a 19-bit format, 8 for the exponent and 10 for the mantissa. AMD's fp24 uses 7 exponent bits and 16 mantissa bits. Note that all of these formats have a single bit for the sign.

Additionally, networks can be quantized to an integer format for inference. Oftentimes the activations and weights are converted to int8 but extremely low precisions with binary quantization are also used. The range of values for activations and weights can be calculated. The range can be used to create a scale and offset value that is used to convert the floating point values to integers. After a layers computation, the output is converted back to floating point using the scale. Networks can be trained in floating point and run in either floating point or quantized integer. Networks can also specifically be trained for quantization, reducing the accuracy loss. Training in integer is

very uncommon, using floating-point during training is more common even when inference uses quantization.

Using reduced format precisions data format can reduce the amount of storage and memory bandwidth needed. Reducing the precisions can also result in significantly faster compute times. However, this requires ALU units and instructions capable of running reduced precisions operations efficiently.

Modern architectures provide support for both integer and floating point operations. But won't necessarily support the reduced precisions effectively as a custom architecture. For instance, quantized inference for matrix multiplication might want to support 8-bit inputs and weights with the result of the multiplication stored at 16 bits. Further accumulation can be supported with 16, 24, or 32-bits. Integer ALUs occupy significantly less area and are faster than floating point ALUs. However, designing an architecture that only uses integer ALUs results in some accuracy loss which can limit the networks and applications. Additionally, it prevents using the custom architecture for training.

A challenging aspect of accelerating DNNs is optimizing for data reuse. Given an input and weight matrix of [n x k] and [k x m] respectively, the reuse factor, or the number of times a weight element is accessed is n. If the value of n is 1, the input is a vector instead of a matrix. It is common for RNNs, MLPs, and fully connected layers in CNNs to have an input vector instead of an input matrix. In the case of RNNs, the input sequence is a matrix of inputs that are separated into vectors by time. Because of a temporal dependency, these vectors cannot be combined into a matrix for greater reuse. However, multiple independent inputs can be combined into a batch. Batching of inputs allows the input vectors to be combined into a matrix, as well as input matrices concatenated together. Batching is done to increase the reuse of the weights. Batching inputs can have huge gains for throughput as it amortizes costly weight loading, but comes at a direct cost to latency. Inputs will have to wait for the other inputs in the batch to finish before the next layer can start executing resulting in increased latency. Additionally, batching also requires that there are available inputs to be batched at the time of inference. Each input will further increase the amount

of on-chip memory needed as each input will create its own intermediates that need to be stored.

In addition to weight reuse, the networks also have reuse with the inputs. With an input and weight matrix of [n x k] and [k x m], each input matrix or vector is applied to each of the m rows of the weight matrix. If the m rows of the weight matrix are separated across multiple compute units, the input would have to be either replicated or passed from one compute unit to another.

Finally, products reuse exists during the accumulation of the matrix multiplication. The output of each MAC is a product that needs to be accumulated with other MACs from the same column in the weight matrix. The product can be reused in the compute unit if a new input and weight are loaded in. The products can also be passed to other compute units where they will be accumulated with another product in its column. Or, the products can all be collected and accumulated outside the matrix compute units.

There are multiple approaches to designing an architecture for accelerating DNNs. One approach is using many parallel compute units for single instruction multiple data (SIMD) processing. The computation is parallelized across the compute units with all compute units acting in lockstep. The outputs from these units would have to be collected and accumulated together. In chapter 4 we present *Neural Cache* which uses a SIMD approach.

Another approach involves using a dataflow architecture such as a systolic array. With a systolic array, one of the 3 operands is kept stationary, with the other two passed to neighboring PE units. This style of architecture is explored in chapter 5 where we present *Eidetic* which uses a dataflow architecture.

CHAPTER 3

Prior Works

In this chapter, we will cover both industry and academic work for designing DNN accelerators. Additionally, we will discuss the prior work on in-memory computing that has led to in-SRAM computing as well as other in-memory DNN accelerators.

3.1 DNN Accelerators

Many industry giants, promising startups, and academic researchers have proposed their own architecture for DNN/ML acceleration. A common approach is to have a small number of cores per chip, with each core having a large or several large matrix compute units. These architectures are often dataflow architectures, with emphasis on data movement between PEs, with a systolic array being a very common option. Another option for ML accelerator architecture is a many-cores solution. GPU for instance is a specific type of many-core architecture. A many-core architecture can improve the granularity compared to a dense matrix multiplication unit. The mapping of operations is not as rigid as these large matrix multiplications units and will have a smaller array of compute units. This can improve efficiency for mapping and leveraging sparsity. A core can be turned off or mapped with other operations. Utilizing unoccupied PEs in a large dense compute array is a harder task. Additionally, each core can have more control over how data is passed and shared. This finer granularity comes at a cost in efficiency. Each core will have added overhead with its control logic. Simple data movement from PE to PE is less expensive in the area, energy, and time compared to moving data from one core to another. Often times the dataflow architecture

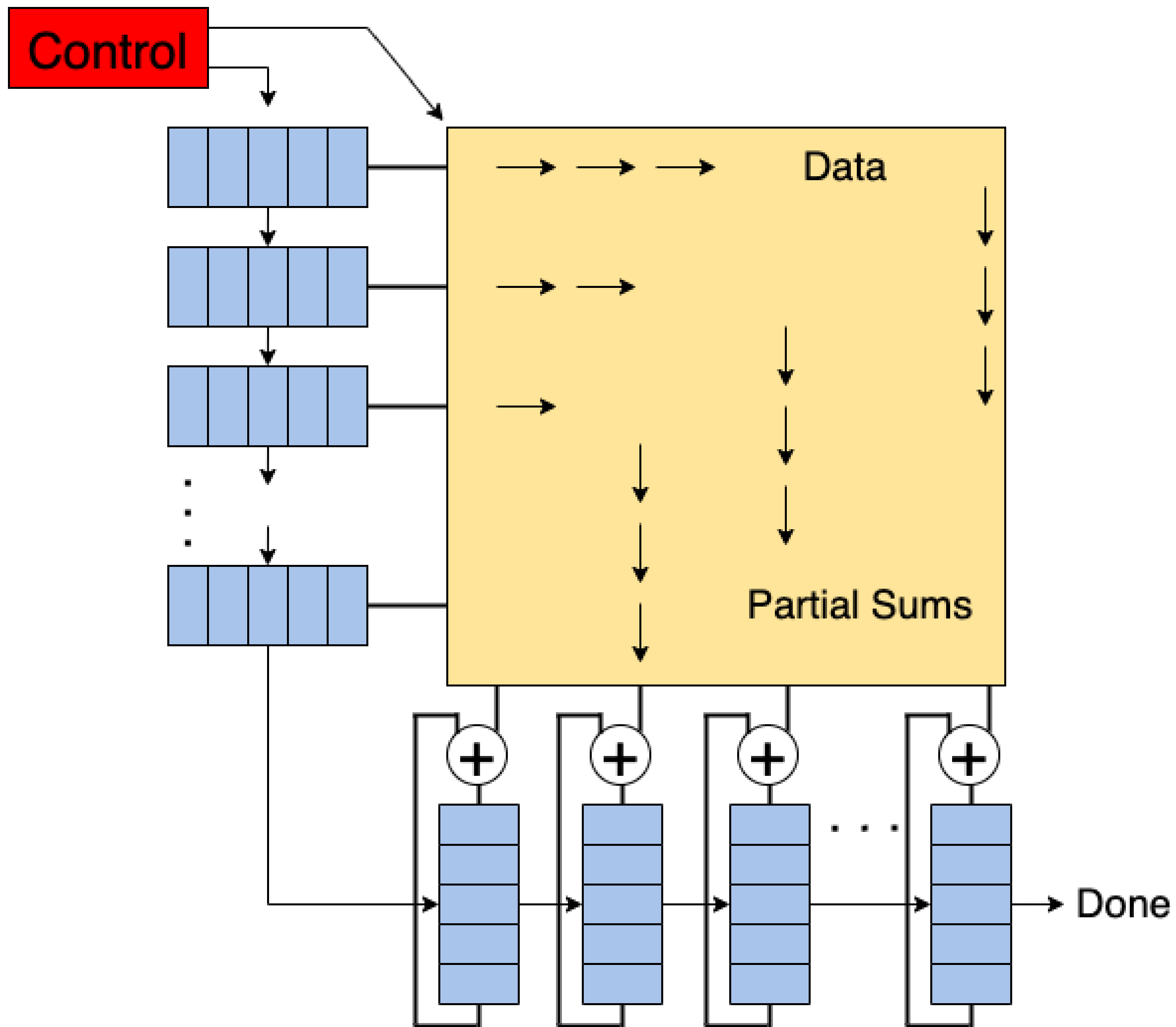

Figure 3.1: Google TPU with its 2D Systolic Array

will have a few complex instructions for the matrix operations. These instructions and the matrices will need to be broken up into many parallel instructions increasing the instruction overhead and the amount of software support needed for a many-core architecture. The work in *Eidetic* supports many large matrix multiplication units due to the density of in-SRAM computing. The 54 systolic arrays on-chip may look similar to a many-core architecture, however, the dataflow and programming between the systolic arrays is more similar to the PE dataflow architecture. Instead of a many-core architecture *Eidetic* is two-level dataflow architecture, connecting PEs together to compute a layer, and systolic arrays together to compute the network.

TPUv1 [7] is a neural network accelerator that Google introduced in 2016 and published the

details of in 2017. The first version of TPU has a matrix multiplication unit that was a 256 x 256 systolic array of PEs. Figure 3.1 shows the TPU systolic array. Each PE supported both 8-bit and 16-bit integer for each input operand. An 8-bit and 16-bit operand would cause the PEs to operate at half speed. Two 16-bit operands would cause them to operate at a quarter speed. Like *Neural Cache* and *Eidetic*, TPU supports a weight stationary mapping. Weight tiles are loaded into the systolic array, inputs are propagated across the systolic array and products are passed downwards. At the bottom of each column in the systolic array is an accumulator to hold the products and accumulate them with the products from the next weight tile. An activation pipeline takes the accumulated output and performs the needed activation and passes its output to the local buffer. TPUv1 devotes a significant 24 MB and 29% of its chip area to its local unified buffer. This allows TPU to support a large batch size to increase the utilization of the systolic array. Additionally, training was not supported in this version.

TPUv2 [8] was first deployed in 2017 and no longer supported integer arithmetic. Instead, it supports its own custom Bfloat16 data format. Additionally, it supported FP32 but resulted in an $8\times$ performance loss from Bfloat16. With a floating point data format, this TPU was able to support inference and training. TPUv2 additionally has two identical cores per chip, with each core featuring a 128 x 128 systolic array. Despite having half the matrix compute units and an improved technology node, the TPUv2 is almost twice the size of TPUv1. This can be explained by the floating point units occupying a significantly larger amount of area than their integer counterparts. TPUv3 introduced a second matrix multiplication unit per core while keeping the same systolic array structure. TPUv2 and v3 feature 32MB of on-chip SRAM.

Amazon Inferentia chips [9] is another example of industry utilizing systolic arrays as their matrix multiplication unit. Each chip consists of 4 Neuron Cores with a systolic array that supports BF16, INT8, and FP16 data types. Each chip has a large on-chip cache and a commodity DRAM.

Tesla's Full Self-Driving Chip features two Neural Processing Units, a light GPU for post-processing, and 12 Cortex-A72 cores for general-purpose processing. Each neural processing unit has a 96x96 multiply-accumulate array for performing MACs. Unlike some other accelerators,

these are not systolic arrays. Additionally, each neural processing unit has 32MB of SRAM for a total of 64 MB on-chip. The SRAM is used to store intermediates and reduce the main memory accesses.

Another prominent DNN accelerator is Project Brainwave [10] from Microsoft. Brainwave is a "soft processor" that is synthesized onto its cloud FPGAs. Like *Eidetic* brainwave seeks to process AI, specifically RNNs, in real-time. Brainwave processes a single input and pins the weights in the distributed on-chip SRAM. However, unlike *Eidetic*, Brainwave's weights are not computed on in-place, requiring the weights to be moved to the compute units for each input. Brainwave relies on RNNs having multiple elements in the input to achieve a high utilization. Additionally, Brainwave uses a block floating point format that shares 5-bit exponents across a group of numbers.

Nvidia introduced Tensor Cores [11] in its Volta GPU micro-architecture. Tensor Core performs matrix-multiply-and-accumulate on 4×4 matrix each cycle. The Tensor Core takes in half-precision input operands and performs the multiplication in half-precision but the accumulation in single precision. Each streaming multiprocessor is partitioned into four processing blocks, each with two Tensor Cores, 8 FP64 cores, 16 FP32 cores, 16 INT32 cores, and one Special Function Unit. The second generation [12] added support for Int8, Int4, and Int1. The third generation included Bfloat16, TF32, and FP64. The fourth generation adds support for FP8. The GPU architecture is designed to support many parallel threads. Each processing block has a large register file to support temporal and spatial locality. The Volta architecture contains 36MB of on-chip memory across the 80 streaming multiprocessors.

Tenstorrent Grayskull [13] architecture is another many-core example with their architecture containing 120 cores in a 12x10 grid. Each core features a small 1MB SRAM, a compute engine, and 5 scalar RISC CPUs. The compute engines are optimized for INT8 but can support FP16 and Bfloat16 as well as a custom FP8 with the same throughput of INT8. Additionally, the chip has 4 super-scalar CPUs to manage the data flow.

Grog's Think Fast TSP [14] presents another take on the many-core architecture. Each tile in the network is designed to implement a specific function. A slice is composed of 20 of these tiles

stacked vertically. All the tiles in a slice use the same instruction stream for SIMD and pipelined execution. Think Fast TSP presents results for ResNet50 with a batch size of one.

In addition to industry, many academic works have proposed various DNN accelerators. Many of these works [15, 16, 17, 18, 19, 20] focus solely on CNN acceleration. Using systolic arrays as the computing engine is also an extremely common theme [15, 21, 22]. Approximation can be used to improve the compute performance and reduce energy consumption while not significantly reducing accuracy. This is done through dynamic quantization and bit-widths [21, 23, 24] and using approximate memory [25]. Leveraging sparsity in DNNs can improve compute performance and save significant amounts of energy. Architectures and mappings to leverage sparsity have also been proposed [15, 19, 26].

Eyeriss [20] was a trend-setting DNN accelerator published and tapped out back in 2016. This accelerator focuses on CNN acceleration, processing a CNN layer-by-layer. The core computing unit employs a spatial array of 168 PEs organized as a 12 x 14 rectangle. The PEs are capable of doing 16-bit fixed-point arithmetic. The convolution is broken into multiple 2D convolutions for inference. They present a row-stationary dataflow to optimize energy efficiency for any convolutional layer. The row stationary dataflow involves passing all three operands to adjacent PEs. Filters are passed horizontally, inputs are passed diagonally, and partial products are passed vertically. *Neural Cache* also uses a layer-by-layer approach and employs a hybrid of weight stationary and output stationary reuse pattern. *Eidetic* uses a weight stationary design and does not follow a layer-by-layer approach.

Our work in *Neural Cache* and *Eidetic* leverage bit-serial computation to exploit parallelism at the level of numerical representation. Stripes [27] leverages bit-serial computation for inner product calculation to accelerate DNNs. Its execution time scales proportionally with the bit length, and thus enables a direct trade-off between precision and speed. Our work in *Neural Cache* differs from Stripe in that *Neural Cache* performs in-situ computation in the SRAM arrays, while Stripe requires arithmetic functional units and dedicated eDRAM. DaDianNao [28] is an architecture for DNNs which integrates filters into on-chip eDRAM. It relieves the limited memory

bandwidth problem by minimizing external communication and doing near-memory computation. Bitfusion [29] introduces bit-level fusion and decomposition among the PEs. The BitBricks PEs can be decomposed to support lower bit-level arithmetic for bit-level granularity for each layer.

3.2 Processing in Memory, In-Memory, and Near-Memory Computing

Processing-in-memory (PIM) [30, 31, 32, 33, 34] was a popular concept in the 90s that had fallen out of favor, but has seen a resurgence with in-memory and near-memory computing. These works moved the compute units *near* main memory (DRAM), thereby reducing the gap between memory and compute units. Today this method of computing is generally referred to as near-memory computing as the computation is performed outside of the actual memory units. This distinction is important because there has also been a rise in in-memory computing, where the compute units exist as part of the memory/storage units. These PIM architectures of the 90s were mainly DRAM and faced challenges integrating the logic inside the DRAM cells or even on the same die. Since then the memory wall problem has accelerated and exasperated the need to improve the memory bandwidth of the Von Neumann architecture. This increased need coupled with the newly available 3D stacked memory has led to a resurgence in moving compute closer to memory. The work presented in this dissertation is an example of in-memory computing as the computations are performed inside the SRAM arrays. This is an example of in-SRAM computing. Processing in/near memory can be done with a variety of storage/memory types. In addition to DRAM and SRAM, other memory technologies such as flash and ReRAM also exist.

Memristors/ReRAM-based architectures were some of the first in-memory computing that was proposed in this current iteration. ReRAM is an extremely dense non-volatile memory that has been looked at to replace Flash memory. ReRAM is often referred to as memristors. Memristors can be programmed with up to 32 different resistance values that remain even without access to power. Applying an input voltage to the memristor creates a resulting current that is the multipli-

cation of the programmed weight and input voltage. When two currents are joined, the resulting current is the sum of the two. This allows multiple rows of memristor cells to be activated with the results accumulated along the shared bitline. Built-in accumulation and ReRAM density make it an attractive option, however, ReRAM faces several challenges that limit ReRAM potential. ReRAM is highly susceptible to process variation. The multiply relies on a linear current-voltage curve for the resistance. Process variation can break this curve causing inaccuracies or forcing the memristor to store fewer bits per cell. Compared to volatile memories, ReRAM has a lower cell life, as well as a higher latency and energy cost to program due to a higher voltage needed. Finally, the representation is limited because negative values cannot be stored in the cells and also because of the limited range of programmed resistance and scalability of ADC and DAC logic. The area and energy overhead of an ADC increases exponentially with the resolution.

PRIME [35] presents a memristor design for accelerating MLPs and CNNs. PRIME handles the bit-precision challenge by using 8-bit weights and 6-bit inputs. The weight and inputs are further split in half across four cells. The four outputs are decoded by a smaller ADC and are pierced together with peripheral adders and shifters. ISAAC [36] is another CNNs memristor accelerator. ISAAC presents a different approach to precision. Weights are still split across the cells in a row, but the input is fed sequentially, one bit at a time. This reduces the DAC and ADC precision needed. ADC peripheral logic is needed to collect each output over the 16 cycles. This allows ISAAC to use 16-bit operands. Additionally, ISAAC creates a pipeline for the CNN layers. The density of the memristors allows it to pin multiple layers in the network and separate the layers into different tiles. The output of one tile is sent to the input of the next, creating the pipeline. ISAAC can replicate the weights of a slow layer to spare tiles to improve the compute performance. FPSA [37] is a full stack solution that proposes reconfigurable routing architecture to break the communication bottleneck. Additionally, they support a spiking schema to improve PE density and efficiency. FloatPIM [38] introduces support for floating point and supports all operations using a NOR operation. The NOR operation is performed only on single-bit operands eliminating the overhead of the costly ADC and DAC. The cycles for FP multiplication are based

on the number of mantissa and exponent bits. Single-bit memristors also increase the feasibility from a manufacturing standpoint. Like the work in *Eidetic* and *Neural Cache*, floatPIM operates on only one bit of the operands per cycle. PipeLayer [39] creates a pipelined ReRam architecture for training CNNs. Training for CNNs increases the complexity of dependencies but does not include support for temporal dependencies and on-demand processing.

In-SRAM/Near-SRAM computing has been well explored in the last several years. A majority of the works listed are designed for neural network acceleration. There are many ways that these works differentiate themselves from each other. Some of the novelties in these in-SRAM computing works involve using analog/mixed sensing, using alternative in-SRAM computing methods, and modifying the SRAM bit cells and peripherals. Additionally, the cells or the hardware and software can be designed to support weight sparsity and processing can be done near-SRAM instead of in-SRAM.

Our work proposes bit-serial arithmetic in the digital domain. From our differential sense-amps, we can calculate if the activated cells are all ones (AND) or all zeros (NOR). Another approach is to use ADC and perform a popcount. These approaches often are used for low precision or binary networks as they do not scale well for higher precision computation and require more energy than a sense amp. Nand-net [40] proposes compressing the popcount by converting XNORs to NANDs which decreases the magnitude of the output allowing the ADC resolution to be considerably reduced. Using a mixed signal [41] has also been proposed to increase the scalability and efficiency of in-memory. Computations that have a high signal-to-noise ratio can be performed in the analog domain. Promise [42] proposes a compiler where the precision of the operations is adjustable based on a target accuracy for an algorithm. The mixed signal approach allows them to reduce accuracy to save energy.

Instead of performing the computation in the SRAM peripherals like the work in this dissertation, it's possible to design arrays that integrate a local compute cell among multiple SRAM cells. Several works have proposed [43, 44] bitwise multiply-units that consist of 16 6T SRAM cells and 1 local computing cell. Another proposes [45] a compute engine 32 6T SRAM cell with a local

computing unit as well as segmented bitlines. These local compute units enable support for more compute operations.

Other work has also explored modifying the SRAM bit-cells for in-SRAM computing. Our work utilizes standard SRAM 6T cells for in-memory computation. Multiple works [46, 47, 48] have proposed modifying the SRAM cells and using 8T cells instead of 6T. One version of the 8T-SRAM allows zero-skipping to support weight sparsity [48]. The work proposed in *Eidetic* supports input sparsity but not weight sparsity. Other versions [46, 47] propose versions of the 8T SRAM to improve stability when increasing the number of activated cells. However, increasing the number of transistors in each cell will increase the area and power usage of the SRAM. Our work does utilize custom 8T SRAM cells, but they are used for performing transpose rather than performing computation.

Other modifications have been proposed to the peripheral logic of the SRAM arrays to improve the efficiency of the computation. Su et. al. [45] propose a design segmenting the bit-lines in the array. By segmenting the bit-lines into smaller groups, the SRAM read energy can be reduced.

Bit Prudent explored leveraging structured sparsity of Neural Networks with in-SRAM computing. DNNs were specifically pruned to coalesce non-zero filters enabling efficient dense computation. Previously mentioned work [48] supported sparsity with their proposed 8T SRAM cell.

Just as there is in-memory and near-memory computing, there is also in-SRAM and near-SRAM computing. All near-SRAM computing differs from the work in this dissertation by performing the computation outside the SRAM arrays and their peripherals. Near-SRAM computing was proposed as far back as the 1990s with Terasys [30]. Terasys presents a bit-serial arithmetic PIM architecture. Terasys reads the data out and performs the computation in bit-serial ALUs outside the array. Further Terasys performs software transposes while *Neural Cache* and *Eidetic* have a dedicated hardware transpose unit, the TMU. More modern examples of near-SRAM computing have specifically been designed for neural network acceleration. Multiple works have proposed using the SRAM arrays as look-up tables [49, 50]. The logic to support the look-ups exists outside the SRAM arrays. Another approach is Reduct [51], which adds tensor functional units outside

each level of the cache. Like *Neural Cache* these works are designed to augment a cache in an existing processor to enable support for parallel computation.

In-DRAM computing is an attractive solution but suffers from several key challenges. The process for manufacturing DRAM is not designed for logic and creates inefficiencies in the logic circuits. The process is not optimized for logic speed and additionally, the in-DRAM logic can incur a 2-3$\times$ area overhead [52, 53] due to the in-DRAM logic only having access to two or three of the metal layers. Another challenge is that the DRAM writes are destructive. Performing in-place computation will corrupt the input operand data. Solutions which copy data and compute on them are possible [54], but slow. Reusing the input operands is very common in neural networks and is heavily utilized in our in-SRAM designs. Finally, data scrambling, and address scrambling in commodity DRAMs [55, 56, 57, 58, 59] contribute to making re-purposing commodity DRAM for computation challenging. One example of in-DRAM is DRISA: A DRAM-based Reconfigurable In-Situ Accelerator [52]. DRISA offers computation both by adding external logic peripherals to the cell as well as modifying the DRAM cell for computation. DRISA optimizes its architecture for CNN acceleration. PIM processing near-DRAM of the 90s was not widely adopted due to inefficiencies with logic and memory on the same die. However, recent advancements in 3D stacked DRAM have led to their resurgence. Multiple separate 2D DRAM dies can be stacked on top of each other with communication along the stack. Since they are separate dies, near-memory logic can exist on its own die where the process is properly optimized. There are several examples of near-DRAM computing. Neurocube[60] creates a compute engine layer that is broken across different tiles that have access to different vaults of a Hybrid Memory Cube (HMC) DRAM. Tetris [61] also utilizes HMC DRAM and creates a NN engine in a vault in a tile. Each engine has 100s of PEs organized in a 2D array. The array of PEs leverage a row-stationary dataflow to optimize for data movement and reuse. NAND-net [40] mentioned for mixed-signal SRAM also supports near-DRAM computing. All of the above near-DRAM accelerators optimize for CNN acceleration and perform layer-by-layer computation. TransPIM [62] uses a token-based dataflow, where computations across multiple layers of the same token are kept in the same memory location to reduce

movement. This design is implemented in high bandwidth memory, a type of 3D stacked DRAM. TransPIM's design is optimized for transformers. TensorDIMM [63] uses near-DRAM computing with a custom DIMM module. This module is integrated into a GPU systems interconnect and targets sparse embedding operations.

