ENERGY EFFICIENT HARDWARE FOR NEURAL NETWORK APPLICATIONS

by

Trishit Dutta

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Electrical Engineering

School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2023

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Kaushik Roy, Chair

School of Electrical and Computer Engineering

Dr. Anand Raghunathan

School of Electrical and Computer Engineering

Dr. Sumeet Gupta

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

Dedicated to my Baba, Ma and Mimo.

ACKNOWLEDGMENTS

I would like to express my wholehearted gratitude to my advisor Prof. Kaushik Roy for his steadfast support and guidance throughout my master's research journey. Their exceptional mentorship has allowed me to develop as an engineering professional and delve into various captivating topics, while also refining my analytical and critical thinking skills. I could not have asked for a more inspiring and dedicated mentor for my master's thesis.

I am also deeply grateful to my committee members Prof. Anand Raghunathan and Prof. Sumeet Gupta for their continuous encouragement and collaboration. Their invaluable insights and suggestions at each stage of my master's journey have kept me motivated and focused on achieving my goals.

I would like to extend my sincere thanks to my fellow researchers at NRL with whom I have had the pleasure of working and sharing memorable experiences. Special thanks go out to Deepika, Dong Eun, Sakshi, and Shubham for the engaging and thought-provoking discussions. Their collective knowledge and camaraderie have enriched my academic journey and contributed significantly to my personal and professional growth.

A special acknowledgment goes to Nicole for her willingness to assist with all administrative tasks. Their support has been crucial in ensuring a smooth and successful research experience.

Lastly, and most importantly, I would like to express my deepest gratitude to my parents, my brother and my friends for their unwavering love and support. Their belief in my abilities and constant encouragement have provided me with the strength to persevere and achieve my goals. This accomplishment would not have been possible without them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

With the explosion of AI in recent years, there has been an exponential rise in the demand for computing resources. Although Moore's law has so far kept up with conventional computational demands in the past, it has become evident that the efficiency and area gains with transistor scaling are no longer exponential, but rather incremental. The standard Von Neumann architecture imposes a limit on efficiency and latency as data is shuttled repeatedly between the compute and memory units. On the other hand, AI workloads rely heavily on matrix-vector-multiplication which get exponentially expensive with vector widths. In-memory and near-memory computing have come up as promising alternatives that addresses both these issues elegantly while reducing energy requirements.

A variety of NN models rely on fast and repetitive evaluation of exponential transcendental functions. In many cases, this is done by range reduction technique and math tables. For optimal energy efficiency and throughput, it is best if these tables reside as close as possible to the circuit where it is consumed. We propose a mixed-signal macro with dual functionality: ability to do matrix vector multiplication as well evaluate exp(x) for 32-bit IEEE 754 floating point number. The said macro consists of 64x64 array of special 8T cells that stores the math tables without hindering normal SRAM functionality. The charge based MVM engine uses two ADCs with reconfigurable precision, allowing faster throughput for sparse inputs. As the outputs of these operations are separate, it allows for high flexibility to use the macro in any neural-network hardware that needs either or both the functions.

Spiking Neural Networks (SNN) can perform sequential learning tasks efficiently by using the inherent recurrence of membrane potential (Vmem) accumulation over several timesteps. However, the data movement of Vmem creates additional memory accesses, which becomes a bottleneck operation. Additionally, SNN input spikes are highly sparse in nature, which can be exploited for efficient hardware implementation. We propose an SNN accelerator based on in-memory processing that addresses these. The said accelerator consists of 9 compute macros and 3 neuron macros, which can be programmed to work either serially (9 compute, 1 neuron) or in 3 parallel sets (3 compute, 1 neuron) to support different layer sizes. Peripheral logic computes the

membrane potential and stores it in the same compute macro, thus avoiding unnecessary data movement. The neuron macro keeps track of final membrane potential and generates output spikes. This accelerator was designed to run at 200Mhz at 1.2v in TSMC 65nm node.

1. INTRODUCTION

With the ever-increasing demand for more compute performance for less cost, Moore's law has kept up its promise for several decades. However, in recent years, we have seen that scaling the transistor further does not yield the same exponential improvements we have come to expect over the past decades. As compute circuits grow faster and faster, it has become increasingly difficult to keep it well-fed with data. This fact becomes especially significant as more bandwidth is given to memory, we hit a point of diminishing returns, where improvement in memory latency falls behind the growing energy and area requirements [1]. The standard Von-Neumann architecture thus presents itself as a limit to speed and efficiency. To add to the list of challenges, scaling in advanced technology nodes is not symmetric for memory and digital logic. As a result, the percentage of area dedicated to on-chip SRAM have steadily trended upwards [2]. With the explosion of Artificial Intelligence in everything, researchers are shifting their focus towards better ways to process Neural Network workloads. In sharp contrast to conventional sequential computing, NN workloads rely heavily on matrix-vector-multiplication which get exponentially expensive with layer size and depth. In-memory and near-memory computing have come up as promising alternatives that address both these issues elegantly while reducing energy requirements.