In addition to in-memory in these established memory technologies, in-memory computing for neural network acceleration has been shown in emerging memory technologies such as SST-MRAM, Phase Change Memory, and Flash Memory.

NAND Flash memory is a well-established NVM memory technology that offers non-volatile and high-density storage. Flash memory has been widely adapted in Solid State Drives (SDD), which have been overtaking Hard Disk Drives (HDD) in recent years. Flash memory additionally supports multicell storage of 4 bits or more. Despite its widespread use, flash offers unique challenges due to its long read time, and even longer write time. Both in and near flash computing have been proposed to accelerate neural networks. One example of near-flash is RM-SSD [64]. RM-SSD uses an SSD with an FPGA-based in-storage computing engine. RM-SSD pipelines the top and bottom MLPs in a recommendation system. The MLPs do not have the increased complexity that RNNs have with temporal dependencies. Mythic AI [65] has developed an in-NAND Flash neural network accelerator. Each flash cell is capable of storing an 8-bit weight based on the capacitance of the cell. Much like in Memristors, a voltage is applied to the cell and the produced current is analog to the multiplication of the voltage and capacitance.

Spin-Transfer Torque Magnetoresistive RAM (STT-MRAM) is another non-volatile memory looking to replace DRAM. It has a faster and more energy-efficient write coupled with a longer lifespan than other NVM technologies. Some previous work has focused primarily on bitwise binary operations [66]. Jain [67] extends support for addition by adding peripherals logic gates to support sum and carry. This is a very similar approach to how our work in *Neural Cache* supports in-SRAM arithmetic. Multi-level cell in-memory computing for STT–MRAM has been shown possible [68]. This works leverages its in-memory computing to accelerate binary convolutional neural networks.

Phase Change Memory (PCM) uses glass that alternates between an amorphous and crystallization structure based on the applied current. Additionally, it can support multi-level cells and has a similar in-memory computing approach to ReRAM. Multiple works [69, 70] have explored using in-memory PCM for neural network acceleration. A mixed precision architecture [71] using PCM has also been proposed. PCM writes take longer and requires significantly more energy than other NVM limiting its potential.

CHAPTER 4

Neural Cache

The amount of on-chip memories, such as caches, has grown significantly in the last decade. This growth, coupled with the rise in popularity of data-parallel neural networks, creates a unique opportunity to leverage these trends by repurposing cache to become a data-parallel DNN accelerator. In this chapter, we present *Neural Cache*, an advancement in in-SRAM computing that allows for an existing cache to be repurposed into a data-parallel DNN accelerator.

4.1 Introduction

In the last two decades, the number of processor cores per chip has steadily increased while memory latency has remained relatively constant. This has led to the so-called memory wall [72] where memory bandwidth and memory energy have come to dominate computation bandwidth and energy. With the advent of data-intensive systems, this problem is further exacerbated and as a result, today a large fraction of energy is spent in moving data back and forth between memory and compute units. At the same time, neural computing and other data-intensive computing applications have emerged as increasingly popular applications domains, exposing much higher levels of data parallelism. In this dissertation, we exploit both these synergistic trends by *opportunistically* leveraging the huge caches present in modern processors to perform massively parallel processing for neural computing.

Traditionally, researchers have attempted to address the memory wall by building a deep memory hierarchy. Another solution is to move compute closer to memory, which is often referred to

as processing-in-memory (PIM). Past PIM [30, 32, 34] solutions tried to move computing logic *near* DRAM by integrating DRAM with a logic die using 3D stacking [73, 74, 60]. This helps reduce latency and increase bandwidth, however, the functionality and design of the DRAM itself remains unchanged. Also, this approach adds substantial cost to the overall system as each DRAM die needs to be augmented with a separate logic die. Integrating computation on the DRAM die itself is difficult since the DRAM process is not optimized for logic computation.

In this dissertation, we instead completely eliminate the line that distinguishes memory from compute units. Similar to the human brain, which does not separate these two functionalities distinctly, we perform computation directly on the bit lines of the memory itself, keeping data in-place. This eliminates data movement and hence significantly improves energy efficiency and performance. Furthermore, we take advantage of the fact that over 70% of silicon in today's processor dies simply stores and provides data retrieval; harnessing this area by re-purposing it to perform computation can lead to massively parallel processing.

The proposed approach builds on an earlier silicon test chip implementation [1] and architectural prototype [5] that shows how simple logic operations (AND/NOR) can be performed directly on the bit lines in a standard SRAM array. This is performed by enabling SRAM rows simultaneously while leaving the operands in-place in memory. This dissertation presents the *Neural Cache* architecture which leverages these simple logic operations to perform *arithmetic computation* (add, multiply, and reduction) *directly in the SRAM array* by storing the data in a transposed format and performing bit-serial computation while incurring only an estimated 7.5% area overhead (translates to less than 2% area overhead for the processor die). Each column in an array performs a separate calculation and the thousands of memory arrays in the cache can operate concurrently.

The end result is that cache arrays morph into massive vector compute units (up to 1,146,880 bit-serial ALU slots in a Xeon E5 cache) that are one to two orders of magnitude larger than modern graphics processor's (GPU's) aggregate vector width. By avoiding data movement in and out of memory arrays, we naturally save vast amounts of energy that is typically spent in shuffling data between compute units and on-chip memory units in modern processors.

Neural Cache leverages opportunistic in-cache computing resources for accelerating Deep Neural Networks (DNNs). There are two key challenges to harnessing a cache's computing resources. First, all the operands participating in an in-situ operation must share bitlines and be mapped to the same memory array. Second, intrinsic data-parallel operations in DNNs have to be exposed to the underlying parallel hardware and cache geometry. We propose a data layout and execution model that solves these challenges and harnesses the full potential of in-cache compute capabilities. Further, we find that thousands of in-cache compute units can be utilized by replicating data and improving data reuse. Techniques for low-cost replication, reducing data movement overhead, and improving data reuse are discussed.

In summary, this dissertation offers the following contributions:

- This is the first work to demonstrate in-situ arithmetic compute capabilities for caches. We present a compute SRAM array design that is capable of performing additions and multiplications. A critical challenge for enabling complex operations in cache is facilitating interaction between bit lines. We propose a novel bit-serial implementation with transposed data layout to address the above challenge. We designed an 8T SRAM-based hardware transpose unit for dynamically transposing data on the fly.

- Compute capabilities transform caches to massively data-parallel co-processors at negligible area cost. For example, we can re-purpose the 35 MB Last Level Cache (LLC) in the server class Intel Xeon E5 processor to support 1,146,880 bit-serial ALU slots. Furthermore, in-situ computation in caches naturally saves the on-chip data movement overheads.

- We present the *Neural Cache* architecture which re-purposes the last-level cache to accelerate DNN inferences. A key challenge is exposing the parallelism in DNN computation to the underlying cache geometry. We propose a data layout that solves these challenges and harnesses the full potential of in-cache compute capabilities. Further, techniques that reduce data movement overheads are discussed.

- The *Neural Cache* architecture is capable of fully executing convolutional, fully connected,

and pooling layers in-cache. The proposed architecture also supports quantization and normalization in-cache. Our experimental results show that the proposed architecture can improve inference latency by 18.3× over state-of-the-art multi-core CPU (Xeon E5), and 7.7× over server-class GPU (Titan Xp), for the Inception v3 model. *Neural Cache* improves inference throughput by 12.4× over CPU (2.2× over GPU), while reducing power consumption by 59% over CPU (61% over GPU).

4.2 Neural Cache Organization

Neural Cache builds on the last level cache organization of a typical multi-core processor. Here we provide a brief overview of a cache's geometry in a modern processor.Figure 4.1 illustrates a multi-core processor modeled loosely after Intel's Xeon processors [75, 76]. Shared Last Level Cache (LLC) is distributed into many slices (14 for Xeon E5 we modeled), which are accessible to the cores through a shared ring interconnect (not shown in the figure). Figure 4.1 (b) shows a slice of LLC cache. The slice has 80 32KB banks organized into 20 ways. Each bank is connected by two 16KB sub-arrays. Figure 4.1 (c) shows the internal organization of one 16KB sub-array, composed of 8KB SRAM arrays. Figure 4.1 (d) shows one 8KB SRAM array. An SRAM array is organized into multiple rows of data-storing bit-cells. Bit-cells in the same row share one wordline, whereas bit-cells in the same column share one pair of bit lines.

Our proposal is to perform in-situ vector arithmetic operations within the SRAM arrays (Figure 4.1 (d)). The resulting architecture can have massive parallelism by re-purposing thousands of SRAM arrays (4480 arrays in Xeon E5) into vector computational units.

We observe that LLC access latency is dominated by wire-delays inside a cache slice, accessing upper-level cache control structures, and network-on-chip. Thus, while a typical LLC access can take ∼30 cycles, an SRAM array access is only 1 cycle (at 4 GHz clock [75]). Fortunately, in-situ architectures such as *Neural Cache* require only SRAM array accesses and do not incur the overheads of a traditional cache access. Thus vast amounts of energy and time spent on wires and

Figure 4.1: Neural Cache overview. (a) Multi-core processor with 8-24 Last Level Cache (LLC) slices. (b) Cache geometry of a single 2.5MB LLC cache slice with 80 32KB banks. Each 32KB bank has two 16KB sub-arrays. (c) One 16KB sub-array composed of two 8KB SRAM arrays. (d) One 8KB SRAM array re-purposed to store data and do bit line computation on operand stored in rows (A and B), (e) Peripherals of the SRAM array to support computation.

higher-levels of memory hierarchy can be saved.

4.3 Neural Cache Arithmetic

Compute cache [5] supported several simple operations (logical and copy). These operations are bit-parallel and do not require interaction between bit lines. *Neural Cache* requires support for more *complex operations (addition, multiplication, reduction)*. The critical challenge in supporting these complex computing primitives is facilitating interaction between bit lines. Consider supporting an addition operation that requires carry propagation between bit lines. We propose **bit-serial implementation** with **transposed data layout** to address the above challenge.

4.3.1 Bit-Serial Arithmetic

Bit-serial computing architectures have been widely used for digital signal processing [77, 78] because of their ability to provide massive bit-level parallelism at low area costs. The key idea

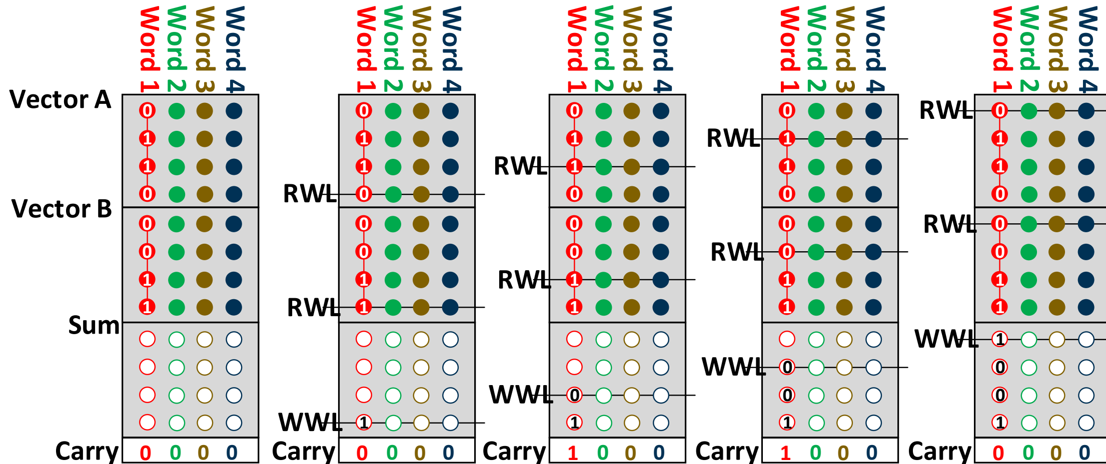

Figure 4.2: Addition operation

is to process one bit of multiple data elements every cycle. This model is particularly useful in scenarios where the same operation is applied to the same bit of multiple data elements. Consider the following example to compute the element-wise sum of two arrays with 512 32-bit elements. A conventional processor would process these arrays element-by-element taking 512 steps to complete the operation. A bit-serial processor, on the other hand, would complete the operation in 32 steps as it processes the arrays *bit-slice by bit-slice instead of element-by-element*. Note that a bit-slice is composed of bits from the same bit position, but corresponding to different elements of the array. Since the number of elements in arrays is typically much greater than the bit-precision for each element stored in them, bit-serial computing architectures can provide much higher throughput than bit-parallel arithmetic. Note also that bit-serial operation allows for flexible operand bit-width, which can be advantageous in DNNs where the required bit width can vary from layer to layer.

Note that although bit-serial computation is expected to have higher latency per operation, it is expected to have significantly larger throughput, which compensates for higher operation latency. For example, the 8KB SRAM array is composed of 256 wordlines and 256 bitlines and can operate at a maximum frequency of 4 GHz for accessing data [75, 76]. Up to 256 elements can be processed in parallel in a single array. A 2.5 MB LLC slice has 320 8KB arrays as shown in Figure 4.1. Haswell server processor's 35 MB LLC can accommodate 4480 such 8KB arrays.

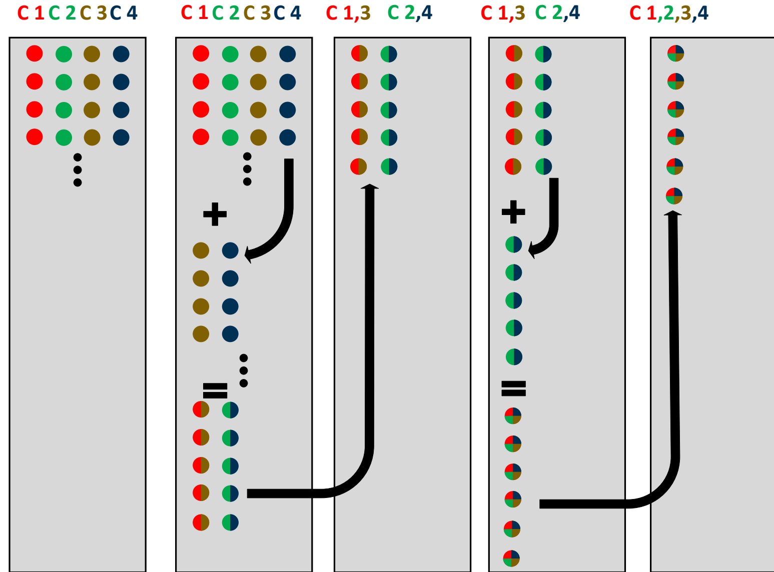

Figure 4.3: Reduction Operation

Thus up to 1,146,880 elements can be processed in parallel, while operating at a frequency of 2.5 GHz when computing. By repurposing memory arrays, we gain the above throughput for near-zero cost. Our circuit analysis estimates an area overhead of additional bit line peripheral logic to be 7.5% for each 8KB array. This translates to less than 2% area overhead for the processor die.

4.3.2 Addition

In conventional architectures, arrays are generated, stored, accessed, and processed element-by-element in the vertical direction along the bit lines. We refer to this data layout as the bit-parallel or regular data layout. Bit-serial computing in SRAM arrays can be realized by storing data elements in a transpose data layout. Transposing ensures that all bits of a data element are mapped to the same bit line, thereby obviating the necessity for communication between bit lines. Section 4.3.6 discusses techniques to store data in a transposed layout. Figure 4.2 shows an example 12×4 SRAM array with a transpose layout. The array stores two vectors A and B, each with four 4-bit elements. Four word lines are necessary to store all bit-slices of 4-bit elements.

We use the addition of two vectors of 4-bit numbers to explain how addition works in the SRAM. The 2 words that are going to be added together have to be put in the same bit line. The

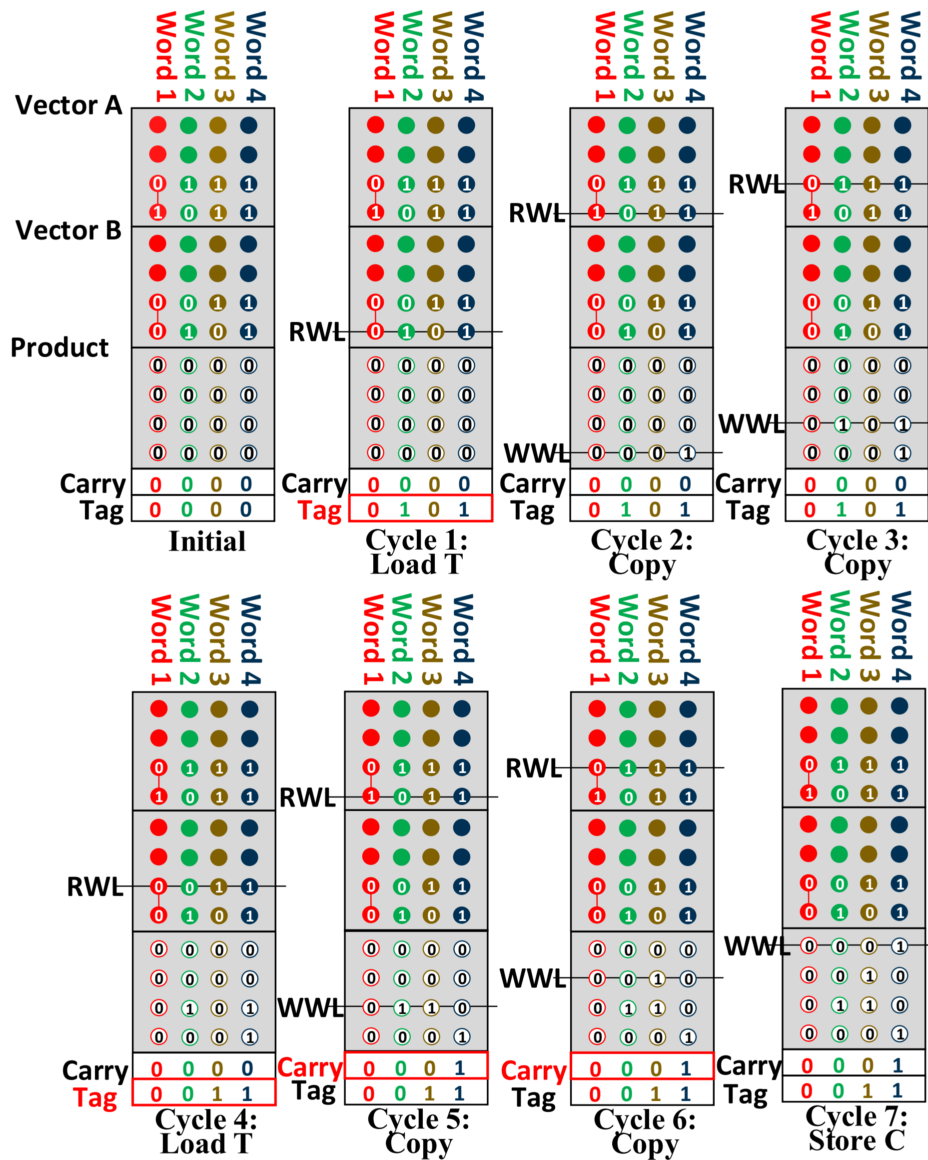

Figure 4.4: Multiplication Operation, each step is shown with four different sets of operands

vectors A and B should be aligned in the array like Figure 4.2. Vector A occupies the first 4 rows of the SRAM array and vector B the next 4 rows. Another 4 empty rows of storage are reserved for the results. There is a row of latches inside the column peripheral for the carry storage. The addition algorithm is carried out bit-by-bit starting from the least significant bit (LSB) of the two words. There are two phases in a single operation cycle. In the first half of the cycle, two read wordlines (RWL) are activated to simultaneously sense and compute on the value in cells on the same bit line. To prevent the value in the bitcell from being disturbed by the sensing phase, the RWL voltage should be lower than the normal VDD. The sense amps and logic gates in the column peripheral (Section 4.3.5) use the 2 bitcells as operands and carry latch as carry-in to generate sum and carry-out. In the second half of the cycle, a write word line (WWL) is activated to store back the sum bit. The carry-out bit overwrites the data in the carry latch and becomes the carry-in of the next cycle. As demonstrated in Figure 4.2, in cycles 2, 3, and 4, we repeat the first cycle to add the second, third, and fourth bit respectively. Addition takes n + 1, to complete with the additional cycle to write a carry at the end.

4.3.3 Multiplication

We demonstrate how bit-serial multiplication is achieved based on addition and predication using the example of a 2-bit multiplication. In addition to the carry latch, an extra row of storage, the tag latch, is added to the bottom of the array. The tag bit is used as an enable signal to the bit line driver. When the tag is one, the addition result sum will be written back to the array. If the tag is zero, the data in the array will remain. Two vectors of 2-bit numbers, A and B, are stored in the transposed fashion and aligned as shown in Figure 4.4. Another 4 empty rows are reserved for the product and initialized to zero. Suppose A is a vector of multiplicands and B is a vector of multipliers. First, we load the LSB of the multiplier to the tag latch. If the tag equals one, the multiplicand in that bit line will be copied to the product in the next two cycles, as if it is added to the partial product. Next, we load the second bit of the multiplier to the tag. If the tag equals 1, the multiplicand in that bit line will be added to the second and third bit of the product in the

next two cycles, as if a shifted version of A is added to the partial product. Finally, the data in the carry latch is stored to the most significant bit of the product. Including the initialization steps, it takes $n^2 + 5n - 2$ cycles to finish an n-bit multiplication. Division can be supported using a similar algorithm and takes $1.5n^2 + 5.5n$ cycles.

4.3.4 Reduction

Reduction is a common operation for DNNs. Reducing the elements stored on different bit lines to one sum can be performed with a series of word line moves and additions. Figure 4.3 shows an example that reduces 4 words, $C1$, $C2$, $C3$, and $C4$. First words $C3$ and $C4$ are moved below $C1$ and $C2$ to different word lines. This is followed by addition. Another set of moves and additions reduces the four elements to one word. Each reduction step increases the number of word lines to move as we increase the bits for the partial sum. The number of reduction steps needed is $\log_2$ of the words to be reduced. In column multiplexed SRAM arrays, moves between word lines can be sped up using sense-amp cycling techniques [6].

When the elements to be reduced do not fit in the same SRAM array, reductions must be performed across arrays which can be accomplished by inter-array moves. In DNNs, reduction is typically performed across channels. In the model we examined, our optimized data mapping is able to fit all channels in the space of two arrays that share sense amps. We employ a technique called packing that allows us to reduce the number of channels in large layers (Section 4.4.1).

4.3.5 SRAM Array Peripherals

The bit-line peripherals are shown in Figure 4.5. Two single-ended sense amps sense the wire-and result from two cells, A and B, in the same bitline. The sense amp in BL gives the result of $A \ \& \ B$, while the sense amp in BLB gives the result of $A' \ \& \ B'$. The sense amps can use reconfigurable sense amplifier design [3], which can combine into a large differential SA for speed in normal SRAM mode and separate into two single-ended SA for area efficiency in computation mode. Through a NOR gate, we can get $A \oplus B$ which is then used to generate the sum ($A \oplus B \oplus C_{in}$)

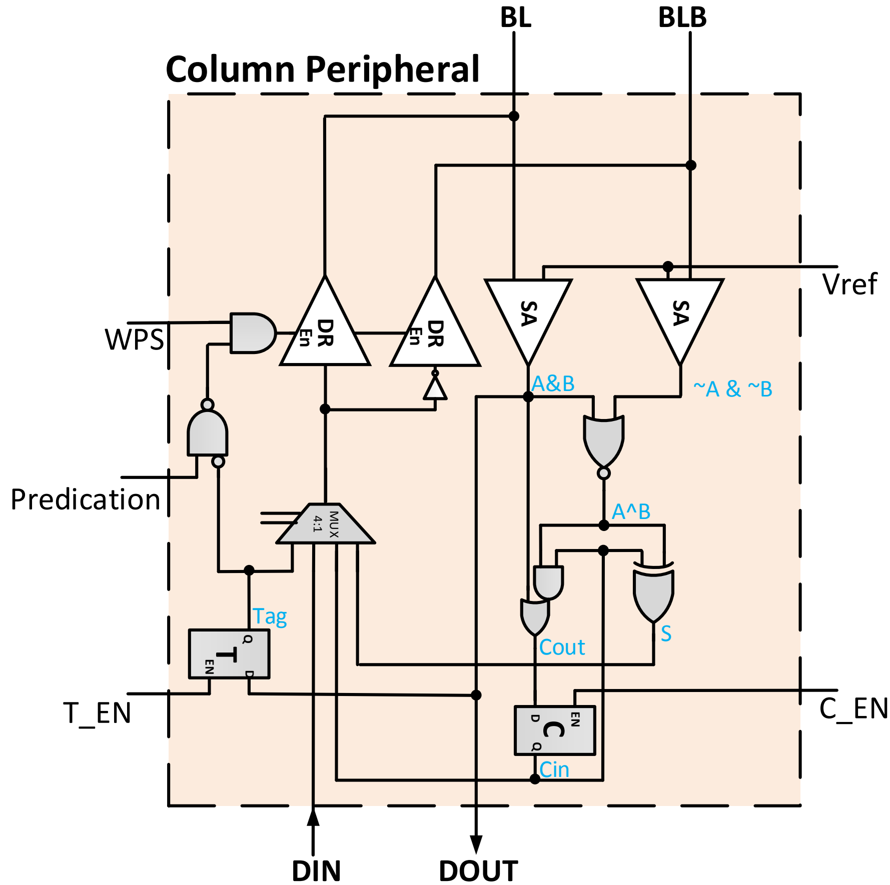

Figure 4.5: Bitline peripheral design

and Carry $((A\ \&\ B) + (A\ \oplus\ B\ \&\ C_{in}))$. As described in the previous sections, C and T are latches used to store carry and tag bits. A 4-to-1 mux selects the data to be written back among Sum, $Carry_{out}$, $Data_{in}$, and Tag. The Tag bit is used as the enable signal for the bit line driver to decide whether to write back or not.

4.3.6 Transpose Gateway Units

The *transpose* data layout can be realized in the following ways. First, leverage programmer support to store and access data in the *transpose* format. This option is useful when the data to be operated on does not change at runtime. We utilize this for filter weights in neural networks. However, this approach increases software complexity by requiring programmers to reason about a new data format and cache geometry.

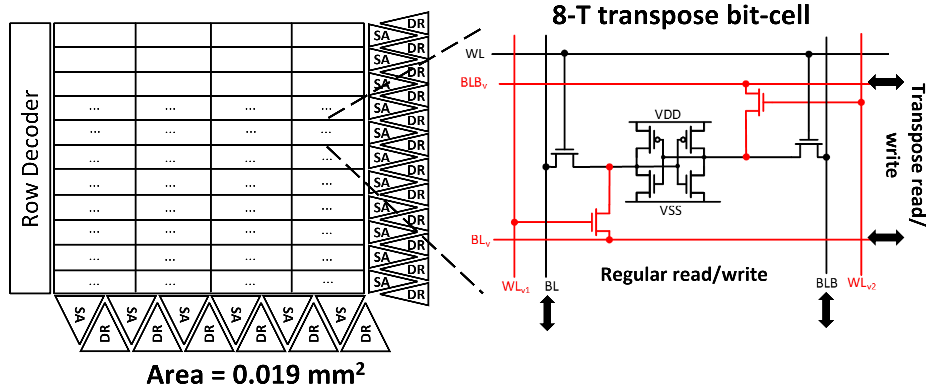

Figure 4.6: Transpose Memory Unit (TMU)

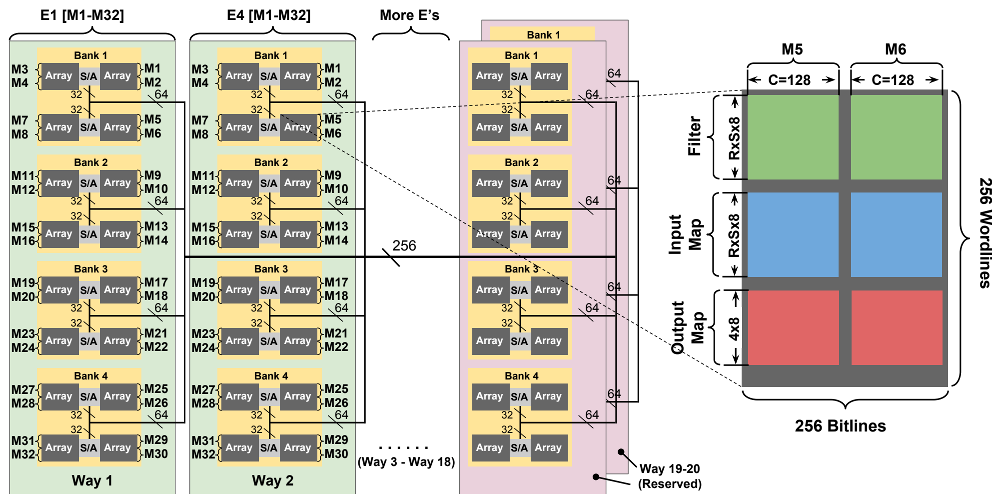

Figure 4.7: Neural Cache Data Layout for one LLC Cache Slice.

Second, design a few hardware *transpose memory units (TMUs)* placed in the cache control box (C-BOX in Figure 4.1 (b)). A TMU takes data in the *bit-parallel or regular* layout and converts it to the *transpose* layout before storing into SRAM arrays or vice-versa while reading from SRAM arrays. The second option is attractive because it supports dynamic changes to data. TMUs can be built out of SRAM arrays with multi-dimensional access (i.e., access data in both horizontal and vertical direction). Figure 4.6 shows a possible TMU design using an 8T SRAM array with sense-amps in both horizontal and vertical directions. Compared to a baseline 6T SRAM, the transposable SRAM requires a larger bitcell to enable read/write in both directions. Note that only a few TMUs are needed to saturate the available interconnect bandwidth between cache arrays. In essence, the transpose unit serves as a gateway to enable bit-serial computation in caches.

4.4 Neural Cache Architecture

The *Neural Cache* architecture transforms SRAM arrays in LLC to compute functional units. We describe the computation of convolution layers first, followed by other layers. Figure 4.7 shows the data layout and overall architecture for one cache slice, modeled after Xeon processors [75, 76]. The slice has twenty ways. The last way (way-20) is reserved to enable normal processing for CPU cores. The penultimate way (way-19) is reserved to store inputs and outputs. The remaining ways are utilized for storing filter weights and computing.

A typical DNN model consists of several layers, and each layer consists of several hundred thousands of convolutions. For example, Google's Inception v3 has 20 layers, most of which have several branches. Inception v3 has $\approx$ 0.5 million convolutions in each layer on average. *Neural Cache* computes layers and each branch within a layer serially. The convolutions within a branch are computed in parallel to the extent possible. Each of the 8KB SRAM arrays computes convolutions in parallel. The inputs are streamed in from the reserved way-19. Filter weights are stationary in compute arrays (way-1 to way-18).

Neural Cache assumes 8-bit precision and quantized inputs and filter weights. Several works [79, 80, 81] have shown that 8-bit precision has sufficient accuracy for DNN inference. 8-bit precision was adopted by Google's TPU [82]. Quantizing input data requires re-quantization after each layer as discussed in Section 4.4.4.

4.4.1 Data Layout

This section first describes the data layout of one SRAM array and the execution of one convolution. Then we discuss the data layout for the whole slice and parallel convolutions across arrays and slices.

The convolutions performed can be broken down into many different dimensions of matrix-vector multiplications. *Neural Cache* breaks down a convolution to vector-vector dot product (vector dimension $R \times S$), followed by reduction across channels (C). The different filter batches

(M) are computed in parallel using the above step. A new input vector is used for each stride. In this paper, we analyze the state-of-art Inception v3 model which has 94 convolutional sub-layers. *Neural Cache* is utilized to accelerate not only convolutional layers but also pooling and fully connected layers.

A *single convolution* consists of generating *one* of the $E \times E \times M$ output elements. This is accomplished by multiplying $R \times S \times C$ input filter weights with the same size window from the input feature map across the channels. *Neural Cache* exploits channel-level parallelism in a single convolution. For each convolution, an array executes $R \times S$ Multiply and Accumulate (MAC) in parallel across channels. This is followed by a reduction step across channels.

An example layout for a single array is shown in Figure 4.8 (a). Every bitline in the array has 256 bits and can store 32 1-byte elements in a transpose layout. Every bitline stores $R \times S$ filter weights (green dots). The channels are stored across bit lines. To perform MACs, space is reserved for accumulating partial sums (lavender dots) and for the scratch pad (pink dots). Partial sums and scratch pad take 3×8 and 2×8 word lines.

Reduction requires an additional 8×8 word lines as shown in Figure 4.8 (b). However, the scratch pad and partial sum can be overwritten for reduction as the values are no longer needed. The maximum size for reducing all partial sums is 4 bytes. So to perform reduction, we reserve two 4-byte segments. After adding the two segments, the resultant can be written over the first segment again. The second segment is then loaded with the next set of reduced data.

Each array may perform several convolutions in series, thus we reserve some extra space for output elements (red dots). The remaining word lines are used to store input elements (blue dots). It is desirable to use as many word lines as possible for inputs to maximize input reuse across convolutions. For example in a 3×3 convolution with a stride of 1, 6 of the 9 bytes are reused across each set of input loads. Storing many input elements allows us to exploit this locality and reduce input streaming time.

The filter sizes ($R \times S$) range from 1-25 bytes in Inception v3. The common case is a 3×3 filter. *Neural Cache* data mapping employs *filter splitting* for large filters. The filters are split

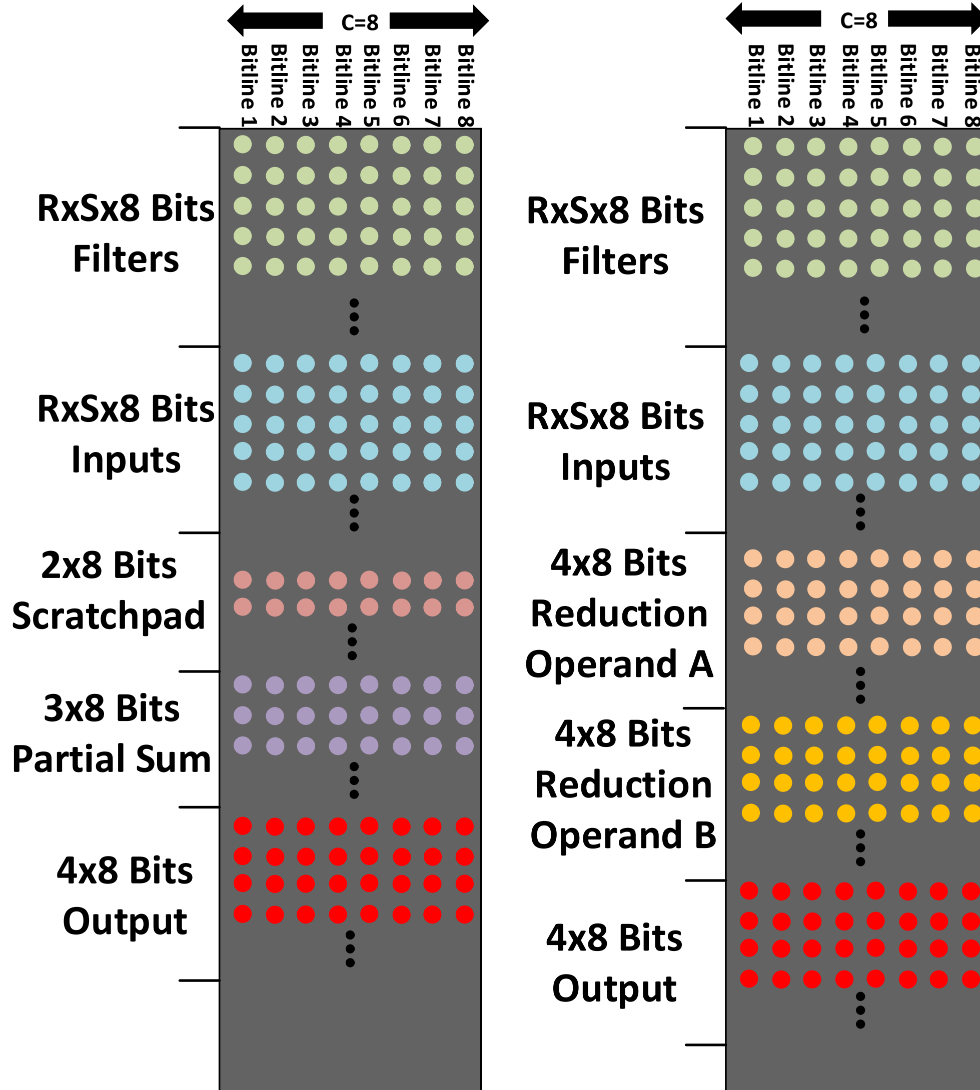

Figure 4.8: Array Data Layout (a) MACs (b) Reduction.

across bitlines when their size exceeds 9 bytes. The other technique employed is *filter packing*. For 1×1 filters we compress multiple channels into the same bit line. Instead of putting a single byte of the filter, we can instead put 16 bytes of the filter. Since 1×1 has no input reuse, we only need one input byte at a time. By packing the filters, the number of reductions is decreased at no cost to input loading. More importantly, by packing all channels in the network it is guaranteed to fit within 2 arrays that share sense-amps, making reduction faster and easier.

When mapping the layers, first the new effective channel size after packing and filter splitting is calculated. This channel number is then rounded up to the nearest power of 2, by padding the

extra channels with zero. Depending on parameters, filters from different output channels (M's) can be placed in the same array as well. For instance, the first layer of Inception v3 has so few channels, that all M's can be placed in the same array.

Finally, although all operations are accomplished at the bit level, each data element is stored as a multiple of a byte, although it may not necessarily require that many bits. This is done for simplicity, software programmability, and easier data movement.

4.4.2 Data Parallel Convolutions

Each layer in the network produces $M \times E \times E$ outputs, and each output requires one convolution. All the outputs can be produced in parallel, provided we have sufficient computing resources. We find that in most scenarios half an array or even a quarter of an array is sufficient to computes each output element. Thus *Neural Cache*'s massively parallel computing resources can be utilized to do convolutions in parallel.

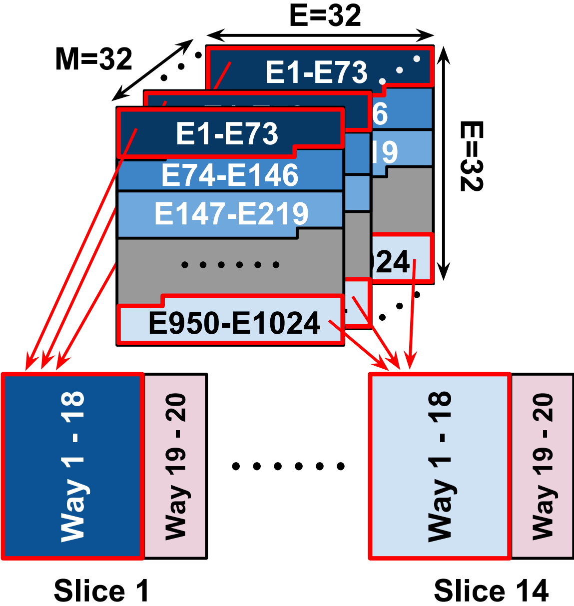

Figure 4.9: Partitioning of Convolutions between Slices.

Figure 4.9 shows the division of convolutions among cache slices. Each slice can be visualized as a massively parallel SIMD core. Each slice gets roughly an equal number of outputs to work on.

Mapping consecutive output elements to the same slice has the nice property that the next layer can find much of its required input data locally within its reserved way. This reduces the number of inter-slice transfers needed when loading input elements across layers.

Figure 4.7 shows the layout of one slice for an example layer. This layer has parameters: $R \times S = 3 \times 3$, $C = 128$, $M = 32$, $E = 32$. Figure 4.7 (right) shows a single array. Each array can pack two complete filters ($R \times S \times C$). The array shown in figure 4.7 packs $M = 5$ and $M = 6$ in the same array. Thus each array can compute two convolutions in parallel. Each way (4 banks, or 16 arrays) computes 32 convolutions in parallel ($E_i \; \forall \; M's \; M1 - M32$). The entire slice can compute 18×32 convolutions in parallel. Despite this parallelism, some convolutions need to be computed serially, when the total number of convolutions to be performed exceeds the total number of convolutions across *all slices*. In this example, about 4 convolutions are executed serially.

When mapping inputs and filters to the cache, we prioritize uniformity across the cache over complete (i.e., 100%) cache compute resource utilization. This simplifies operations as all data is mapped in such a way that all arrays in the cache will be performing the exact same instruction at the same time and also simplifies data movement.

4.4.3 Orchestrating Data Movement

Data parallel convolutions require replication of filters and streaming of inputs from the reserved way. This section discusses these data-movement techniques.

We first describe the interconnect structure. Our design is based on the on-chip LLC [75] of the Intel Xeon E5-2697 processor. There are in total 14 cache slices connected by a bidirectional ring. Figure 4.7 shows the interconnect within a cache slice. There is a 256-bit data bus that is capable of delivering data to each of the 20 ways. The data bus is composed of 4 64-bit data buses. Each bus serves one quadrant. A quadrant consists of a 32KB bank composed of four 8KB data arrays. Two 8 KB arrays within a bank share sense-amps and receive 32 bits every bus cycle.

Filter Loading: We assume that all the filter data for a specific layer reside in DRAM before

the entire computation starts. Filter data for different layers of the DNN model are loaded in serial. Within a convolution layer, regardless of the number of output pixels done in serial, the positions of filter data in the cache remain stationary. Therefore, it is sufficient to load filter data from *memory only once per layer*.

Across a layer, we use M sets of $R \times S$ filters. By performing more than M convolutions, we will replicate filters across and within slices. Fortunately, the inter-slice ring and intra-slice bus are both naturally capable of broadcasting, allowing for easy replication for filter weights. Each filter weight loaded from DRAM is broadcasted to all slices over the ring and all ways over the intra-slice bus.

In each array that actively performs computation, $R' \times S' \times 8$ word lines are loaded with filter data, where $R' \times S'$ is the equivalent filter dimension after packing and filter splitting, and 8 is the bit-serial factor due to 8-bits per element. *Neural Cache* assumes that filter weights are preprocessed to a transposed format and laid out in DRAM such that they map to correct bitlines and wordlines. Our experiments decode the set address and faithfully model this layout. Software transposing of weights is a one time cost and can be done cheaply using x86 SIMD shuffle and pack instructions [83, 84].

Input Data Streaming: We only load the input data of the first layer from DRAM, because, for the following layers, the input data have already been computed and temporarily stored in the cache as outputs. In some layers, there are multiple output pixels to be computed in serial, and we need to stream in the corresponding input data as well since different input data is required at any specific cache array for generating different output pixels.

Within each array that actively performs computation, $R' \times S' \times 8$ word lines of input data are streamed in, where $R' \times S'$ is the equivalent filter dimension after packing and filter splitting and 8 is the bit-serial factor. When loading inputs from DRAM for the first layer, input data is transposed using the TMUs at C-BOX.

Since each output pixel requires a different set of input data, input loading time can be high. We exploit duplicity in input data to reduce transfer time. For different output channels (M's) with

the same output pixel position (i.e. same E_i for different M's), the input data is the same and can be replicated. Thus for scenarios where these different output channels are in different ways, the input data can be loaded in one intra-slice bus transfer. Furthermore, a large portion of input data can be reused across output pixels done in serial as discussed in Section 4.4.1. This helps reduce transfer time. We also observe that often the input data is replicated across arrays within a bank. We put a 64-bit latch at each bank so that the total input transfer time can be halved.

Note, that intra-bus transfers happen in parallel across different cache slices. Thus distributing E's across slices significantly reduces input loading time as well by leveraging intra-slice network bandwidth.

Output Data Management: One way of each slice (128 KB), is reserved for temporarily storing the output data. After all the convolutions for a layer are executed, data from compute arrays are transferred to the reserved way. We divide the computation into different slices according to the output pixel position. Contiguous output pixels are assigned to the same slice so that one slice will need neighboring inputs for at most $R \times E$ pixels. This design significantly reduces inter-slice communication for transferring outputs.

4.4.4 Supporting Functions

Max Pooling layers compute the maximum value of all the inputs inside a sliding window. The data mapping and input loading would function the same way as convolution layers, except without any filters in the arrays.

Calculating the maximum value of two or more numbers can be accomplished by designating a temporary maximum value. The temporary maximum is then subtracted by the next output value and the resultant is stored in a separate set of word lines. The most significant bit of the result is used as a mask for a selective copy. The next input is then selectively copied to the maximum location based on the value of the mask. This process is repeated for the rest of the inputs in the array.

Quantization of the outputs is done by calculating the minimum and maximum value of all

the outputs in the given layer. The min can be computed using a similar set of operations described for max. For quantization, the min and max will first be calculated within each array. Initially, all outputs in the array will be copied to allow performing the min and max at the same time. After the first reduction, all subsequent reductions of min/max are performed the same way as channel reductions. Since quantization needs the min and max of the entire cache, a series of bus transfers are needed to reduce min and max to one value. This is slower than in-array reductions, however, unlike channel reduction, min/max reduction happens only once in a layer making the penalty small.

After calculating the min and max for the entire layer, the result is then sent to the CPU. The CPU then performs floating point operations on the min and max of the entire layer and computes two unsigned integers. These operations take too few cycles to show up in our profiling. Therefore, it is assumed to be negligible. The two unsigned integers sent back by the CPU are used for in-cache multiplications, adds, and shifts to be performed on all the output elements to finally quantize them.

Batch Normalization requires first quantizing to 32-bit unsigned. This is accomplished by multiplying all values by a scalar from the CPU and performing a shift. Afterward, scalar integers are added to each output in the corresponding output channel. These scalar integers are once again calculated in the CPU. Afterward, the data is re-quantized as described above.

In Inception v3, **ReLU** operates by replacing any negative number with zero. We can write zero to every output element with the MSB acting as an enable for the write. Similar to max/min computations, ReLU relies on using a mask to enable selective write.

Avg Pool is mapped in the same way as in max pool. All the inputs in a window are summed and then divided by the window size. Division is slower than multiplication, but the divisor is only 4 bits in Inception v3.

Fully Connected layers are converted into convolution layers in TensorFlow. Thus, we are able to treat the fully connected layer as another convolution layer.

4.4.5 Batching

We apply batching to increase the system throughput. Our experiments show that loading filter weights takes up about 46% of the total execution time. Batching multiple input images significantly amortizes the time for loading weights and therefore increases system throughput. *Neural Cache* performs batching in a straightforward way. The image batch will be processed sequentially in the layer order. For each layer, at first, the filter weights are loaded into the cache as described in Section 4.4.1. Then, a batch of input images is streamed into the cache, and computation is performed in the same way as without batching. For the whole batch, the filter weights of the involved layer remain in the arrays, without reloading. Note that for the layers with heavy-sized outputs, after batching, the total output size may exceed the capacity of the reserved way. In this case, the output data is dumped to DRAM and then loaded again into the cache. In the Inception v3, the first five layers require dumping output data to DRAM.

4.4.6 ISA support and Execution Model

Neural Cache requires supporting a few new instructions: in-cache addition, multiplication, reduction, and moves. Since at any given time only one layer in the network is being operated on, all compute arrays execute the same in-cache compute instruction. The compute instructions are followed by move instructions for data management. The intra-slice address bus is used to broadcast the instructions to all banks. Each bank has a control FSM which orchestrates the control signals to the SRAM arrays. The area of one FSM is estimated to be 204 μm^2, across 14 slices which sums to 0.23 mm^2. Given that each bank is executing the same instruction, the control FSM can be shared across a way or even a slice. We chose not to optimize this because of the insignificant area overheads of the control FSM. *Neural Cache* computation is carried out in 1-19 ways of each slice. The remaining way (way-20) can be used by other processes/VMs executing on the CPU cores for normal background operation. Intel's Cache Allocation Technology (CAT) [85] can be leveraged to dynamically restrict the ways accessed by CPU programs to the reserved way.

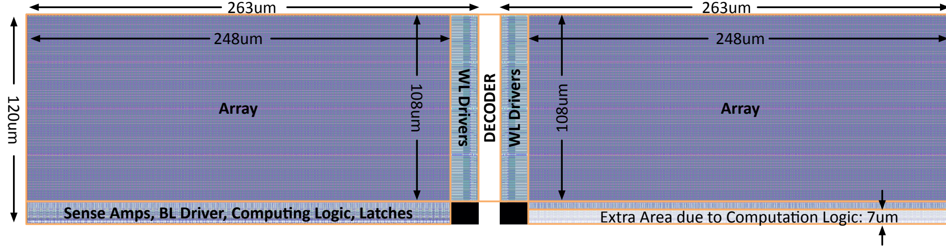

Figure 4.10: SRAM Array Layout.