Prior research done in this domain has yielded varying levels of success in maximizing throughput and efficiency from conventional memory cell architectures [3]-[7]. Our focus is on using or re-using the SRAM array to do computation while minimizing data movement. Although the operation of SRAM in conventional context is digital in nature, it is worthwhile to note that analog or mixed-signal computation is also possible and generally achieves better throughput at the cost of accuracy. In a digital context, turning on multiple read-wordlines have the effect of NOR/AND operation on the read-bitlines. Similarly, in an analog setting, by turning on multiple read-wordlines, we achieve multiplication and accumulate effect in the read-bitline voltage. This gives us a great advantage in terms of parallel operations. To that effect we explore two implementations of compute-in-memory architecture.

Our first work involves embedding ROM data on pre-existing SRAM cells in an MVM engine. This enables us to evaluate exponential functions that form the basis of many transcendental functions. Evaluation of transcendental functions are frequently required in certain workloads such as LSTMs. A single macro that enables us to do both MVM operations as well as evaluating transcendental functions not only minimizes latency, but also maximizes energy efficiency. Our second work is a continuation of [17] and is focused on accelerating SNNs. We demonstrate how a modular approach to architecture provides great flexibility in implementation. We also back up our claims by presenting real-world silicon data of the said SNN accelerator.

2. CIM MACRO WITH DUAL FUNCTION - MVM ENGINE AND FAST EVALUATE EXPONENTIAL FUNCTION

2.1 Introduction

A variety of NN models rely on fast and repetitive evaluation of exponential transcendental functions. For example, calculation of hidden state in LSTMs [15] uses tanh function and calculation of membrane potential in LIF-SNN uses exponential function. Due to the nature of transcendental functions, it is often not practical to evaluate those function by the Maclaurin series expansion. In such cases, fast and accurate evaluation is done by means of range reduction technique and math tables [13]. For optimal energy efficiency and throughput, it is best if these tables reside as close as possible to the circuit where it is consumed, which is likely the neural network hardware itself. For that, these math tables must either exist as on-chip ROM or can be brought from main memory to on-chip cache. Fabricating these tables as on-chip ROM necessitates the use of additional silicon area, which is costly. On the other hand, swapping-in these tables from main memory incurs heavy latency and energy penalties, which defeats the purpose of fast evaluation.

The author of [19] describes a modification to the standard layout of the 8T cell such that it can store ROM data. This modification neither hinders the normal SRAM functionality, nor does it need require additional silicon area for the cell. The SRAM array remains virtually identical, except for addition of one extra vertical metal wire. Most of the peripheral control circuits, such as decoder, sense amplifiers, wordline and bitline drivers are re-used from the original circuit. This way, we end up extracting more functionality from the original hardware with minimal modifications to the design.

To that effect, we propose a mixed-signal macro with dual functionality: ability to do matrix vector multiplication as well evaluate exp(x) for 32-bit IEEE 754 floating point number. The said macro consists of 64x64 array of previously mentioned special 8T. The charge based MVM engine uses two ADCs with reconfigurable precision of 2/3/4/5/6-bits, allowing faster throughput for sparse inputs. As the outputs of these operations are separate, it allows high flexibility to use the macro in any neural-network hardware that needs either or both the functions. It is also worthy to

note that the silicon area consumed by the entire macro is no greater than the MVM engine [20] it is based on.