Layer	H	$R \times S$	E	C	M	Conv	Filter Size / MB	Input Size / MB
Conv2D_1a_3x3	299	9	149	3	32	710432	0.001	0.256
Conv2D_2a_3x3	149	9	147	32	32	691488	0.009	0.678
Conv2D_2b_3x3	147	9	147	32	64	1382976	0.018	0.659
MaxPool_3a_3x3	147	9	73	0	64	0	0.000	1.319
Conv2D_3b_1x1	73	1	73	64	80	426320	0.005	0.325
Conv2D_4a_3x3	73	9	71	80	192	967872	0.132	0.407
MaxPool_5a_3x3	71	9	35	0	192	0	0.000	0.923
Mixed_5b	35	1-25	35	48-192	32-192	568400	0.243	0.897
Mixed_5c	35	1-25	35	48-256	48-256	607600	0.264	1.196
Mixed_5d	35	1-25	35	48-288	48-288	607600	0.271	1.346
Mixed_6a	35	1-9	17	64-288	64-384	334720	0.255	1.009
Mixed_6b	17	1-9	17	128-768	128-768	443904	1.234	0.847
Mixed_6c	17	1-9	17	160-768	160-768	499392	1.609	0.847
Mixed_6d	17	1-9	17	160-768	160-768	499392	1.609	0.847
Mixed_6e	17	1-9	17	192-768	192-768	499392	1.898	0.847
Mixed_7a	17	1-9	8	192-768	192-768	254720	1.617	0.635
Mixed_7b	8	1-9	8	384-1280	192-1280	208896	4.805	0.313
Mixed_7c	8	1-9	8	384-2048	192-2048	208896	5.789	0.500
AvgPool	8	64	1	0	2048	0	0.000	0.125
FullyConnected	1	1	1	2048	1001	1001	1.955	0.002

Table 4.1: Parameters of the Layers of Inception V3.

4.5 Evaluation Methodology

Baseline Setup: For a baseline, we use dual-socket Intel Xeon E5-2697 v3 as CPU, and Nvidia Titan Xp as GPU. The specifications of the baseline machine are in Table4.2. Note that the CPU specs are per socket. Note that the baseline CPU has the exact cache organization (35 MB L3 per socket) as we used in *Neural Cache* modeling. The benchmark program is the inference phase of the Inception v3 model [86]. We use TensorFlow as the software framework to run NN inferences on both baseline CPU and GPU. The default profiling tool of TensorFlow is used for generating execution time breakdown by network layers for both CPU and GPU. The reported baseline results are based on the unquantized version of Inception v3 model, because we observe that the 8-bit

CPU	**Intel Xeon E5-2697 v3**
Base Frequency	2.6 GHz
Cores/Threads	14/28
Process	22 nm
TDP	145 W
Cache	32 kB i-L1 per core, 32 kB d-L1 per core, 256 kB L2 per core, 35 MB shared L3
System Memory	64 GB DRAM, DDR4

GPU	**Nvidia Titan Xp**
Frequency	1.6 GHz
CUDA Cores	3840
Process	16 nm
TDP	250 W
Cache	3MB shared L2
Graphics Memory	12 GB DRAM, GDDR5X

Table 4.2: Baseline CPU & GPU Configuration.

quantized version has a higher latency on the baseline CPU due to a lack of optimized libraries for quantized operations (540 ms for quantized / 86 ms for unquantized). To measure the execution power of the baseline, we use RAPL[87] for CPU power measurement and Nvidia-SMI [88] for GPU power measurement.

Modeling of Neural Cache: To estimate the power and delay of the SRAM array, the SPICE model of an 8KB computational SRAM is simulated using the 28 nm technology node. The nominal voltage for this technology is 0.9V. Since we activate two read word lines (RWL) at the same time in computation, we reduce the RWL voltage to improve bit cell read stability. But lowering the RWL voltage will increase the read delay. We simulated various RWL voltages and to achieve the industry standard 6 sigma margin, we choose 0.66V as the RWL voltage. The total computing time is 1022ps. The delay of a normal SRAM read is 654ps given by the standard foundry memory compiler. So the computation SRAM delay is about $1.6\times$ larger than the normal SRAM read. Xeon processor cache arrays can operate at 4 GHz [75, 76]. We conservatively choose a frequency of 2.5 GHz for *Neural Cache* in the compute mode. Our SPICE simulations

also provided the total energy consumption in one clock cycle for reading out 256 bits in SRAM mode or operating on 256 bitlines in computation mode. This was estimated to be 13.9pJ for SRAM access cycles and 25.7pJ for compute cycles. Since we model *Neural Cache* for the Intel Xeon E5-2697 v3 processor at 22 nm, the energy was scaled down to 8.6pJ for SRAM access cycles and 15.4pJ for compute cycles. The SRAM array layout is shown in Figure 4.10. Compute capabilities incur an area overhead of 7.5% increase due to extra logic and an extra decoder.

For the in-cache computation, we developed a cycle-accurate simulator based on the deterministic computation model discussed in Section 4.4. The simulator is verified by running data traces on it and matching the results with traces obtained from instrumenting the TensorFlow model for Inception v3. To model the time of data loading, we write a C micro-benchmark, which sequentially reads out the exact sets within a way that needs loading with data. The set decoding was reverse engineered based on Intel's last level cache architecture [75, 76]. We conduct this measurement for all the layers according to their different requirement of sets to be loaded. Then, the measured time is multiplied with a factor that accounts for the sequential data transfer across different ways, as well as the sequential computation of different output pixels within a layer. Note that the measured time includes the data loading from DRAM to the on-chip cache, but for input data loading, the data is already in the cache (except the first layer). For more accurate modeling, the micro-benchmark is profiled by the VTune Amplifier[89] for estimating the percentage of DRAM bound cycles, and such DRAM-loading time is excluded from the input data loading time.

4.6 Results

4.6.1 Latency

Figure4.11 reports the latency of all layers in the Inception v3 model. A majority of time is spent on the mixed layers for both CPU and GPU, while *Neural Cache* achieves significantly better latency than baseline for all layers. This is primarily because *Neural Cache* exploits the available data parallelism across convolutions to provide low-latency inference. *Neural Cache*'s data mapping not

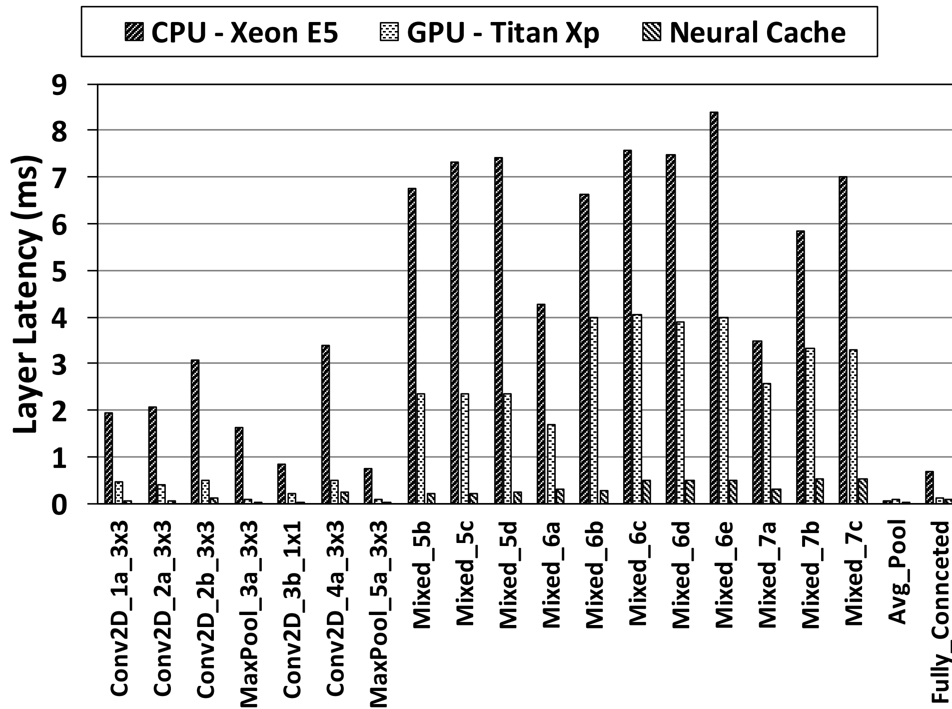

Figure 4.11: Inference latency by Layer of Inception v3.

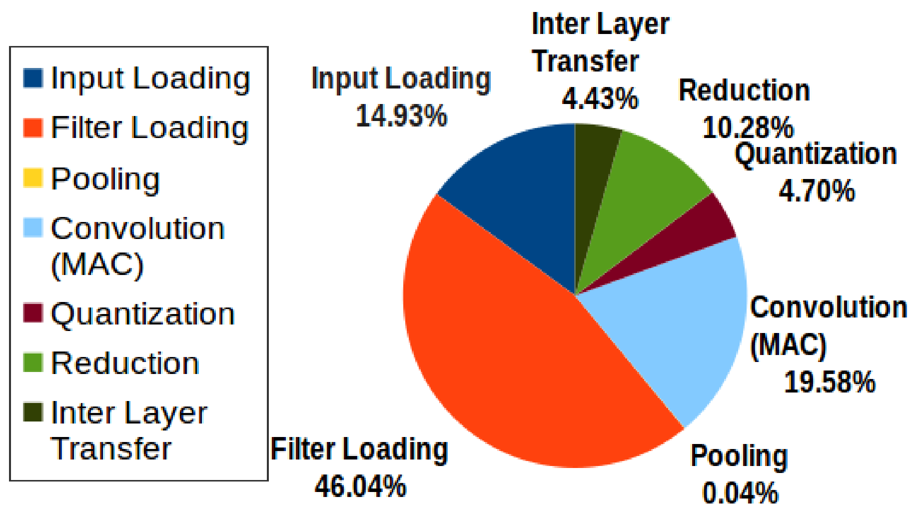

Figure 4.12: Inference Latency Breakdown.

only makes the computation entirely data independent, but the operations performed are identical, allowing for SIMD instructions.

Consider an example layer, Conv2D_Layer_2b_3×3. This layer computes $\approx$ 1.4 *million* convolutions, out of which *Neural Cache* executes $\approx$ 32 *thousand* convolutions in parallel and 43 in series. The compute cache arrays show 99.7% utilization for this layer during convolutions (after data loading). Each convolution takes 2784 cycles (236 cycles/MAC$\times$ 9 + 660 reduction cycles). Then reduction takes a further 660 cycles. The whole layer takes 117912 cycles (43 convolutions in series $\times$ 2784 cycles), taking 0.0479 ms to finish the convolutions for *Neural Cache* running at 2.5 GHz. The remaining time for the layer is attributed to data loading. CPU and GPU cannot take advantage of data parallelism on this scale due to a lack of sufficient compute resources and on-chip data-movement bottlenecks.

Figure4.12 shows the breakdown of *Neural Cache* latency. Loading filter weights into the cache and distributing them into arrays takes up 46% of the total execution time, and 15% of the time is spent on streaming in the input data to the appropriate position within the cache. Transferring output data to the proper reserved space takes 4% of the time. Filter loading is the most time consuming part since data is loaded from DRAM. The remaining parts are for computation, including multiplication in convolution (MACs) (20%), reduction (10%), quantization (5%), and pooling (0.04%).

Figure4.13 shows the total latency for *Neural Cache*, as well as the baseline CPU and GPU, running inference on the Inception v3 model. *Neural Cache* achieves a 7.7× speedup in latency compared to baseline GPU, and 18.3× speedup on baseline CPU. The significant speedup can be attributed to the elimination of high overheads of on-chip data movement from the cache to the core registers, and data parallel convolutions.

4.6.2 Batching

Figure 4.14 shows the system throughput in the number of inferences per second as the batch size varies. *Neural Cache* outperforms the maximum throughput of baseline CPU and GPU even

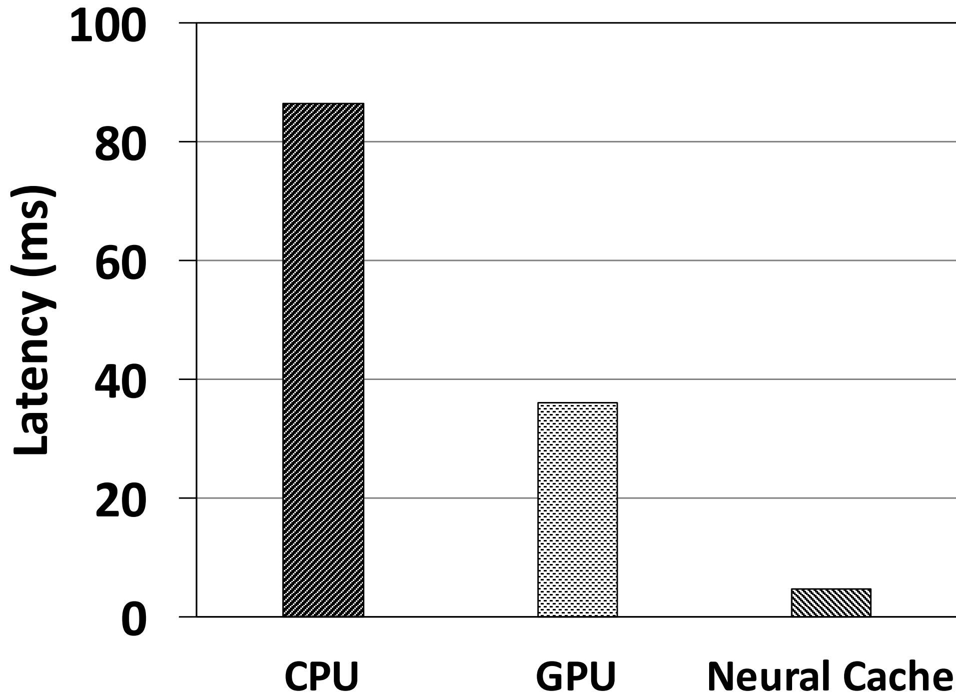

Figure 4.13: Total Latency on Inception v3 Inference.

without batching. This is mainly due to the low latency of *Neural Cache*. Another reason is that *Neural Cache* scales linearly with the number of host CPUs, and thus the throughput of *Neural Cache* doubles with a dual-socket node.

As the batch size N increases from 1 to 16, the throughput of *Neural Cache* increases steadily. This can be attributed to the amortization of filter loading time. For even higher batch sizes, the effect of such amortization diminishes, and therefore the throughput plateaus. As shown in the figure, the GPU throughput also plateaus after batch size exceeds 64. At the highest batch size, *Neural Cache* achieves a throughput of 604 inferences/sec, which is equivalent to $2.2\times$ GPU throughput, or $12.4\times$ CPU throughput.

4.6.3 Power and Energy

This section discusses the energy and power consumption of *Neural Cache*. Table4.3 summarizes the energy/power comparison with baseline. *Neural Cache* achieves an energy efficiency that is $16.6\times$ better than the baseline GPU and $37.1\times$ better than the CPU. The energy efficiency improve-

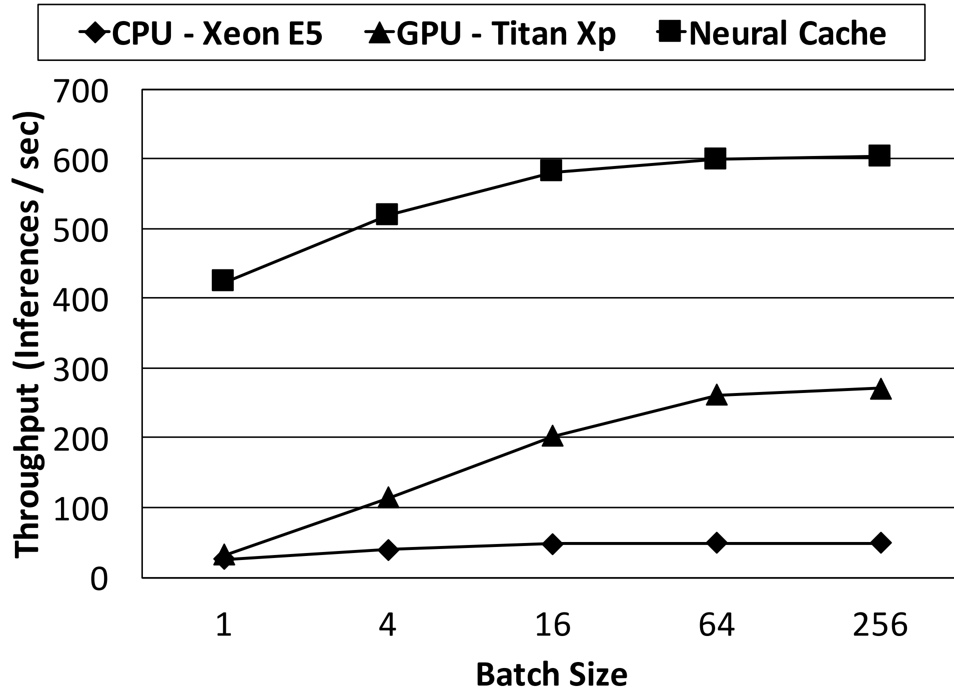

Figure 4.14: Throughput with Varying Batching Sizes.

	CPU	GPU	Neural Cache
Total Energy / J	9.137	4.087	0.246
Average Power / W	105.56	112.87	52.92

Table 4.3: Energy Consumption and Average Power.

ment can be explained by the reduction in data movement, the increased instruction efficiency of SIMD-like architecture, and the optimized in-cache arithmetic units.

The average power of *Neural Cache* is 53.11% and 49.87% lower than the GPU and CPU baselines respectively. Thus, *Neural Cache* does not form a thermal challenge for the servers. *Neural Cache* also outperforms both CPU and GPU baselines in terms of energy-delay product (EDP), since it consumes less energy and has a shorter latency.

Cache Capacity	35MB	45MB	60MB
Inference Latency	4.72 ms	4.12 ms	3.79 ms

Table 4.4: Scaling with Cache Capacity (Batch Size=1).

4.6.4 Scaling with Cache Capacity

In this section we explore how increasing the cache capacity affects the performance of *Neural Cache*. We increase the cache size from 35MB (14 slices) to 45MB (18 slices), and 60MB (24 slices). Increasing the number of slices speedups most aspects of the inference. The total number of arrays that compute increases, thereby increasing convolutions that can be done in parallel. This reduces the convolution compute time. Filter loading will not be affected as unique filters are not done across slices, rather filters are replicated across the slices. Input loading, however, will decrease since we are using the additional intra-slice bandwidth of new slices to decrease the time it takes to broadcast inputs. Inter-layer data transfer will also be reduced due to fewer outputs being transferred in each slice.

4.7 Summary

Caches have traditionally served only as intermediate low-latency storage units. Our work directly challenges this conventional design paradigm and proposes to impose a dual responsibility on caches: store *and* compute data. By doing so, we turn them into massively parallel vector units and drastically reduce on-chip data movement overhead. In this dissertation, we propose the *Neural Cache* architecture to allow massively parallel compute for Deep Neural Networks. Our advancements in compute cache arithmetic and neural network data layout solutions allow us to provide competitive performance comparably to modern GPUs with negligible area overheads. Nearly three-fourths of a server class processor die area today is devoted to caches. Even accelerators use large caches. Why would one *not* want to turn them into compute units?

CHAPTER 5

Eidetic

Neural Cache showed huge gains for CNNs in throughput, latency, and energy by repurposing the LLC of a Xeon for in-SRAM computing. However, this architecture was limited by the existing cache structure. A custom architecture would allow for modifications to make storage, compute, and dataflow more efficient. In this chapter we present *Eidetic*, an in-memory matrix multiplication accelerator for neural networks.

5.1 Introduction

In-memory computing has emerged as a way to bridge the memory wall where memory bandwidth and memory energy have come to dominate computation bandwidth and energy [72]. Combining the memory and compute into a single unit increases both the compute and storage resources available on-chip. An increase in available memory capacity also allows applications with larger working sets to be stored on-chip, reducing the effect of the memory wall. In particular, in-SRAM computing requires minimal modifications to the SRAM array to enable arithmetic computation, making it an ideal candidate for getting the best of compute and memory together.

The rise in popularity of neural networks has further exasperated the problems of the memory wall. Neural networks can be compute-bound from the matrix multiplication, data-bound from the large number of weights and intermediates results or sometimes both compute-bound and data-bound. Further, accelerators designed for these applications often need to balance the strict latency

requirements and the need to support batching to offset weight loading overheads and to keep the compute units saturated.

Many accelerators have achieved this in a variety of ways. Common methods include increasing the available SRAM storage on-chip, increasing the amount of batching to amortize weight loading overheads, increasing the bandwidth to memory, using a lower precision to reduce weight sizes, and taking advantage of sparsity. In-SRAM computing is an attractive solution for the memory wall problem experienced by neural networks [1, 3, 4, 5, 90, 42, 41, 40, 91, 92, 93, 94, 46, 47, 48, 45, 43, 44]. On-chip SRAM memory can be morphed into compute units, effectively increasing the amount of memory on-chip without sacrificing area for compute units. Further, bit-serial based in-SRAM computing techniques can inherently support variable bit-precisions without requiring additional hardware [90, 95].

In this dissertation, we present *Eidetic*, an in-SRAM computing based matrix multiplication accelerator for deep neural networks. *Eidetic* utilizes a system of systolic arrays composed of hybrid SRAM-compute capable processing elements (PEs) interconnected using a ring network. Multiple matrix multiplication operations can be mapped concurrently across the many systolic arrays. *Eidetic* additionally provides support for non-matrix operations commonly found in neural networks. *Eidetic*'s dataflow architecture is also specifically well-suited for running neural networks. The control logic is carefully designed to allow for easy programmability.

Eidetic is able to capture and map data flow graphs which can be used to describe the common operations of most neural networks. Networks such as Long Short-Term Memory (LSTMs) increase the complexity of these graphs with temporal elements. *Eidetic* central control logic drives its dataflow architecture, allowing high-level programmability where functions or operations are mapped to the chip. Using its central control logic *Eidetic* is able to establish the connections between mapped operations and facilitate data movement from one function to another. After being programmed with setup instructions for the initial operations, *Eidetic* processes inputs (identified by their corresponding labels) on-demand based on their availability. This eliminates the need for compiler support to generate instructions for each batch of inputs and enables a true on-demand

system. The high-level programmability will make extending support for *Eidetic* easier compared to accelerators that only support lower-level compute and data orchestration instructions. Further, it allows mappings of multiple layers from a neural network, or even the entire network to be mapped on-chip.

In summary, this dissertation makes the following contributions:

- This dissertation proposes the *Eidetic* architecture, an in-SRAM computing based matrix multiplication engine for deep neural networks. *Eidetic* eliminates weight loading stalls by storing weights in on-chip SRAM. Further, *Eidetic* leverages in-SRAM computing to reduce the overheads of on-chip data movement between storage and compute elements.

- *Eidetic* consists of multiple 1-D systolic arrays that are interconnected using a ring network. Each systolic array is composed of a novel system of compute-capable SRAM processing elements (PEs) that can efficiently support scalar-vector multiplication.

- *Eidetic* implements a novel dataflow architecture optimized for RNNs with custom control logic to allow for high-level programmability while also reducing off-chip interactions in the form of instructions and data.

- We evaluate *Eidetic* on Google's Neural Machine Translation (GNMT) encoder and observe a 4.30$\times$ increase in throughput over the TPUv2 Board and a 17.20$\times$ increase of a single TPU chip. Additionally, this is achieved with on-demand input processing instead of batching resulting in 7.77$\times$ lower average latency when compared to TPUv2.

5.2 Eidetic Abstractions and Execution Model

5.2.1 Matrix Multiplication Systolic Arrays

Systolic arrays are a popular architecture for implementing matrix multiplication. A weight stationary mapping is a common approach where the weight matrix is stored across the PE units. The

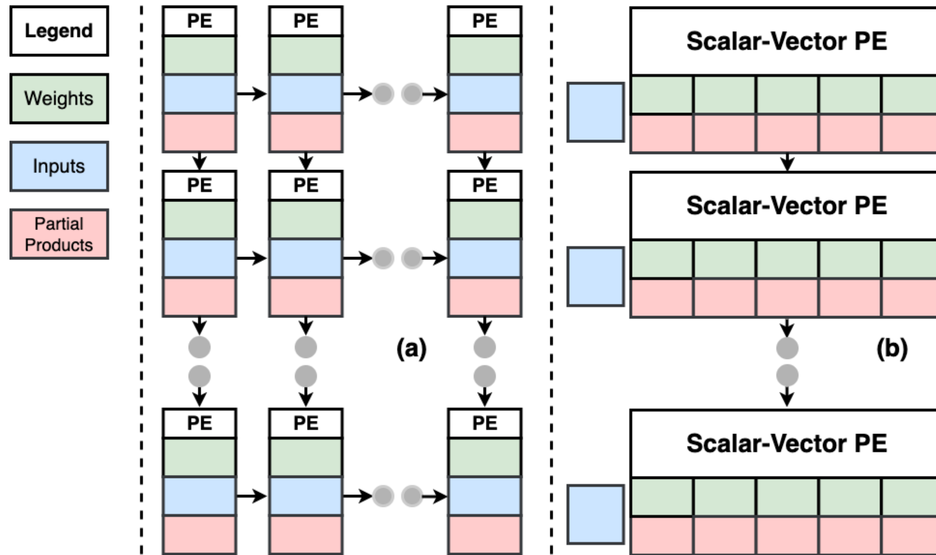

Figure 5.1: (a) Systolic Array (b) Optimized Systolic Array

input matrix is propagated along one dimension of the array. Each PE can perform a multiply-and-accumulate (MAC) involving the weight, input, and an existing product. Afterward, the input is propagated to the next PE. Across the other dimension, the product is passed to the next adjacent PE allowing accumulation of the matrix multiplication's products.

Systolic arrays work particularly well when the input is a matrix with a large number of columns. Systolic arrays are optimal when a continuous stream of inputs is available to ensure high PE utilization. In practice, a continuous stream is not possible due to limitations on batch size and latency requirements. Batches can only be so large before exceeding storage space and violating latency requirements. With data-bound applications, new weight tiles being loaded will eventually cause the array to stall.

The weight stationary systolic array can be further optimized to reduce data propagation and latency. A scalar-vector PE can be used to replace a row of PEs that propagates the inputs from one PE to another. The scalar-vector PE unit merges the PEs in a row into a single PE that shares an input. Figure 5.1 shows a typical systolic array on the left and our optimized implementation on the right. Previous work [96] [97] has created a systolic tensor array with multiple PEs across a row and column sharing operand buffers and operating in parallel.

5.2.2 Dataflow Architecture

Eidetic has a flexible dataflow architecture that is able to map multiple operations or layers on-chip. Each operation includes a reference to its corresponding inputs and outputs. *Eidetic* uses these references to match outputs of an operation to inputs of other operations and facilitates the data movement between them. Unconnected outputs are sent back to the host. This allows *Eidetic* to take a model broken into nodes, and map as many supported nodes that can fit on-chip, with any remaining nodes being handled by the host or a co-processor.

Operation mapping and connections are predefined before execution. Additionally, weights are preloaded. This allows our data-flow architecture to significantly reduce the number of instructions that are sent from the host. After the setup period, the host only needs to feed inputs to *Eidetic* with a corresponding identifier to uniquely identify each input. Removal of host instructions also makes way to implement more real-time processing. Inputs can be sent to *Eidetic* the moment they become available, without waiting to group inputs into a batch or to generate instructions for the inputs.

Our data-flow architecture makes *Eidetic* an ideal accelerator to pair with a host as a co-processor, or even be integrated into a system with other accelerators. The reduction/elimination of the need to transfer weights, instructions, and intermediate results can greatly improve the performance of data-bound applications (detailed evaluation in Section 5.7). Further, it will help reduce bottlenecks arising from frequent host interactions.

5.2.3 Dependencies and Layers

Neural network models can be broken down into directed graphs with each node representing a core operation. The nodes can use feedback in the case of recurrent networks, creating a cyclical directed graph. A variety of frameworks with varying paradigms exist for implementing neural networks, ranging from TensorFlow dataflow programming to JAX [98] functional programming. Regardless of the programming paradigms, most neural networks can be described with the directed graph abstraction.

Depending on the operations, *Eidetic* can map a single operation or a group of operations as a single node. We refer to any singular or grouped operations as a layer with the inputs and outputs connections as dependencies. Each layer is assigned a node ID, with each unique output/input given an edge ID. Additionally, a input index is used to differentiate each new input. Lastly, a temporal index is used to support inputs with various time steps. Both layers and dependencies are referenced with their temporal and input index as well as their node ID and edge ID respectively.

Figure 2.5 illustrates the computation graph for two dimensions of a modified GNMT encoder. Each box represents a layer composed of an operation(s) in the network. Each arrow represents a dependency or edge that connects the outputs of one LSTM cell or node to the inputs of another. Vertically in the graph, each cell has a unique node ID, and each unique data output from a node has a unique edge ID. Moving horizontally introduces a time step. Some nodes will use edges that have a different time step introducing horizontal data movement. Finally, this 2-D graph would be replicated for each input with a unique input index. The replicated instances of the 2-D graph have no connections between them and are independent of each other.

5.2.4 Concurrent Execution of Pipelined Layers

Two execution flows were considered for *Eidetic* design; a *layer-by-layer* execution and a *concurrent execution of pipelined layers*. The layer-by-layer execution is analogous to how most CPUs, GPUs, TPUs, and many other accelerators operate. However, some work [99] has performed pipelined inference on CPUs/GPUs. The weights for a layer/tile will be loaded on-chip, and a batch of inputs would be run through the PEs. Upon completion of the batch, the next layer can be started. In contrast, concurrent pipelined execution partitions the PE resources so that each layer mapped to the chip is assigned to its own subset of PEs. This allows multiple/all layers to be executed concurrently. In both approaches, weight loading can be skipped as *Eidetic* stores multiple layers/weight tiles on-chip. Changing the execution flow from layer-by-layer to concurrent pipelined changes how the data is mapped. A layer-by-layer approach would maximize the number of PEs that are used to store weights for each layer. Extra space in each PE would be used

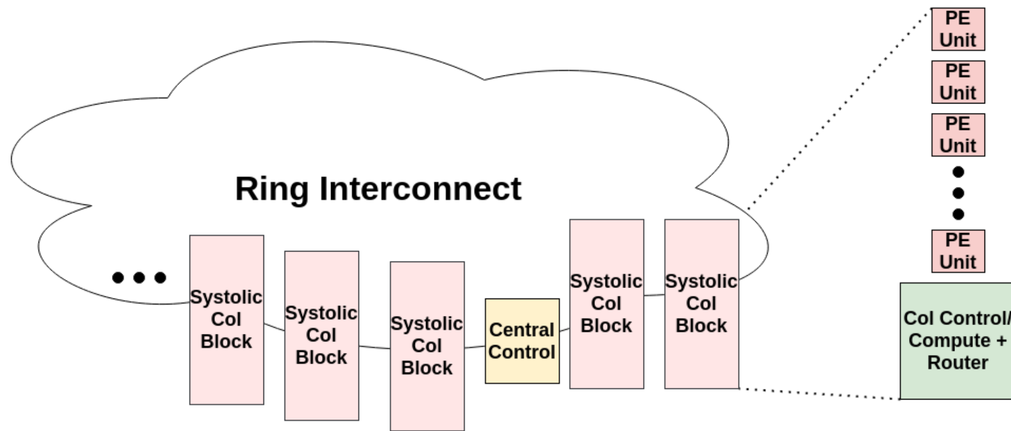

Figure 5.2: Eidetic Overview

to map additional layers. By spreading a layer out across more PEs, the layer gains more computation resources. On the other hand, concurrent execution seeks to map each layer to as few PEs as possible. The additional PEs are then able to map additional layers, allowing multiple layers to be executed simultaneously.

The layer-by-layer approach fits with the traditional batch model. Neural network accelerators typically group inputs together in a batch. Despite frameworks being designed for batches of inputs, on-demand input processing has several advantages over the traditional batch approach. Firstly, streaming inputs can improve both inference latency and system latency. With a batch, each input is forced to wait for the entire batch to finish before moving on. In networks with sequences of inputs, the penalty can be more pronounced, as the batch is constricted by the longest sequence in the batch. With on-demand processing, inputs can be moved to the next layer as soon as they finish, and downstream resources become available. This will reduce the latency of most inputs. Further, waiting for a full batch of inputs from the host side will delay the start of computation. If the input can be started the moment it becomes available, it will reduce the average and tail latency of the system.

5.3 Architecture

Eidetic is a `matmul` accelerator that utilises systolic arrays of processing elements composed of compute-capable SRAM. Combining compute and storage into a single unit allows for an increased amount of compute and storage that is available on-chip. The nature of SRAM computing allows for data to be stored where it is operated on. To minimize the amount of on-chip data movement and reduce complexity, the compute SRAM PEs are organized in a system of systolic arrays that utilizes a weight stationary data flow. Figure 5.2 shows the different components of the *Eidetic* architecture.

We now provide a high-level overview of the different components of *Eidetic*.

(1) Compute SRAM-based PEs. *Eidetic* PEs are in-memory computing SRAMs. Every PE can perform scalar-vector multiplication and accumulation needed for matrix multiplication. Each PE can operate on a 256-element weight vector. The PEs can hold additional weights in addition to the weight set it is currently operating on. Additionally, the data for the dependencies can be stored within the PEs.

(2) Column Block which contains the systolic array matrix multiplication engine of compute SRAM PEs, element-wise compute engines, intermediate storage, and block controller.

(3) Central Control Logic which tracks the dependencies of each layer and creates and issues the layers. It ensures that layers are only issued when all dependencies are available. Further, it tracks the address of each dependency and generates fetch instructions to move the dependency from its storage location to its needed Column Block.

5.3.1 PE Unit

Figure 5.3 shows the composition of each PE Unit in *Eidetic*. Each PE consists of a 9.5 KB modified compute SRAM array, a control FSM, an input latch, and a data buffer. The PEs are then organized into a systolic array.

Compute SRAM Array: Our SRAM contain a modified version of the 8KB compute SRAM

arrays used in Neural Cache [90]. The modified SRAM arrays have bitline peripherals that allow us to perform in-place computation. Our SRAM PEs are only used to perform matrix multiplications. Thus, these SRAMs have been modified to only support scalar-vector multiply and vector-vector accumulation over the ability to perform a vector-wise MAC or other operations.

The length of the bitlines in the SRAM arrays have been extended from 256 to 304, increasing the number of wordlines in an array and bringing the SRAM to 9.5 KB. Extending the bitlines has two benefits: *firstly*, storage density is increased, *secondly*, it allows networks to be mapped more efficiently. The extra 48 wordlines are used to hold products and perform computation. Many networks' dimensions are powers of two, thus leaving 256 wordlines for weights allows better mapping than the 208 wordlines without the extension. Additionally, the SRAM arrays use 4x1 sense amp column muxing and 1x1 write driver muxing.

Input Latch: In previous compute SRAM works [90, 95], the SRAMs performed element-wise operations, with an element from both of the operand vectors in each bitline. However, in *Eidetic*, every weight across the bitlines in the SRAM will eventually be multiplied by the same inputs. Instead of propagating the input across bitlines, it is more efficient to have every bitline use the same input. If every bitline is using the same input, the input can be pulled out of the SRAM and stored in a specific input latch, eliminating the need to write the input to every bitline in the SRAM and also saving space in the SRAM for more weights. Thus, every PE has an input latch.

FSM: Each PE also has an FSM attached to it. The FSM controls data propagation from one PE to the next, controls the input latch, and activates the correct wordlines to perform a bit-serial MAC.

Buffer: At the bottom of each SRAM is a double-buffer. The buffer enables the SRAMs to start moving products as soon as it creates them, while also accepting products from the buffer above it, as soon they are available and the PE has a free slot for it.

PE MAC: Similar to prior in-SRAM computing works, weights are stored in the transposed format in SRAM arrays. With the transposed data format, each 16-bit weight occupies 16 wordlines and each PE array can hold 256 weights across the bitlines. Each PE has 256 wordlines and

is able to accommodate up to 4096 elements of the weight matrix. The PE can be configured with the number of weight sets in use. The FSM will perform a scalar-vector MAC on a set of weights across the bitlines before cycling to the next set of weights across the wordlines. Additionally, the PEs can be configured to keep the product accumulating it with the MAC from the next weight set, or the product can be immediately propagated to the next PE.

5.3.2 PE Memory Addressing

In order to facilitate easy addressing and mapping, we provide the *storage slot* abstraction which is the smallest addressable granularity in the *Eidetic* architecture. One storage slot consists of 16 rows of 256-bit wordlines in the PEs SRAM in a systolic array. This amounts to 0.5 KB of storage.

Every PEs SRAM has 19 storage slots. 3 of these are reserved for computation and the remaining 16 can be used for weights or data. Storage slots are addressed with a 19-bit value. The first 6 bits are used to address the specific systolic array column. The next 5 bits address the storage slot in the PE. And the last 8 address the PE in the systolic array.

Fetching/Storing Slots: A fetch or store request is sent to the top of the systolic array where it is propagated down to the correct PE. In the case of a store, the data is also sent to the top of the systolic array and propagated. Similarly, a fetch propagates the data downwards from where it is stored. Data from the storage slots can be fetched at a smaller granularity than a single slot. A fetch request can additionally include the number of rows to fetch, and starting row within the slot. The latency for fetching/storing slots is 256 cycles plus transfer time to the column block which is low as the storage columns blocks can be placed next to blocks producing the outputs. This is less than the average number of cycles for a MAC.

Column Control: In addition to the PE, each column has a column control unit which is used to facilitate data movement between columns for accumulation and to execute micro-operations.

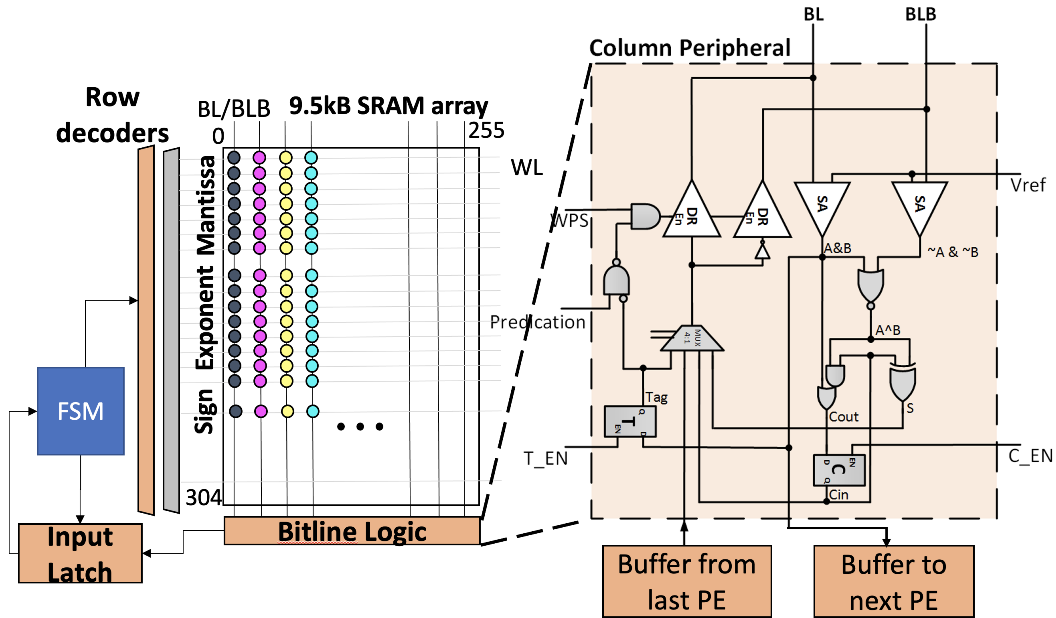

Figure 5.3: SRAM-based *Eidetic* PE Unit

5.3.3 Column Block

The Column Block is where all computations take place, and all data is stored. The Column Block is composed of the 1-D systolic array of PEs, the storage slot FIFO, the block controller, the intermediate storage and element-wise compute units. The Column Block is also responsible for performing all non-matrix multiplication operations, such as element-wise multiplication and addition and other non-linear operations. Additionally, the Column block is responsible for some data movement, sending inputs to the systolic array and moving some outputs to other columns. The operations are repeated over and over for each set of outputs from the systolic array, for each set of inputs to the systolic array, and for each set of intermediate results from other columns. Figure 5.4 shows the details of the Column Block logic.

Systolic Array: The systolic array consists of 256 PE arranged in a column. Each PE is directly connected with a 256-bit bus to the PE below it. The buffer from the top PE sends data downwards to the bottom PE SRAM. This allows products to be passed down and accumulated through the Column Block.

Instructions Circular FIFO: All instructions for intermediate operations are programmed into

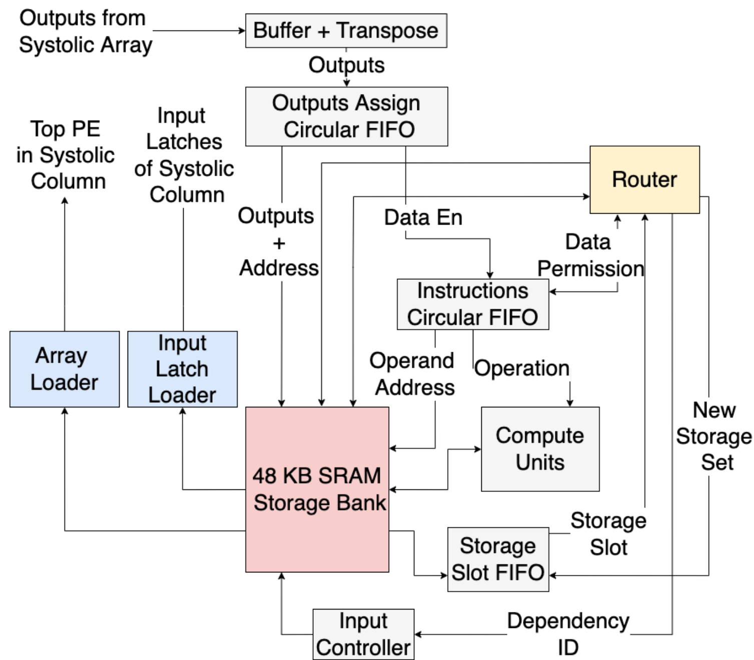

Figure 5.4: Column Block Logic Overview

this FIFO. These instructions contain the operation to perform, the addresses of the two input operands as well as the output address. Additionally, the instruction has a read stall for each operand to indicate if it hasn't been generated yet. Additionally, the instruction FIFO can contain instructions to move data to another Column Block. In addition to a read stall, the move instruction also has a write stall in order to prevent a Column Block from overwriting another column that is still working on its data.

Outputs Assignment Circular FIFO: The systolic array produces outputs in fragments of 256 elements. A single output can have up to 16 (i.e. the maximum number of weight slots) of these fragments. These output fragments are written into a circular FIFO that contains up to 32 entries with intermediate storage addresses. This allows it to store two sets of outputs.

Input Controller: Fetching of inputs needed for the systolic array is handled by the input controller. Depending on the mapping, each systolic column can use up to 256 different inputs. As a result, each input is collected in smaller segments, with the exact size determined by the data mapping. The inputs for each `matmul` will be stored in predetermined locations in the intermediate storage. When the input loader has space, the controller will load the input from the intermediate

storage and if the said input has no more reuse, it can start fetching a new input to fill that spot.

Intermediate Storage Bank: Every Column Block has a storage bank for intermediate results and inputs that need to be loaded.

Storage Slot FIFO: Each Column Block contains a FIFO containing free storage slots' addresses. The FIFO receives addresses from the central control logic.

5.3.4 Central Control

The central control logic is used to generate layer IDs from output dependencies and track the storage slots' addresses of the dependencies. It controls when layers can be issued, when dependencies need to be fetched, and to where, and sends free slots to the Column Blocks. The central control does not hold or operate on any data.

The relationship between dependencies and layers can be described as parents and children. Each dependency will have a variable number of child layers that need this dependency to issue. Each of these child layers needs a variable number of parent dependencies to issue. Figure 5.5 shows the flow diagram of the central control module.

The central control can be split into 2 separate parts; Layer ID Generation and Issuing, and Storage Slot Tracking. Whenever an output is created by a Column Block, the Column Block sends the edge ID of the dependency it created as well as the storage address of where it is being stored.

5.3.4.1 Layer ID Generation and Issue

The central control only receives the edge ID of each intermediate output dependency that is generated from a Column Block, while the temporal index and the input index are missing. The full dependency ID consisting of the tuple (edge ID, input index, and temporal index) is obtained from the *Output Dependency Lookup Unit*. Every intermediate output that is generated can be an input dependency for several other child layers. Using the dependency ID of an output, the *Layer ID Generator* generates the child layer IDs for the layers that use that output. Each of the child layer

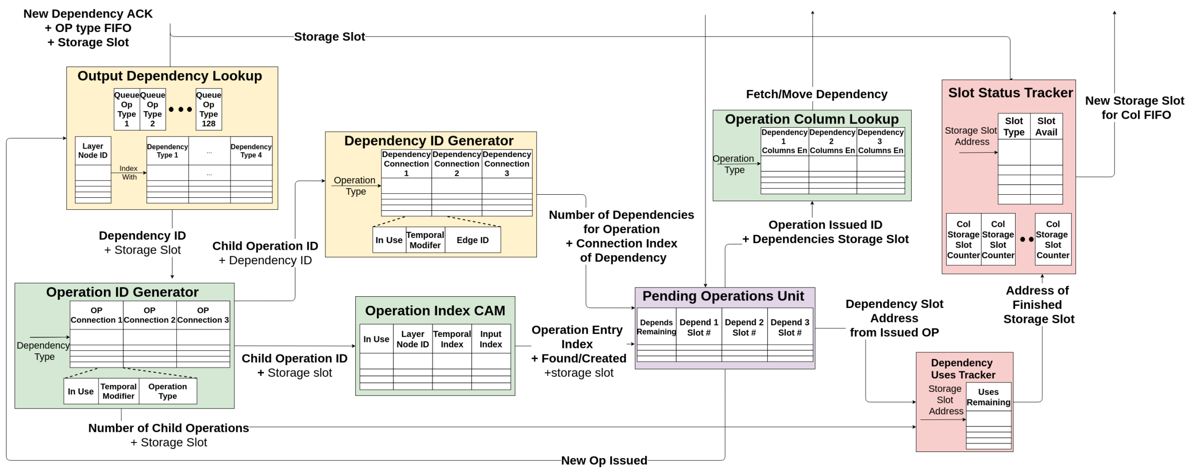

Figure 5.5: Control Flow of Central Control Unit. Bold text between units indicates the passed information was created in one of the units.

IDs generated can now be tracked along with the storage slot address of the parent dependency. In order to issue a layer, the central control needs to ensure all of its dependencies have been created and it needs to track the storage slot address of each dependency. Whenever a new layer ID is generated the central control needs to determine if the layer is already being tracked. It also needs to determine which of the possible parent dependencies generated the layer ID so the storage slot address can be matched to it. This is handled concurrently by two units: the *Dependency ID Generator* and the *Layer CAM Index*. The *Pending Layer Unit* keeps track of the dependencies remaining for each layer ID before it can be issued. If the layer is issued, the layer ID as well as the dependency storage slot addresses where its output is stored are sent to the *Layer Lookup Unit*. The unit can then generate fetch instructions to retrieve the generated outputs and move them to the correct Column Blocks.

5.3.4.2 Storage Slot Tracking

Each Column Block has a FIFO with the storage slots reserved for the outputs it will create. The central control is responsible for tracking which storage slots are open and sending available storage slots' addresses to the Column Block FIFO. The storage slot for each generated output dependency must be kept until all of its child layers have been issued. In addition to its other functions,

the Layer ID Generator also sends the number of child layers that consume each output dependency to the *Dependency Uses Tracker* unit. Every time a Column Block finishes using an input, it sends the slot number to the central control allowing the slot number for that table entry to be decremented and be made available if the *uses remaining* reaches zero. Upon reaching zero, the storage slot status can be updated in the *Slot Status Tracker* unit described below.

The *Slot Status Tracker* stores both the type and availability of every storage slot. Storage slots can have different types based on whether they store weights or intermediate outputs. Weights slots are reserved during setup. In addition, this unit has a counter for each Column Block, tracking the number of elements in their FIFOs. The unit also allows contiguous storage slots to be grouped together to create a larger data slot for intermediate results. Without grouping, each slot would need to be tracked as a separate dependency, vastly increasing the number of dependencies tracked per layer.

5.4 Data Mapping and Eidetic Instructions

5.4.1 Data Mapping

5.4.1.1 Weight Matrix Mapping

Matrix multiplication in a layer is performed by mapping the weight matrix onto the systolic array. However, most networks will have weight matrices that are too large to fit in a single systolic array. Instead, the matrix multiplication is broken up across multiple systolic arrays with accumulation and/or concatenation performed outside the systolic array. Figure 5.6 illustrates how matrix multiplications are mapped to systolic arrays with a very small layer and shows the 3 granularities of the mapping. The example shows miniature systolic arrays, each with 2 bitlines in a PE, 3 PEs in a systolic array, and 2 weight storage slots in a PE. The weight matrix can be referred to with the dimensions (k,m). With matrix multiplication, partial products along the "k" dimension are accumulated whereas along the "m" dimension they are not. The different indices in the weight

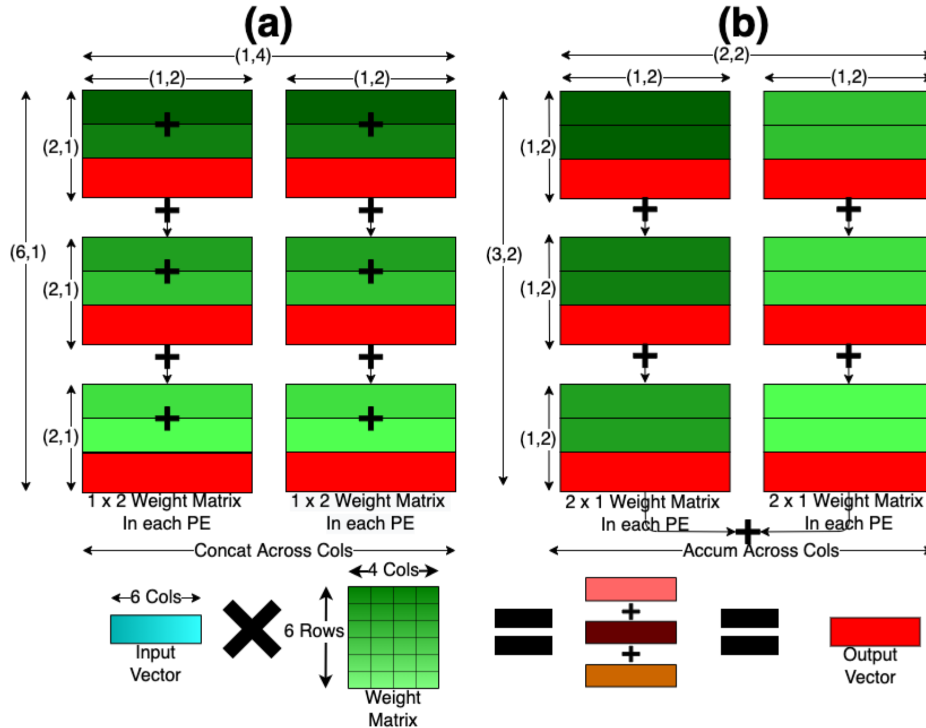

Figure 5.6: Mapping of weights to a systolic array

matrix are highlighted using different shades of green. We use the same color coding to show how each element of the weight matrix is mapped to each PE.

(1) Bitline Mapping: Across 2 bitlines in a PE's SRAM. The "m" dimension of the weight matrix is mapped across the bitlines in a PE's SRAM. The bitlines have no communication between each other and elements across the weight matrix columns are not accumulated.

(2) PE Column Mapping: Across the 3 PEs in the systolic array. The "k" dimension of the weight matrix is mapped along each PE. This is because partial products are easily accumulated as they are passed down from one PE to another.

(3) PE storage slot Mapping: Finally, each PE's SRAM has 2 weight storage slots where elements are stored across the PE's wordlines. This granularity can hold elements from either the "k" or "m" dimension. By mapping "k", each PE will perform 2 MACs before propagating the partial product. Conversely, if "m" is mapped, each PE will perform a single MAC and propagate the partial product.

The bitline mapping and PE column mapping policies are fixed in their weight dimension

mapping. The flexibility in the PE storage slot mapping is shown in Figure 5.6. In Figure 5.6 (a), the systolic arrays have the "k" dimension of the weight matrix stored in the PE storage slots. In this example, the outputs of the systolic arrays must be concatenated together. Conversely, in Figure 5.6 (b), the "m" dimension of the weight matrix is stored in the PE storage slots. Accumulation needs to be performed across the 2 systolic arrays.

5.4.1.2 Data Mapping Flexibility

Eidetic ability to map either dimension of the weight matrix across the PE storage slots increases the efficiency of mapping the weight matrix with a range of shapes and sizes. With 256 bitlines in a PE, 256 PEs in a systolic column and 16 PE storage slots in a PE, *Eidetic* can map a [256x4096] or [4096x256] weight matrix to the systolic column. However, the 16 PE storage slots can be split into both dimensions, allowing any weight matrix between [256x4096] or [4096x256] such as [1024x1024].

How weights are mapped in the weight storage slots has a large effect on the number of inputs needed to keep a systolic array saturated. By performing accumulations in the weight storage slots Figure 5.6 (a), no input segment is reused for a PE and each PE will be using an input with a different input index. However, the alternate mapping Figure 5.6 (b) will result in the same input segment being reused. While the input segment is being reused, the next PE will be using a different input segment of the same input index. The greater the time that an input segment is reused, the greater the number of PEs that will be concurrently sharing the same input index, reducing the number of inputs needed for saturation. The number of inputs needed for saturation is: (Number of PEs – 1) / (Input Reuse) + 1

Both Figures 5.6 (a) and (b) describe a packed mapping scheme where the amount of weights stored in a systolic array is maximized. If the network does not utilize all the systolic arrays, some matrix multiplication operations can be spread out across more systolic arrays. This is particularly useful when a layer requires more concurrent inputs to maintain saturation than the other layers. It is also useful for matrix-matrix multiplication. In the first case; consider a layer with a weight

matrix of dimension [6x4] that maps to the systolic array in Figure 5.6. With a packed mapping, the matrix multiplication occupies only one systolic array, and thus there is no flexibility in the weight storage slot mapping. This mapping requires 3 different inputs concurrently to keep the systolic array saturated. If it is anticipated that not enough inputs will be available, the matrix can be spread out across multiple systolic arrays. This increases the amount of compute for the layer decreasing its saturation requirements. In the other case, an unbalanced matrix-matrix multiplication can be balanced by spreading out across more systolic arrays. This can prevent such a layer from becoming the bottleneck due to a lack of computation. In both cases, the storage slots in the systolic arrays will be underutilized. These unused slots can be used to store data intermediates instead of storing them in a separate systolic array.

5.4.1.3 Other Operations

While the bulk of the focus is on matrix multiplication, *Eidetic* also allows mapping of other operations to reduce the need to go off-chip. Inside the column block, separate from the systolic arrays, are more compute-based SRAM and other compute units. Other operations needed are mapped to these units.

5.4.2 Instructions

Eidetic is programmed using 3 categories of instructions. *Central Control Setup Instructions*, *Column Block Setup Instructions* and *Input Loading Instructions*. *Eidetic* is designed to use setup instructions to define the model and allows for inputs to be processed continually without additional instructions. *Eidetic* does not have an explicit `matmul` instruction called for two operands. Rather the user defines all the connections in the model. With the model defined, inputs are sent in and are handled entirely by the control logic without additional user instructions.

5.4.2.1 Central Control Setup Instructions

The central control's primary focus is facilitating data movement for the column blocks by determining when a layer's operations can be issued and tracking the dependencies for the layer. The central control accomplishes this by tracking the storage location of all dependencies as well as tracking when a layer has all of its dependencies available.

Weight Loading: This instruction is used to load weights into a PE's storage slot. Weights are sent from off-chip in 512 B increments. The user provides the storage slot address and the weights to be stored there.

Layer to Dependencies Connections: This instruction is used to define the dependencies that are needed for each layer. The connections for each node ID are defined. A connection consists of an edge ID and a temporal modifier. Up to 3 connections are available for each node ID.

Dependency to Layers Connections: This instruction is the inverse of the Layer to Dependencies Connections. Each dependency type has 3 connections to the layers that use it.

Layer Outputs: Every layer can create up to 4 different dependencies as an output. The edge ID of each possible one is defined in this instruction. The output of a layer has the same temporal index and input index as the layer, so these values are not defined in the instruction.

Layer Dependency Columns: This instruction defines which column blocks need to receive a layer's dependencies when the layer is issued.

Host Output Edge Type: All dependencies with this edge ID are sent off-chip to the host.

5.4.2.2 Column Block Setup Instructions

There are two types of column block setup instructions.

Column Matrix Multiplication Configuration: This instruction is used to enable the flexible mapping of the matrix multiplications in the systolic arrays. The instruction has two components: the number of weight storage slots and MACs per propagation. The number of weight storage slots is the number of storage slots in each PE that are storing weights. Each PE will only cycle through the number of weight storage slots indicated, rather than all 16 possible weight storage

slots. MACs per propagate are based on the weight mapping scheme used for the systolic array. Depending on the mapping, each partial product can be accumulated with 16 MACs or a single MAC or some number in-between.

Intermediate Operation Instructions: In order to perform operations on intermediates, the column block is programmed with several types of instructions such as: assigning an operand ID/address to outputs from the systolic array, performing an operation on two operands, moving an operand to another column, and giving read/write permissions to other units.

5.4.2.3 Input Instructions

Inputs can be received and segmented into elements of their temporal sequence. The edge ID, the input index, and the temporal index are included with the input.

5.4.3 Programmability Example

Below is an example of Eidetic programmability. The 4-layer network in Figure 2.5 is programmed with 6 nodes. The addition between Layers 3 and 4 gets its own nodes as it is separate from the LSTM cells. We describe the different node types below: Nodes 1-4 (LSTM), Node 5 (Add), and Node 6 (LSTM).

As shown in Figure 5.5, many of the units are driven by the instructions programmed into the table. Each LSTM node has two outputs, H_t and C_t. H_t is passed to the next time step and the next layer, while C_t is just passed to the next time step. X_t is either the initial input to the graph or the output of the Add node. The following shows the data programmed into the Dependency ID Generator and Output Dependency Lookup. The number before X, H, and C indicate the layer ID. Additionally, the Layer ID Generator contains the inverse of the Dependency ID Generator.

Dependency ID Generator:

Node1: $0X_t, 1H_{t-1}, 1C_{t-1}$, **Node2:** $0X_t, 2H_{t+1}, 2C_{t+1}$, **Node3:** $1H_t, 2H_t, 3H_{t-1}, 3C_{t-1}$, **Node4:** $3H_t, 4H_{t-1}, 4C_{t-1}$, **Node5:** $3H_t, 4H_t$, **Node6:** $5X_t, 5H_{t-1}, 5C_{t-1}$

Output Dependency Lookup:

Node1: $1C_t$, $1H_t$, **Node2:** $2C_t$, $2H_t$, **Node3:** $3C_t$, $3H_t$, **Node4:** $4C_t$, $4H_t$, **Node5:** $5X_t$, **Node6:** $6C_t$, $6H_t$

Additionally, *Eidetic* has instructions to assign nodes to systolic column blocks and configure the mapping of each Matmul. Each type of node i.e Matmul, LSTM, Add would generate instructions stored in the Col Control.

5.5 Putting It Together

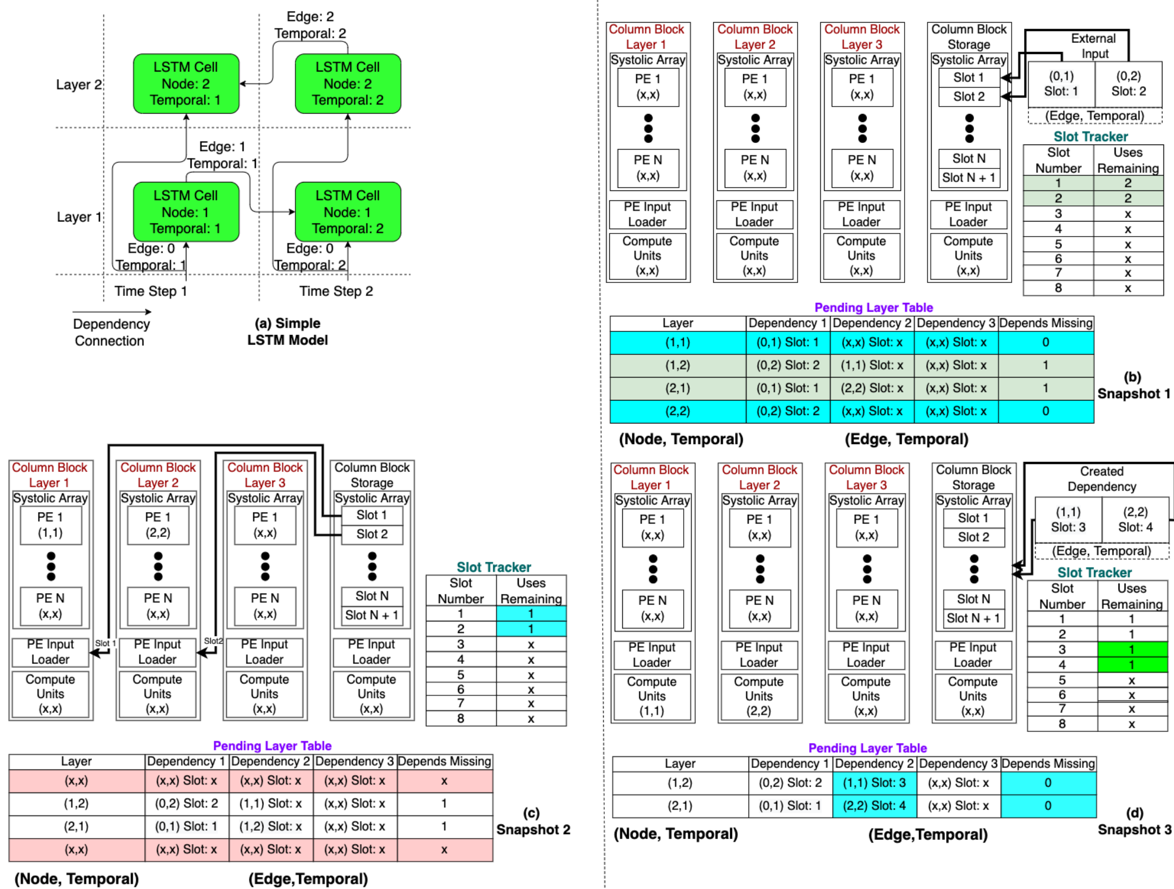

Figure 5.7: Putting it together

Figure 5.7 (a) shows an example of a very small and simple LSTM, showing how layers are issued and executed. In this example, a 2-layer network with a pair of bidirectional layers is shown with a single input for two time steps (i.e., with a temporal width of 2). The network has been

simplified to have a single output dependency that connects across the layers and time steps.

Snapshot 1: Two inputs that are a part of the same sequence are fed in and stored in free slots in a systolic array that has been reserved for storage. Both elements generate two child layers each. Two of the 4 generated layers have zero missing dependencies and can issue. Layer 1, the forward layer, starts at the first time step in the sequence, while layer 2, the backward layer, starts at the end. The slot tracker records the number of child operations involved for each input. This number is recorded as Uses Remaining.

Snapshot 2: The two layers ready to issue are removed from the pending layer table. The data stored in the slots needed for each layer are fetched and sent to the PE input loader. The slot tracker decrements the uses remaining for that slot. Execution of the two layers begins in their respective column block.

Snapshot 3: The matrix multiplication for the first input sequence has finished for both layers while the matrix multiplication for the second input is starting. The layer execution has moved onto the compute units where all non-matrix operations such as non-linear activations are performed. The output of each layer produces the missing dependency for the same layers at the next/previous time step.

5.6 Evaluation methodology

***Eidetic* Performance Model:** We developed a cycle-accurate simulator in C++ to model the systolic array execution. The simulator creates systolic array PE structures capable of simulating each cycle in a bit-serial MAC to get accurate utilization estimates for the PEs. The simulator overestimates the cycles by counting the cycles for the longest MAC across every PE on-chip. The simulator also tracks the number of dependencies that have been created and are waiting to be used. A delay of 300 cycles is added to account for the non-matrix operation, storing and fetching data dependencies, and issuing instructions from the output dependencies. The setup instructions are used to program the layer connections and load weights into the systolic arrays. Then the simulator

is fed the input trace to begin execution. The simulator provides us with cycle-accurate profiling needed to evaluate *Eidetic* performance. It provides utilization of *Eidetic* at multiple levels of granularity, the throughput achieved and the amount of storage that is needed.

Validation: The GNMT encoder model was taken from MLPerf [100] and the inputs, weights, and outputs to each layer were collected. The inputs and weights were then run through our *Eidetic* simulator. The outputs were matched to ensure accuracy. The Verilog design for the central control was further verified by using the same input trace for our simulator and matching the output traces.

Frequency Estimation: At the baseline frequency of 4 GHz for the SRAM arrays [101] with 8:1 sense amplifier multiplexing, during a read cycle, ∼75% of the cycle is spent precharging the bitlines. The remaining ∼25% is used in sense amplifier cycling [90]. Additionally, on write cycles, ∼60% of the time is used to perform writebacks. We chose 4:1 multiplexing for sense amplifiers and 1:1 multiplexing for write drivers as it created the best balance between frequency/throughput and area.

Area Estimation: To estimate the area of our custom SRAM array, we used previously published numbers from Intel 22nm SRAM [101], scaled to 9.5 KB and accounting for the area of an additional decoder needed for In-SRAM computing. Sense amplifier and write drivers account for 25% of the area of the SRAM array, with the remaining I/O area dedicated to the decoder. A write driver has the same area as ∼40 bitcells. The FSM, Input latch, and Buffer, Central Control, and Column Block Control were written in Verilog and synthesized. All area estimates are scaled to 22 nm. The Sigmoid and Tanh functions are implemented with LUT [102].

Power Estimation: To obtain the SRAM dynamic energy for read, write, and computation, we used previously reported estimates from Neural Cache [90]. The SRAM read/write cycles take 8.6 pJ while computation cycles take 15.4 pJ. For leakage energy of the SRAM arrays, we use reported numbers from [101]. Scaling these estimates for the increased bitline length (304 vs 256) and reduced frequency of *Eidetic* SRAM arrays, we estimate the energy for read/write to be 5.67 pJ and computation to be 8.78 pJ.

TPU Profiling: For comparison we ran the same GNMT model on TPUv2. Our experiment

used 1 board, which consists of 4 chips and 8 total cores. To measure the throughput and latency of the chip, we ran 102,400 inputs in various batch sizes from 8 to 1024. For our experiment, we gave all inputs in a batch the same sentence length. This is ideal for running on TPU, however, it is not always possible given the variability in workloads.

Additionally, we ran several ML microbenchmarks on TPU v2. The batch size was increased until the throughput was saturated. On *Eidetic* the benchmarks were replicated across multiple systolic columns where possible.

5.7 Results and Analysis

5.7.1 Performance

The major focus of our evaluation is demonstrating the acceleration of GNMT's encoder. The version we implemented is taken from MLPerf [100]. Additionally we compare *Eidetic* and TPUV2 to several ML microbenchmarks.

5.7.1.1 Latency and Throughput

Figure 5.8 shows how *Eidetic* compares to TPU for throughput of GNMT. Additionally, the average latency is also shown in Figure 5.9. It is important to note that for both throughput and latency, *Eidetic* uses an input batch size of 16, whereas the TPU uses a batch size of 1024 for throughput and 8 for latency. 1024 was the max batch size tested for TPU and gave the best throughput. Further 8 is the minimum batch size for TPU and provided the best latency. An input batch size of 16 was used for *Eidetic* because it is the sweet spot for both throughput and latency. As shown in Figure 5.10, *Eidetic* offers minimal improvements in throughput past an input batch size of 16. Additionally, latency remains mostly constant up to an input batch size of 16 and then increases linearly afterward. An Input batch size of 16 is enough to saturate the systolic arrays. As previously mentioned that batch size refers to the size of a streaming window of inputs that drop out after the last temporal step starts inference.

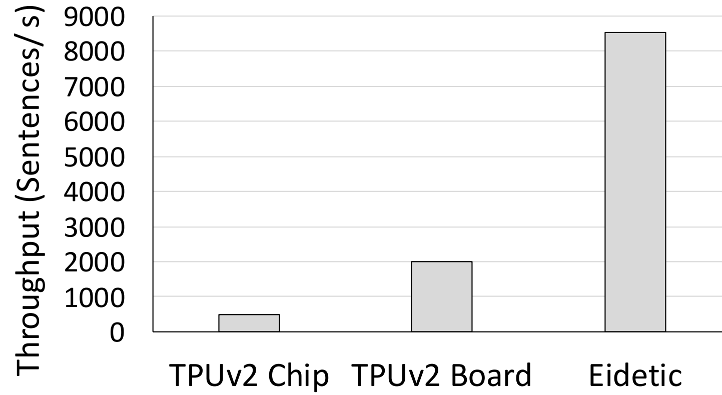

Figure 5.8: Throughput of TPU Batch Size 1024 vs *Eidetic* Input Batch Size 16

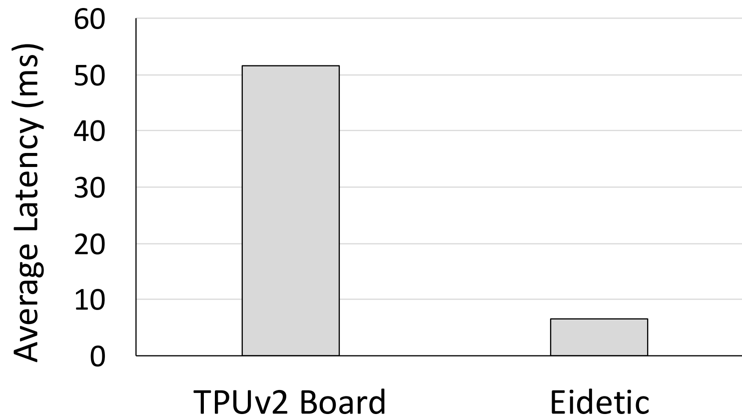

Figure 5.9: Latency of TPU Batch Size: 8 vs *Eidetic* Input Batch Size: 16

5.7.1.2 Utilization

Eidetic PEs can be configured to store data or compute. PE compute utilization is the percentage of arrays that are actively doing compute. Allocating PEs for data storage will result in reduced utilization. Additionally, some compute arrays can become inactive without any inputs. One factor is the warm-up and warm-down penalty of the systolic propagation and layer propagation. Each PE in a systolic column must wait for the PE above it to perform a MAC before it can. Additionally, inputs with different sequence lengths can cause minor imbalances that reduce the utilization. On-demand processing reduces this effect but does not eliminate it. *Eidetic* running GNMT has 81.5% of PEs configured for computation. Of those PEs, 94.6% of them are active during inference for a total utilization of 76.2%. Accelerators such as TPU struggle to have even a modest utilization

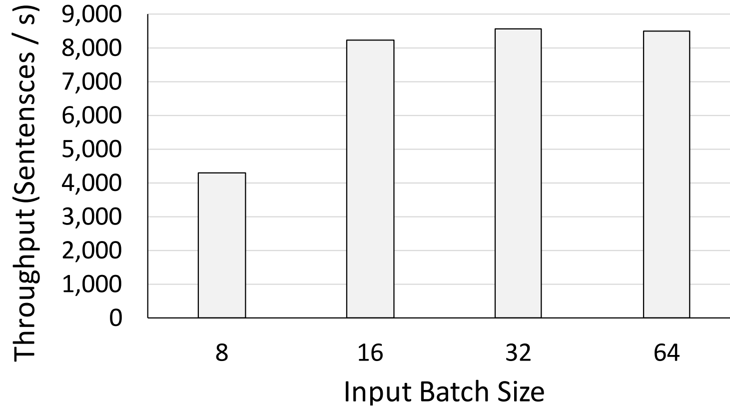

Figure 5.10: *Eidetic* throughput by input batch size

Component	Sub-component	Area (mm^2)	Group Area (mm^2)
Systolic Column Block	Compute SRAM	185.05	234.43
	PE Overhead	37.12	
	Control Logic	12.26	
Central Control	Control Logic	0.23	2.57
	SRAM CAM	0.61	
	SRAM TABLE	1.73	
MISC	I/O		24.45
Total	-	-	266.40

Table 5.1: Area breakdown of *Eidetic* by component. There are 54 Systolic Column Blocks each with 2.375MB of compute SRAM

for RNN and MLP inference. GNMT inference results from TPUv1 show an extremely low useful MAC as only 8.2% of cycles. Weight stall and weight shift cycles take up 58.1% and 15.8% of the cycles respectively, for a total time percentage of 73.9% of the cycles stalling for weights. With the *Eidetic* design, weights can be preloaded and pinned in place, eliminating the need to spend time in weight stall and weight shift for inference. Instead, weight stalls/shifts would be performed during setup time.

5.7.2 Energy and Area

The area breakdown for the different components in *Eidetic* is shown in Table 5.1. The vast majority of the area is dedicated to the Systolic Column Block, with the Compute SRAM arrays being the main component. The total area for *Eidetic* is 256.48mm^2. Additionally, the power breakdown is also shown in Figure 5.11. SRAM consumes power in 3 ways: performing compute, moving

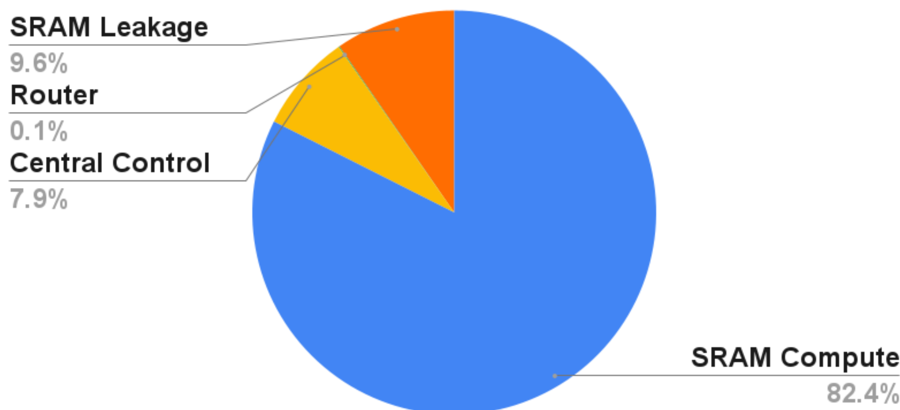

Figure 5.11: Power Breakdown

Accelerator	Eidetic (this work)	Google TPUv3 [8]	Alibaba Hanguang 800 [103]	Groq ThinkFast TSP [14]	Neural Cache [90]
Technology	22nm	12nm	12nm	14nm	22nm
Precision Supported	Versatile Precision	fp16, fp32	int8, int16, fp24	int8, fp16	int8
Frequency (GHz)	1.5	-	0.7	0.9	-
Area (mm^2)	266.4	648.0	709.0	725.0	268.0
TFLOPS (fp16)	41.8	123.0	-	410.0	-
TOPS (int8)	175.5	-	825.0	820.0	28.0
POWER (W)	235.7	450.0	275.9	300.0	52.9
KB/mm^2	481.4	18.0	-	110.3	130.6
GOPS/mm^2	659.0	-	423.1	411.2	104.4
GOPS/mm^2/W	2795.6	-	1022.3	913.9	1974.5
GFLOPS/mm^2	156.9	69.0	-	205.6	-
GFLOPS/mm^2/W	665.8	102.2	-	456.9	-

Table 5.2: Area and power normalized throughput comparison for different accelerators. For comparison, the metrics in the last five rows are normalized to 22nm using DeepScaleTool [2]. For TPUv3 and Hanguang 800, we conservatively assume reported numbers are in 14nm for normalization to 22nm.

data, and from SRAM leakage. SRAM compute dominates the power profile at 82.4%. Followed by this, SRAM leakage contributes 9.6% and Central control contributes 7.9%. Other components such as Router and SRAM Data contribute a negligible fraction to the total power consumed. SRAM Data refers to the energy to move and fetch intermediates to and from array storage slots. *Eidetic* operates at a TDP of 161W and consumes 81.43W on average.

5.7.3 Throughput comparison with other accelerators

Table 5.2 shows how *Eidetic* compares to other accelerators in this space. We compared to Google TPUv3 [8], Alibaba Hanguang 800 [103] Grog ThinkFast TSP ,[14] and Neural Cache [90]. We

	Input Matrix	Weight Matrix	MB of Weights	Replications	BL ALU Util	Weight Slots Util	Eidetic Inf/sec	TPU V2 Inf/secs
Matrix Mult	[1500,2560]	[2560,5124]	25.02	3	73%	69%	556.4	142.1
Matrix Mult	[7000,4096]	[4096,4096]	32.00	3	89%	89%	113.0	23.9
Matrix Mult	[16,2048]	[2048,6144]	24.00	4	79%	89%	58575.7	13860.1
	Dim(H,W,c,r,s)	**Pad, Stride**						
Conv	[14,14,1024,1,1]	[0,2]	4.00	24	71%	89%	103785.0	25762.0
	Hidden Units	**Time Steps**						
GRU	2816.00	1500.00	90.75	1	76%	84%	15967.4	2450.1

Table 5.3: ML Microbenchmarks Eidetic Mapping and TPU V2 and Eidetic Performance.

normalized power and area to 22nm in the last 4 rows using DeepScaleTool [2]. TPUv3 and Hanguang 800 are implemented at 12nm. We conservatively assumed area and power estimates for these two accelerators to be at 14nm for the purpose of normalization to 22nm. When normalized to 22nm, *Eidetic* outperforms the next best accelerator by 1.56×, 1.89× and 1.95× for GOPS/mm^2, GOPS/mm^2/W, and GFLOPS/mm^2 respectfully. On GFLOPS/mm^2, ThinkFast TSP outperforms *Eidetic* by 1.31×.

Additionally Table 5.3 compares the performance of *Eidetic* and TPU on ML microbenchmarks [104]. Where possible, the microbenchmark is replicated on *Eidetic* to improve performance, with the bitline ALU and weight storage utilization provided. The throughput per chip speedup ranges from 3.9× to 6.5× across the microbenchmarks. *Eidetic* 17.20× speedup for GNMT can be explained by the increased amount of intermediate storage needed for the bidirectional layer, coupled with the inefficiency of variable time steps in a batch that TPU would experience.

5.7.4 Versatile Precision

Eidetic bit-serial arithmetic naively supports variable bit-precision. The FSMs that control each PE can be designed with multiple algorithms and/or with dynamic values for the bit precision allowing an extremely wide range of precision without complex hardware. Figure 5.12 shows how the number of cycles needed for a MAC changes based on the data precision used.

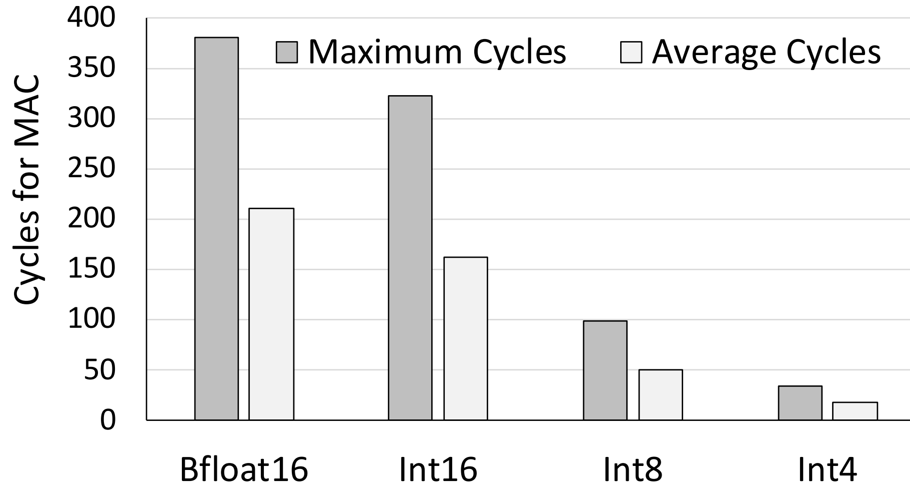

Figure 5.12: Avg. and Max Cycles for MAC – different precision

5.8 Summary

This dissertation proposes the *Eidetic* architecture, a novel dataflow architecture customized for RNN execution. *Eidetic* is designed based on the observation that RNN accelerators like the TPU spend significant amounts of time and energy in weight loading stalls and on-chip data movement. To eliminate weight loading stalls, *Eidetic* pins all weights to on-chip SRAM. To reduce on-chip data movement, *Eidetic* leverages a novel system of in-SRAM computing based processing elements, organized into 1-D systolic arrays. *Eidetic* supports execution of both low-level matrix operations as well as entire neural network models. We evaluate *Eidetic* on the GNMT encoder model and observe a 17.20× increase in throughput over a single TPUv2 chip.

CHAPTER 6

Comparison of Cache vs Custom Architecture

This chapter discusses the advantages and disadvantages of using the two in-SRAM computing architectures to accelerate different types of neural networks. How does the performance compare between the two and what factors lead to the performance gap.

6.1 Introduction

While both *Neural Cache* and *Eidetic* present in-SRAM computing for neural network acceleration, the two architectures differ in terms of costs, benefits, and use cases. *Neural Cache* repurposes an existing cache structure in a Xeon processor to accelerate DNNs, with a focus on accelerating CNNs. Modifications are made to the SRAM arrays in the last level cache to support in-SRAM computing along the bitlines. The modifications to the SRAM arrays are minimal to preserve the original cache functionality. Additionally, the original cache geometry is left intact. *Neural Cache* utilizes a layer-by-layer approach for DNN inference. A single layer is mapped across the cache, with the weights replicated as many times as possible. By replicating the weights, the required computations are spread out across more PEs. After a layer is finished the next layer is loaded in. *Neural Cache* provides a large amount of compute ability to one layer at a time and reduces on-chip data movement.

Eidetic is a custom ASIC that uses SRAM PEs as its main source of storage and compute. The SRAM PEs are arranged in a systolic column that provides easy accumulation from one PE to another. Additionally, *Eidetic* has a custom control logic to facilitate on-chip data movement

and allow for high-level dataflow programming. *Eidetic* uses concurrent pipelining as its execution model. This allows it to map multiple layers on-chip that are executed in parallel while also pinning weights on-chip. *Eidetic* spreads out its compute across multiple layers and reduces on and off-chip data movement.

The major advantages of *Neural Cache* are that it has a low cost for implementation and works well with compute-bound applications. Implementing *Neural Cache* requires minimal modifications to an existing processor cache to turn the SRAM arrays into parallel compute engines. Server processors such as the Xeon are common and relatively inexpensive. Furthermore, with the layer-by-layer execution model, *Neural Cache* is optimized for breaking through compute bottlenecks. Additional modifications to the cache would be nonviable if they could cause the processor to no longer be effective in its original workloads. Because *Neural Cache* is limited to an existing cache structure, it is unable to take advantage of more customized SRAM arrays and SRAM organization. The cache design is inherently a limitation as it can restrict the mapping of networks. Additionally, a large amount of area is dedicated to traditional compute units, limiting the total amount of SRAM compute area on-chip. This area is wasted space and is limiting especially if model sizes continue to grow.

The major advantages of *Eidetic* are in its customization and its data-bound focus. The customization allowed for more efficient SRAM arithmetic that reduces the energy and time it takes to perform computations. Additionally, the custom control logic allows for dataflow programming, reducing the amount of off-chip data movement for instructions. The concurrent pipelining execution and a larger amount of on-chip SRAM allows for the weights to be pinned in place, greatly reducing the off-chip data movement. However, the TOC is much higher for an ASIC, and thus the performance gains must be higher to justify its usage. In addition, there is a cost to adding a host-accelerator interaction. Inputs and instructions have to go through another layer, from the host to the accelerator, creating energy and delay penalties.

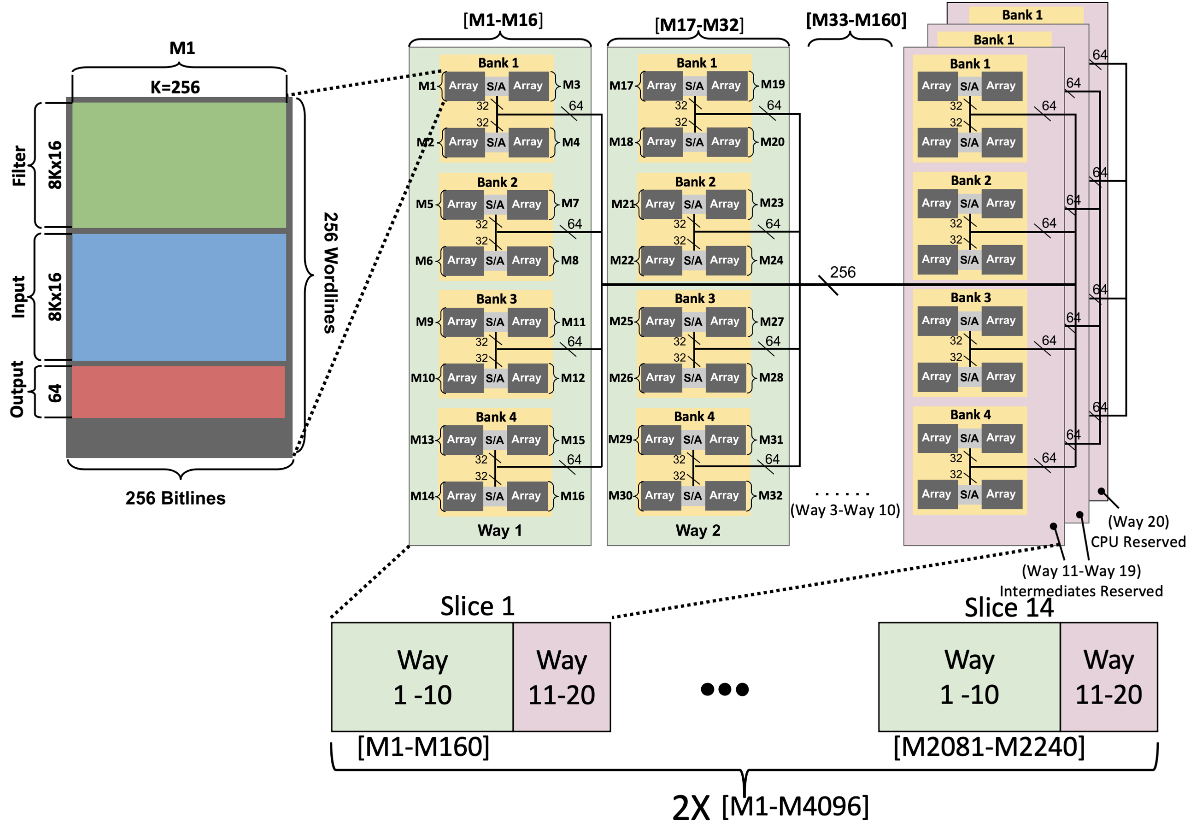

Figure 6.1: Mapping of *Eidetic* on *Neural Cache*

6.2 Data Mapping

Mapping networks on *Neural Cache* have a lower granularity due to *Neural Cache* rigid cache structure and lockstep execution. The granularity is fixed based on four SRAM arrays in a bank, four banks in a way, 20 ways in a slice (18 for computation), and 14 slices. When mapping the weight matrix to the cache structure, the amount of units needed is rounded if more than half the units in the group are needed. For instance, if 3 SRAM arrays are needed to finish mapping one dimension, all four SRAMs in the bank will be reserved, leaving one unoccupied. *Neural Cache* does not support spilling over the mapping as it would create more complexities for instructions and data movement. Each SRAM array has 256 wordlines with 48 of the wordlines reserved for compute and a partial product, and another 16 wordlines for the inputs. This leaves 12 storage slots available for 16-bit weights. We show the mapping of GNMT on *Neural Cache* in 6.1 as well as discuss why this mapping was chosen below.

Multiple weights can be stored in each bitline in the SRAM arrays. Unlike in *Eidetic*, mapping GNMT on *Neural Cache* prioritizes mapping the accumulating dimension of the weight matrix across the bitlines in the SRAM array and within the SRAM array. There are two main reasons that *Eidetic* prioritizes the non-accumulating dimension of the weight matrix across the bitlines. Firstly this mapping results in every MAC using the same input, which in *Eidetic* means the inputs can be pulled out into the input latch. Instead, *Neural Cache* keeps the inputs in the array, so having repeated inputs across each bitline doesn't give an advantage. Secondly, *Eidetic* systolic columns allow for efficient dataflow and accumulation of partial products. The fixed cache structure of *Neural Cache* limits easy accumulation between SRAM arrays. If the non-accumulating dimension of the weight matrix is mapped across the weight sets, each weight set would produce a partial product that either needs to be stored in the array or accumulated with a partial product outside its bitline or array. Mapping the accumulating dimension allows for the partial products for each weight set to be accumulated. Accumulating across other bitlines or arrays would only be needed after cycling through all the weight sets.

GNMT has multiple layers with 2048 and one layer with 3072 as the accumulating dimension. Despite having 12 storage slots, the GNMT mappings would occupy 8 and 12 storage slots for the 2048 and 3072 dimensions respectively. In the case of 2048, the layers do not need all 256 bitlines and 12 storage slots to map the dimension. Reducing the number of storage slots occupied is more efficient than reducing the number of bitlines as the bitlines all operate in parallel while the MACs from the storage slots are all sequential.

The accumulating dimension is satisfied across a PE's bitlines and the storage slots. The non-accumulating dimension is then mapped across the arrays in a bank, banks in a way, and ways in a slice. However, the GNMT layers are too large to fully map the weight matrix in the cache and require partitioning the network into two. Half the weights are loaded on-chip and after the matrix multiplication, the second half of the weights are loaded in. At least one way needs to be reserved for the CPU while another is needed for intermediates. However, using 18 ways versus 10 ways has only a marginal effect on performance as the layer still needs to be split in two. Thus, 10 ways

are used for compute, with 9 ways used to store intermediates, allowing for a larger batch size and thus better performance.

6.3 Results and Analysis

6.3.1 Eidetic comparison to *Neural Cache*

Eidetic significantly outperforms *Neural Cache* in several key metrics. *Eidetic* has a 19.5× improvement in throughput as seen in Table 6.1. The large difference in throughput can be explained by several key factors. *Eidetic* has $3.2\times$ as many SRAM PE arrays with 14336 compared to *Neural Cache* 4480 SRAM PE arrays. Due to mapping factors, *Eidetic* has 11264 SRAM PE arrays active for compute while *Neural Cache* only has 2080. This is roughly 5.4× more active PEs. Further impacting the performance is the utilization of PEs with compute mapped to them. Concurrent execution with efficient data movement in combination with a fused MAC operation allows the *Eidetic*'s PEs to have a utilization of almost 2.5× of *Neural Cache* with lockstep execution. Although *Neural Cache* frequency is $1.67\times$ higher than *Eidetic*, this factor is more than offset by *Neural Cache* taking 2 cycles for a compute operation due to the higher frequency and lower amount of write drivers in the SRAM PEs. Finally, *Eidetic* MACs finish roughly $1.2\times$ faster than *Neural Cache* because they are able to skip cycles.

Eidetic improvement in throughput does not come at a cost of energy, instead, it is 5.3× more energy efficient per sentence than *Neural Cache*. The main difference is how 55.6% of *Neural Cache*'s energy is spent in compute vs. 88.2% of *Eidetic*. *Eidetic* is more efficient in moving inputs and eliminates filter loading. *Eidetic* MAC speedup is limited by other PEs in the systolic column that does not skip as many cycles. *Eidetic* has an average wordline activation during the MACs of 72% while also having $1.2\times$ fewer cycles in the MAC. Additionally, within the compute, *Eidetic* is more energy efficient. *Eidetic* operates at a lower frequency and is thus more efficient per SRAM activation. Additionally, Eidetic is able to skip MAC cycles, resulting in fewer SRAM activations per MAC.

	NC Batch Size 1	NC Batch Size 32	Eidetic Batch Size 32
Latency Per Sentence (ms)	10.51	70.89	6.63
Throughput (Sentences/Sec)	95.2	451.3	8,833

Table 6.1: Comparison of Eidetic and *Neural Cache* Latency and Throughput

	NC Batch Size 1	NC Batch Size 32	Eidetic Batch Size 32
Average Watts (W)	27.4	35.6	115.0
Energy Per Sentence (J)	0.288	0.079	0.013

Table 6.2: Comparison of Eidetic and *Neural Cache* Energy and Power

Past in-SRAM computing for neural network acceleration has involved modifying a Xeon LLC to augment the SRAM arrays with computing peripherals [90]. This previous work, *Neural Cache*, involved limited modifications to maintain the LLC's original functionality and performance. *Eidetic* allows for significant customization to improve on performance of *Neural Cache*. *Eidetic* has the ability to pin weights for computation, has a custom interconnect and dataflow architecture, more efficient MACs, and a more efficient and finer mapping granularity.

6.3.2 Custom Interconnect and Dataflow Architecture

Neural Cache cache structure is optimized for efficiently fetching and storing data while balancing access latency, read/write throughput, energy per access, and area. However, with *Eidetic*, the SRAM arrays can be organized into systolic columns that simplify the data movement between the SRAM arrays. Compared to the layout of slices to ways to quads to banks to arrays, connecting the SRAM arrays to the arrays above and below in the systolic column is a simpler structure and is more efficient for the data movement required in neural networks.

The systolic arrays also allow *Eidetic* to map either dimension of the weight matrix to the storage slots. *Neural Cache* on the other hand only supports mapping the accumulating dimension. GNMT mapped on *Neural Cache* experiences reduced weight storage density in the SRAM arrays as several layers only used 8 of the storage slots. The reduced density played a large role in requiring the layers to be partitioned in two.

Neural Cache is limited without a well-defined programming model to handle its data move-

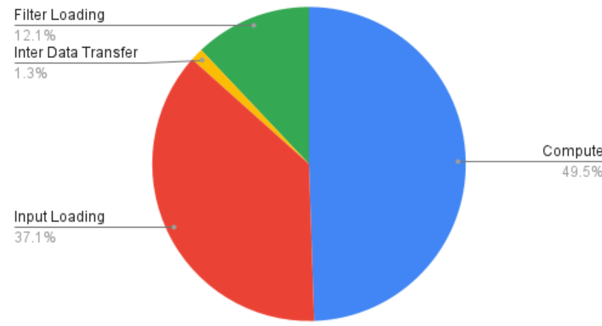

Figure 6.2: Breakdown of *Neural Cache*'s Throughput

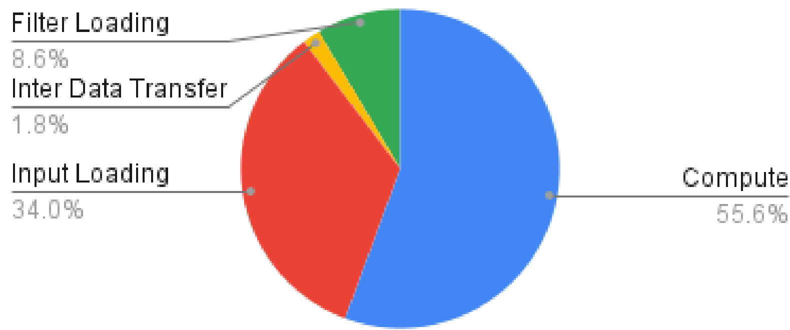

Figure 6.3: Breakdown of *Neural Cache*'s Energy

ment. *Eidetic* is able to take advantage of the directed graph nature of neural networks with its dataflow architecture and its custom control logic to facilitate both the required data movement and the scheduling of operations for concurrent parallel execution of multiple layers. The custom control logic allows multiple layers each having multiple inputs executing in the systolic columns, all without off-chip instructions. The output of layers will become the input to future layers, allowing the control logic to seamlessly issue the layers as all of its inputs become available. These architectural designs make it possible to fully take advantage of pinned weights and offer concurrent parallel execution with minimal off-chip instructions.

6.3.3 Increase Storage for Weight Pinning and In-Place Computation

Neural Cache and *Eidetic* are effectively area neutral with areas of 268.0 and 266.4 mm^2 respectively. However, despite being area neutral, *Eidetic* offers 128.25 MB of compute SRAM compared

to *Neural Cache* 35 MB, allowing *Eidetic* to store a much larger amount of weights on-chip. *Eidetic* offers more on-chip storage due to its simplified SRAM layout, higher density SRAM arrays, and lack of a separate compute area for the CPU.

By keeping weights on-chip, *Eidetic* is able to run multiple layers in parallel if correctly partitioned. Running multiple layers in parallel offers the significant advantage of delivering low latency results without sacrificing throughput. Systolic arrays need multiple inputs to remain saturated, a layer-by-layer approach would thus need batching and have a penalty when switching from layer to layer increasing the latency of inference.

Increasing on-chip weight storage significantly helps data-bound applications. Bottlenecks in data-bound applications can stem from both loading weights from off-chip and also from moving weights from on-chip storage to the compute units. Increasing the weights storage and pinning weights addresses both of these issues respectively. Finally, extending the length of the bitlines in the SRAM PEs makes *Eidetic* denser for storing weights at a slight cost of compute density.

6.3.4 Efficient MACs

Eidetic pulling the inputs out of the array and into the input latch allows for more efficient MACs in both energy and throughput. If the number of cycles differs across the PEs in a systolic column, the slowest PE dictates the throughput of the column. This is the case with the MAC algorithm that skips cycles based on both multiplier and multiplicand. But by pulling the input out and performing vector-scalar multiplication, *Eidetic* is able to reduce the number of cycles for the worst-case MAC and the average MAC. Reducing the worst-case MAC reduces energy consumption and increases throughput. Decreasing the average MAC further saves energy. *Neural Cache* MAC is approximately 400 cycles, while *Eidetic* average MAC is 238 cycles, and the average worst case MAC is 330 cycles.

6.3.5 Mapping Granularity

Eidetic benefits from a finer and more efficient mapping granularity. By storing the inputs outside the SRAM and extending the length of the bitlines, *Eidetic* can store a weight matrix of 16x256 in the PE. *Neural Cache* on the other hand, can store a 12x256 weight matrix in its SRAM arrays. Each of *Eidetic* 54 systolic columns can map a 16x[256x256] weight matrix. Additionally, the weight matrix can be as small as [256x256] without losing any compute resources. *Neural Cache* multi-tier cache hierarchy increases the likelihood that a network will have an inefficient mapping. Consider an example where the weight matrix mapping only occupies 3 of the arrays in a bank. The 4th array would be unused across all banks, eliminating 25% of the cache's compute power. Each level in the hierarchy introduces another opportunity for inefficient mapping. *Eidetic* SRAM structure allows for far greater control and more efficient mapping of weight matrices.

6.4 Summary

In *Neural Cache* we extend the capabilities of a Xeon processor to increase its parallel computing capabilities with a focus on CNNs. In-cache in-SRAM computing shows benefits for both data-bound and compute-bound neural network layers. However, more of the benefits come from increasing the amount of compute available rather than addressing available data. For instance, with a batch size of 1 for GNMT, *Neural Cache* spends 10.4% of the time doing compute with the remaining 89.6% spent moving data. With the batch size increased, however, this split becomes closer to 50% for each.

With *Eidetic*, 16 of the 330 cycles for a MAC are used to move data between PE units. Additionally, during execution, the PE units are active 94.6% of the time. Between these two factors, Eidetic can keep its PEs active at 90.0% with only 10.0% spent on data stalls.

The optimizations in *Eidetic* are particularly suited for data-bound applications, where *Neural Cache* falls short. The pinning of weights helps to solve off-chip data movement. The systolic arrays are able to target on-chip data movement, and the custom control logic and the interconnect

are optimized for data dependencies. Further, it has enough compute units that it is capable of producing GFLOPS/mm^2 in line with other industry ASICs.

However, *Eidetic* is an ASIC that is considerably expensive to design and build. Applications with low data reuse and complex data dependencies will be needed to justify its cost. This lines up with MLPs, RNNs, and Transformers.

CNNs are primarily compute-bound applications. *Eidetic* however focuses on how to remove off-chip data movement. While CNNs will benefit from a reduction of off-chip data movement Amdahl's law tells us that speedup gains will be limited. Furthermore, the execution model for *Eidetic* does not favor CNNs. RNNs and MLPs utilize vector-matrix multiplication while CNNs use convolutions that can be converted into matrix-matrix multiplication. The matrix-matrix multiplication results in unbalanced compute to storage requirements which create slowdowns in a concurrent pipelined approach. Instead, a layer-by-layer approach or a hybrid approach where multiple layers are pipelined together before moving to the next grouping of layers might be needed.

Instead, *Neural Cache* is a much cheaper option that can be applied to many different cache structures. *Neural Cache* can add significant amounts of parallel computing to a chip that otherwise had little to none. The problem is these compute units are not optimized for data movement or facilitating pinning of weights. *Neural Cache* is great for compute-bound applications such as CNNs.

CHAPTER 7

Conclusion

The field of computer architecture is experiencing an inflection point for the current computing paradigm. The slowing of Moore's law, the end of Denard's scaling, and the memory wall are forcing the community to innovate in order to continue realizing performance gains they were previously accustomed to.

Further complicating the issue is the immense increase in the amount of data available as well as the popularity of data-parallel applications such as DNNs. Increasing the emphasis on parallel computing with CPU vector processing, multi-core, GPUs, GPGPUs, and accelerators is a good start for handling data-parallel applications. The trade-off with focusing on parallel computing is that it comes at the cost of general computing and requires more specialized and separate units. Further, while ASICs can achieve significant performance increases, we can further improve the performance gains by eliminating the constraints of separate storage and compute.

Near-memory, in-memory, and emerging memory technologies can offer us new opportunities if we are willing to stray from traditional computing. In-memory computing provides a new computing paradigm that can help alleviate the costs that come with traditional computing. In-SRAM bit-serial computing also has the advantage of programmable precision. The computation is driven by an FSM which can be designed to support multiple precisions without having additional hardware for separate units. This adds significant flexibility over traditional ALU units.

First, in Chapter 4 we have demonstrated how an existing processor can be repurposed into a neural network accelerator by transforming the LLC into compute units capable of performing

arithmetic operations. With this, we provide the mapping strategy for using the SRAM PE to process neural networks. The re-purposed SRAM adds minimal area overhead but vastly improves the performance over the original CPU. Instead of devoting area and power to more vector units, more cores, or off-loading the work off-chip, we can simply modify a CPU's cache. This allows CPUs to gain the ability of parallel computing without sacrificing their general computing abilities.

Second, in Chapter 5 we have proposed a design for a custom ASIC that targets the challenges of data-bound DNNs by removing the need to go off-chip for both data and instructions and provides a data mapping that allows for pinned weights to be computed on in place. *Eidetic* goes further by handling on-chip data movement and dependencies, allowing for on-demand processing and concurrent layer execution. Finally, the PE units were customized to improve the compute efficiency. These improvements allow our design to compete with and outperform other state-of-the-art accelerators.

Industry leaders and startups have proposed in/near memory computing for memory-bound applications and/or neural network acceleration. These solutions and products span existing DRAM technologies to emerging such as Flash and PCM. Samsung has proposed three separate products that use processing near-memory for DRAM. These products; Aquabolt-XL HBM2-PIM, AXDIMM, and LPDDR5-PIM [105, 106, 107], target bandwidth-intensive workloads. Aquabolt-XL HBM2-PIM takes an eight-stack HBM and replaces four of the dies with a hybrid PIM DRAM die. Each PIM DRAM die contains 32 compute blocks. Each compute block dedicates half its area for DRAM and the other half to programmable compute units for a total of 128 compute cores in the system. Aquabolt-XL was integrated with commercially available GPU and FPGA to accelerate neural networks. Aquabolt-XL showed the post-performance gain from memory-bound applications with a batch size of one. AXDIMM offers acceleration buffers inside the DIMM. DIMM PIM offers a plug-in solution to existing commercial systems. The acceleration buffers contain programmable FPGA fabrics allowing support for a range of computations and applications. Samsung has also proposed PIM with mobile DRAM to target edge and low-energy applications. These three products are a significant investment in near/in-memory computing by

an industry leader. The startup mythic AI [65] has designed two generations of in-flash analog computing for neural network acceleration. IBM has created a prototype chip using computational phase-change memory [108] to accelerate neural network inference.

In Chapter 3 we detail the vast amount of academic work for in/near-memory computing with a particular focus on neural network acceleration. This, coupled with a large investment from industry, shows the importance of retooling our computing paradigm to support in/near-memory computing. The Von Neumann architecture is insufficient for today's computing and industry and academics know it. The work in this dissertation details both repurposing existing on-chip memory structures and a custom architecture with in-SRAM computing. Together, our work offers a complete solution that is missing from other works. In Chapter 3 we discuss emerging memory works such as ReRAM, STT-MRAM, Flash, and PCM. Additionally Ferroelectric RAM (FeRAM) [109] and Carbon Nanotube RAM (NANO RAM/ NRAM) [110] have also been proposed as alternative memories. Our work in *Neural Cache* and *Eidetic* utilized in-SRAM computing but is applicable to any on-chip in-memory computing technology.

Managing both computation and data movement/availability will be a huge challenge in designing efficient architectures. Large data-parallel applications with heavy compute requirements are here to stay. This dissertation offers scalable solutions for applications in high demand. By offering both the ability to extend support to a general purpose processor for parallel computing as well as a custom architecture, we show that in-SRAM computing has high potential to become a new computing paradigm.

BIBLIOGRAPHY

[1] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[2] S. Sarangi and B. Baas, "Deepscaletool: A tool for the accurate estimation of technology scaling in the deep-submicron era," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2021.

[3] S. Jeloka, N. Akesh, D. Sylvester, and D. Blaauw, "A configurable tcam / bcam / sram using 28nm push-rule 6t bit cell," IEEE Symposium on VLSI Circuits, pp. C272–C273, IEEE, 2015.

[4] M. Kang, E. P. Kim, M. s. Keel, and N. R. Shanbhag, "Energy-efficient and high throughput sparse distributed memory architecture," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 2505–2508, IEEE, 2015.

[5] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA-23)*, pp. 481–492, IEEE, 2017.

[6] A. Subramaniyan, J. Wang, E. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 259–272, ACM, 2017.

[7] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, ACM, 2017.

[8] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.

[9] "Deep dive into Amazon Inferentia: A custom-built chip to enhance ML and AI,"

[10] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, IEEE, 2018.

[11] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," *CoRR*, vol. abs/1803.04014, 2018.

[12] "Nvidia tensor cores: Versatility for hpc and; ai."

[13] L. Gwennap, "Tenstorrent scales ai performance," Apr 2020.

[14] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, *et al.*, "Think fast: a tensor streaming processor (tsp) for accelerating deep learning workloads," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 145–158, IEEE, 2020.

[15] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 821–834, 2019.

[16] A. Azizimazreah and L. Chen, "Shortcut mining: Exploiting cross-layer shortcut reuse in dcnn accelerators," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 94–105, 2019.

[17] C.-T. Huang, Y.-C. Ding, H.-C. Wang, C.-W. Weng, K.-P. Lin, L.-W. Wang, and L.-D. Chen, "ecnn: A block-based and highly-parallel cnn accelerator for edge inference," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 182–195, 2019.

[18] S. Gudaparthi, S. Narayanan, R. Balasubramonian, E. Giacomin, H. Kambalasubramanyam, and P.-E. Gaillardon, "Wire-aware architecture and dataflow for cnn accelerators," in *Proceedings of the 52nd annual ieee/acm international symposium on microarchitecture*, pp. 1–13, 2019.

[19] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 151–165, 2019.

[20] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 367–379, IEEE, 2016.

[21] G. Shomron and U. Weiser, "Non-blocking simultaneous multithreading: Embracing the resiliency of deep neural networks," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 256–269, IEEE, 2020.

[22] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 940–953, IEEE, 2020.

[23] Z. Song, B. Fu, F. Wu, Z. Jiang, L. Jiang, N. Jing, and X. Liang, "Drq: Dynamic region-based quantization for deep neural network acceleration," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1010–1021, IEEE, 2020.

[24] A. D. Lascorz, S. Sharify, I. Edo, D. M. Stuart, O. M. Awad, P. Judd, M. Mahmoud, M. Nikolic, K. Siu, Z. Poulos, *et al.*, "Shapeshifter: Enabling fine-grain data width adaptation in deep learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 28–41, 2019.

[25] S. Koppula, L. Orosa, A. G. Yağlıkçı, R. Azizi, T. Shahroodi, K. Kanellopoulos, and O. Mutlu, "Eden: Enabling energy-efficient, high-performance deep neural network inference using approximate dram," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 166–181, 2019.

[26] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 749–763, 2019.

[27] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[28] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE Computer Society, 2014.

[29] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775, IEEE, 2018.

[30] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The terasys massively parallel pim array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[31] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "Flexram: toward an advanced intelligent memory system," in *Computer Design, 1999. (ICCD '99) International Conference on*, 1999.

[32] P. M. Kogge, “Execube-a new architecture for scaleable mpps,” in *Parallel Processing, 1994. Vol. 1. ICPP 1994. International Conference on*, vol. 1, pp. 77–84, IEEE, 1994.

[33] M. Oskin, F. Chong, and T. Sherwood, “Active pages: a computation model for intelligent memory,” in *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, 1998.

[34] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, “A case for intelligent ram,” *Micro, IEEE*, vol. 17, no. 2, pp. 34–44, 1997.

[35] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, “Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory,” in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 27–39, IEEE Press, 2016.

[36] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 14–26, IEEE Press, 2016.

[37] Y. Ji, Y. Zhang, X. Xie, S. Li, P. Wang, X. Hu, Y. Zhang, and Y. Xie, “Fpsa: A full system stack solution for reconfigurable reram-based nn accelerator architecture,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 733–747, 2019.

[38] M. Imani, S. Gupta, Y. Kim, and T. Rosing, “Floatpim: In-memory acceleration of deep neural network training with high precision,” in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 802–815, IEEE, 2019.

[39] L. Song, X. Qian, H. Li, and Y. Chen, “Pipelayer: A pipelined reram-based accelerator for deep learning,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 541–552, 2017.

[40] H. Kim, J. Sim, Y. Choi, and L.-S. Kim, “Nand-net: Minimizing computational complexity of in-memory processing for binary neural networks,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 661–673, IEEE, 2019.

[41] H. Jia, H. Valavi, Y. Tang, J. Zhang, and N. Verma, “A programmable heterogeneous microprocessor based on bit-scalable in-memory computing,” *IEEE Journal of Solid-State Circuits*, vol. 55, no. 9, pp. 2609–2621, 2020.

[42] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag, “Promise: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 43–56, IEEE, 2018.

[43] J.-W. Su, X. Si, Y.-C. Chou, T.-W. Chang, W.-H. Huang, Y.-N. Tu, R. Liu, P.-J. Lu, T.-W. Liu, J.-H. Wang, *et al.*, “15.2 a 28nm 64kb inference-training two-way transpose multibit 6t

sram compute-in-memory macro for ai edge chips," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 240–242, IEEE, 2020.

[44] X. Si, Y.-N. Tu, W.-H. Huang, J.-W. Su, P.-J. Lu, J.-H. Wang, T.-W. Liu, S.-Y. Wu, R. Liu, Y.-C. Chou, *et al.*, "15.5 a 28nm 64kb 6t sram computing-in-memory macro with 8b mac operation for ai edge chips," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 246–248, IEEE, 2020.

[45] J.-W. Su, Y.-C. Chou, R. Liu, T.-W. Liu, P.-J. Lu, P.-C. Wu, Y.-L. Chung, L.-Y. Hung, J.-S. Ren, T. Pan, *et al.*, "16.3 a 28nm 384kb 6t-sram computation-in-memory macro with 8b precision for ai edge chips," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 250–252, IEEE, 2021.

[46] Q. Dong, M. E. Sinangil, B. Erbagci, D. Sun, W.-S. Khwa, H.-J. Liao, Y. Wang, and J. Chang, "15.3 a 351tops/w and 372.4 gops compute-in-memory sram macro in 7nm finfet cmos for machine-learning applications," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 242–244, IEEE, 2020.

[47] X. Si, J.-J. Chen, Y.-N. Tu, W.-H. Huang, J.-H. Wang, Y.-C. Chiu, W.-C. Wei, S.-Y. Wu, X. Sun, R. Liu, *et al.*, "24.5 a twin-8t sram computation-in-memory macro for multiple-bit cnn-based machine learning," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 396–398, IEEE, 2019.

[48] J.-H. Kim, J. Lee, J. Lee, H.-J. Yoo, and J.-Y. Kim, "Z-pim: An energy-efficient sparsity aware processing-in-memory architecture with fully-variable weight precision," in *2020 IEEE Symposium on VLSI Circuits*, pp. 1–2, IEEE, 2020.

[49] A. K. Ramanathan, G. S. Kalsi, S. Srinivasa, T. M. Chandran, K. R. Pillai, O. J. Omer, V. Narayanan, and S. Subramoney, "Look-up table based energy efficient processing in cache support for neural network acceleration," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 88–101, IEEE, 2020.

[50] A. Dhar, X. Wang, H. Franke, J. Xiong, J. Huang, W.-m. Hwu, N. S. Kim, and D. Chen, "Freac cache: folded-logic reconfigurable computing in the last level cache," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 102–117, IEEE, 2020.

[51] A. V. Nori, R. Bera, S. Balachandran, J. Rakshit, O. J. Omer, A. Abuhatzera, B. Kuttanna, and S. Subramoney, "Reduct: Keep it close, keep it cool! : Efficient scaling of dnn inference on multi-core cpus with near-cache compute," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 167–180, 2021.

[52] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 288–301, ACM, 2017.

[53] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations

using commodity dram technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 273–287, ACM, 2017.

[54] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. Kozuch, O. Mutlu, P. Gibbons, and T. Mowry, "Fast bulk bitwise and and or in dram," *Computer Architecture Letters*, 2015.

[55] Intel Corporation, *Desktop 4th Generation Intel® Core™ Processor Family, Desktop Intel® Pentium® Processor Family, and Desktop Intel Celeron® Processor Family*, 2015.

[56] Intel Corporation, *5th Generation Intel® Core™ Processor Family, Intel® Core™ M Processor Family, Mobile Intel® Pentium® Processor Family, and Mobile Intel® Celeron® Processor Family Datasheet*, 2015.

[57] Intel Corporation, *6th Generation Intel® Processor Datasheet for S-Platforms*, 2015.

[58] I. Skochinsky, "Secrets of intel management engine." "http://www.slideshare.net/codeblue_jp/igor-skochinsky-enpub". Accessed: 2016-02-17.

[59] S. Gueron, "A memory encryption engine suitable for general purpose processors.," *IACR Cryptology ePrint Archive*, vol. 2016, p. 204, 2016.

[60] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Proceedings of ISCA*, vol. 43, pp. 380–392, IEEE, 2016.

[61] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *SIGARCH Comput. Archit. News*, vol. 45, p. 751–764, apr 2017.

[62] M. Zhou, W. Xu, J. Kang, and T. Rosing, "Transpim: A memory-based acceleration via software-hardware co-design for transformer," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1071–1085, IEEE, 2022.

[63] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," *CoRR*, vol. abs/1908.03072, 2019.

[64] X. Sun, H. Wan, Q. Li, C.-L. Yang, T.-W. Kuo, and C. J. Xue, "Rm-ssd: In-storage computing for large-scale recommendation inference," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1056–1070, 2022.

[65] Mythic, "Taking powerful, efficient inference to the edge." https://mythic.ai/wp-content/uploads/2022/02/MythicWhitepaper-2019oct31.pdf. Accessed: 2022-8-15.

[66] W. Kang, H. Wang, Z. Wang, Y. Zhang, and W. Zhao, "In-memory processing paradigm for bitwise logic operations in stt–mram," *IEEE Transactions on Magnetics*, vol. 53, no. 11, pp. 1–4, 2017.

[67] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, “Computing in memory with spin-transfer torque magnetic ram,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 470–483, 2017.

[68] Y. Pan, P. Ouyang, Y. Zhao, W. Kang, S. Yin, Y. Zhang, W. Zhao, and S. Wei, “A multi-level cell stt-mram-based computing in-memory accelerator for binary convolutional neural network,” *IEEE Transactions on Magnetics*, vol. 54, no. 11, pp. 1–5, 2018.

[69] G. Burr, R. Shelby, C. di Nolfo, J. Jang, R. Shenoy, P. Narayanan, K. Virwani, E. Giacometti, B. Kurdi, and H. Hwang, “Experimental demonstration and tolerancing of a large-scale neural network (165,000 synapses), using phase-change memory as the synaptic weight element,” in *2014 IEEE International Electron Devices Meeting*, pp. 29.5.1–29.5.4, 2014.

[70] G. W. Burr, P. Narayanan, R. M. Shelby, S. Sidler, I. Boybat, C. di Nolfo, and Y. Leblebici, “Large-scale neural networks implemented with non-volatile memory as the synaptic weight element: Comparative performance analysis (accuracy, speed, and power),” in *2015 IEEE International Electron Devices Meeting (IEDM)*, pp. 4.4.1–4.4.4, 2015.

[71] S. R. Nandakumar, M. Le Gallo, I. Boybat, B. Rajendran, A. Sebastian, and E. Eleftheriou, “Mixed-precision architecture based on computational memory for training deep neural networks,” in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2018.

[72] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[73] “Hybrid memory cube specification 2.0,” 2014.

[74] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture,” in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 336–348, IEEE, 2015.

[75] M. Huang, M. Mehalel, R. Arvapalli, and S. He, “An energy efficient 32-nm 20-mb shared on-die L3 cache for intel® xeon® processor E5 family,” *J. Solid-State Circuits*, vol. 48, no. 8, pp. 1954–1962, 2013.

[76] W. Chen, S.-L. Chen, S. Chiu, R. Ganesan, V. Lukka, W. W. Mar, and S. Rusu, “A 22nm 2.5 mb slice on-die l3 cache for the next generation xeon® processor,” in *VLSI Technology (VLSIT), 2013 Symposium on*, pp. C132–C133, IEEE, 2013.

[77] K. E. Batcher, “Bit-serial parallel processing systems,” *IEEE Transactions on Computers*, vol. 31, no. 5, pp. 377–384, 1982.

[78] P. B. Denyer and D. Renshaw, *VLSI signal processing: a bit-serial approach*, vol. 1.

[79] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1737–1746, 2015.

[80] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[81] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[82] N. P. Jouppi, C. Young, N. Patil, D. Patterson, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, ACM, 2017.

[83] "Parabix Transform.." `http://parabix.costar.sfu.ca/wiki/ParabixTransform`. Accessed: 2017-11-20.

[84] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron, "Parabix: Boosting the efficiency of text processing on commodity processors," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12, IEEE, 2012.

[85] Intel Corporation, "Cache Allocation Technology." `https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology`. Accessed: 2017-11-20.

[86] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818–2826, 2016.

[87] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: memory power estimation and capping," in *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pp. 189–194, IEEE, 2010.

[88] Nvidia Corporation, "Nvidia system management interface." `https://developer.nvidia.com/nvidia-system-management-interface`. Accessed: 2017-11-18.

[89] Intel Corporation, "Intel vtune amplifier performance profiler." `https://software.intel.com/en-us/intel-vtune-amplifier-xe`. Accessed: 2017-11-18.

[90] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 383–396, IEEE, 2018.

[91] J. Yue, X. Feng, Y. He, Y. Huang, Y. Wang, Z. Yuan, M. Zhan, J. Liu, J.-W. Su, Y.-L. Chung, *et al.*, "A 2.75-to-75.9 tops/w computing-in-memory nn processor supporting set-associate block-wise zero skipping and ping-pong cim with simultaneous computation and weight updating," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 238–240, IEEE, 2021.

[92] R. Guo, Z. Yue, X. Si, T. Hu, H. Li, L. Tang, Y. Wang, L. Liu, M.-F. Chang, Q. Li, *et al.*, "15.4 a 5.99-to-691.1 tops/w tensor-train in-memory-computing processor using bit-level-sparsity-based optimization and variable-precision quantization," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 242–244, IEEE, 2021.

[93] J. Yue, Z. Yuan, X. Feng, Y. He, Z. Zhang, X. Si, R. Liu, M.-F. Chang, X. Li, H. Yang, *et al.*, "14.3 a 65nm computing-in-memory-based cnn processor with 2.9-to-35.8 tops/w system energy efficiency using dynamic-sparsity performance-scaling architecture and energy-efficient inter/intra-macro data reuse," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 234–236, IEEE, 2020.

[94] H. Jia, M. Ozatay, Y. Tang, H. Valavi, R. Pathak, J. Lee, and N. Verma, "15.1 a programmable neural-network inference accelerator based on scalable in-memory computing," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 236–238, IEEE, 2021.

[95] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 397–410, 2019.

[96] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference," 2020.

[97] Z.-G. Liu, P. N. Whatmough, Y. Zhu, and M. Mattina, "S2ta: Exploiting structured sparsity for energy-efficient mobile cnn acceleration," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 573–586, 2022.

[98] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018.

[99] Y. Xiang and H. Kim, "Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, pp. 392–405, 2019.

[100] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 446–459, IEEE, 2020.

[101] E. Karl, Y. Wang, Y. Ng, Z. Guo, F. Hamzaoglu, U. Bhattacharya, K. Zhang, K. Mistry, and M. Bohr, "A 4.6ghz 162mb sram design in 22nm tri-gate cmos technology with integrated active vmin-enhancing assist circuitry," in *2012 IEEE International Solid-State Circuits Conference*, pp. 230–232, 2012.

[102] A. Namin, K. Leboeuf, H. Wu, and M. Ahmadi, "Artificial neural networks activation function hdl coder," pp. 389 – 392, 07 2009.

[103] Y. Jiao, L. Han, R. Jin, Y.-J. Su, C. Ho, L. Yin, Y. Li, L. Chen, Z. Chen, L. Liu, Z. He, Y. Yan, J. He, J. Mao, X. Zai, X. Wu, Y. Zhou, M. Gu, G. Zhu, R. Zhong, W. Lee, P. Chen,

Y. Chen, W. Li, D. Xiao, Q. Yan, M. Zhuang, J. Chen, Y. Tian, Y. Lin, W. Wu, H. Li, and Z. Dou, "7.2 a 12nm programmable convolution-efficient neural-processing-unit chip achieving 825tops," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 136–140, 2020.

[104] https://asc.llnl.gov/sites/asc/files/2020-09/coral2-ml-dl-microbenchmark-summary-v3.pdf. Accessed: 2022-7-18.

[105] J. H. Kim, S.-h. Kang, S. Lee, H. Kim, W. Song, Y. Ro, S. Lee, D. Wang, H. Shin, B. Phuah, *et al.*, "Aquabolt-xl: Samsung hbm2-pim with in-memory processing for ml accelerators and beyond," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, pp. 1–26, IEEE, 2021.

[106] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, *et al.*, "Near-memory processing in action: Accelerating personalized recommendation with axdimm," *IEEE Micro*, vol. 42, no. 1, pp. 116–127, 2021.

[107] J. H. Kim, S.-h. Kang, S. Lee, H. Kim, Y. Ro, S. Lee, D. Wang, J. Choi, J. So, Y. Cho, *et al.*, "Aquabolt-xl hbm2-pim, lpddr5-pim with in-memory processing, and axdimm with acceleration buffer," *IEEE Micro*, 2022.

[108] V. Joshi, M. Le Gallo, S. Haefeli, I. Boybat, S. R. Nandakumar, C. Piveteau, M. Dazzi, B. Rajendran, A. Sebastian, and E. Eleftheriou, "Accurate deep neural network inference using computational phase-change memory," *Nature communications*, vol. 11, no. 1, pp. 1–13, 2020.

[109] T. Mikolajick, C. Dehm, W. Hartner, I. Kasko, M. Kastner, N. Nagel, M. Moert, and C. Mazure, "Feram technology for high density applications," *Microelectronics Reliability*, vol. 41, no. 7, pp. 947–950, 2001.

[110] W. Zhang, N. K. Jha, and L. Shang, "Low-power 3d nano/cmos hybrid dynamically reconfigurable architecture," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 6, no. 3, pp. 1–32, 2010.