2.2 Embedding ROM data in SRAM

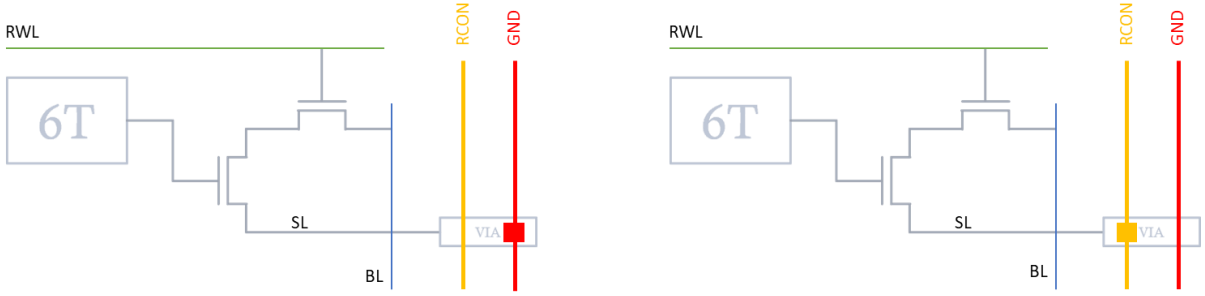
The modern SoC today packs several billions of transistors in a tiny area. Typically, the most transistor-dense parts of an SoC are SRAMs, which cover over half of the total silicon area. But scaling in advanced technology nodes is not symmetric for memory and digital logic, that is, digital logic scales more than memory devices. As a result, the percentage of area dedicated to on-chip SRAM have steadily trended upwards [2]. This motivates us to extract as much functionality as possible out of SRAM arrays. ROM arrays have traditionally been unpopular with designers as large ROM tables take up significant area. They also require peripheral circuitry that in turn take up considerable area as well. Typically, ROM data is not stored on expensive on-chip RAM but is stored in slow and inexpensive off-chip devices. It makes sense to bring the whole or parts of the ROM data to on-chip RAM as and when needed. However, in certain cases, this may not be feasible. Eg, Evaluation of transcendental functions in certain math libraries use large math tables. These tables may be too large to bring to on-chip cache. Even if parts of the table are brought to cache, the next evaluation may need to access some other line in the table, which may not be present in the cache. Another pathological scenario may arise when the cache replacement algorithm repeatedly pushes the math table out of the cache. Thus, each time the function needs to be evaluated, the processor is forced to wait hundreds of cycles for the table to be available, which defeats the purpose of using tables in the first place.

The authors in [19] explore ways in which ROM data may be stored in conventional SRAM arrays without hindering SRAM functionality or increasing area taken up by the array itself. In most cases, the parts of the peripheral circuit, such as decoder, sense amplifiers, wordline and bitline drivers may be shared between the SRAM and ROM modes of operation, thus extracting more functionality. These ROM-embedded SRAM (R-SRAM) can now store math table entries, which enables us to avoid fetching data from off-chip devices and saves us the associated latency and energy penalty. Let us discuss in brief the circuit level aspects of R-SRAM in the context of a conventional 8T cell.

(a) (b)

Figure 2.1 (a) Schematic of a typical 8-T SRAM cell (b) 8-T SRAM cell with SL connected to VDD

The schematic of a typical 8T cell is shown in figure 2.1(a). Note that the SL terminal is connected to GND. When this cell stores a 1 (Q=1) and RWL is asserted, both the read transistors turn on and drain the pre-charged RBL. Let us consider a variation of this cell in figure 2.1(b), where the SL is connected to VDD instead of GND. Now, when this cell stores a 1 (Q=1) and RWL is asserted, both the read transistors turn on and but there is no current flow. In this case, the RBL retains its pre-charged state.



(a) (b)

Figure 2.2: (a) Cell storing ROM data ‘0’ (b) Cell storing ROM data ‘1’

After observing the behavior of cells in figure 2.1, let us create a new terminal RCON. For all the cells we want to store ROM value of 1, we connect SL to RCON and for other cells where we want to store a ROM value of 0, we connect SL to GND. Note that when RCON is connected to GND, the SL of all cells are connected to GND and the entire array behave like a conventional

SRAM array. However, when RCON is connected to VDD, and all cells in that row store 1, certain RBLs are prevented from being discharged. This is the ROM mode of operation.

Also note that we must first write 1's to all cells in the row of which we want to fetch the ROM data. As this destroys the data previously present in that row, we must first copy the SRAM data to a temporary buffer and restore it after the ROM fetch is done. The whole operation is done in 4 cycles.

1. Cycle 1:
   a. Set RCON=0.
   b. Assert RWL to bring SRAM data to RBLs.
   c. Copy SRAM data to buffer.
2. Cycle 2:
   a. Set all WBL=1 and WBLB=0.
   b. Assert WWL to write 1's to all cells in row.
3. Cycle 3:
   a. Set RCON=1.
   b. Assert RWL to bring ROM data to RBLs.
   c. Read ROM data from RBLs.
4. Cycle 4:
   a. Assert WWL to write data from buffer to WBL/WBLB.

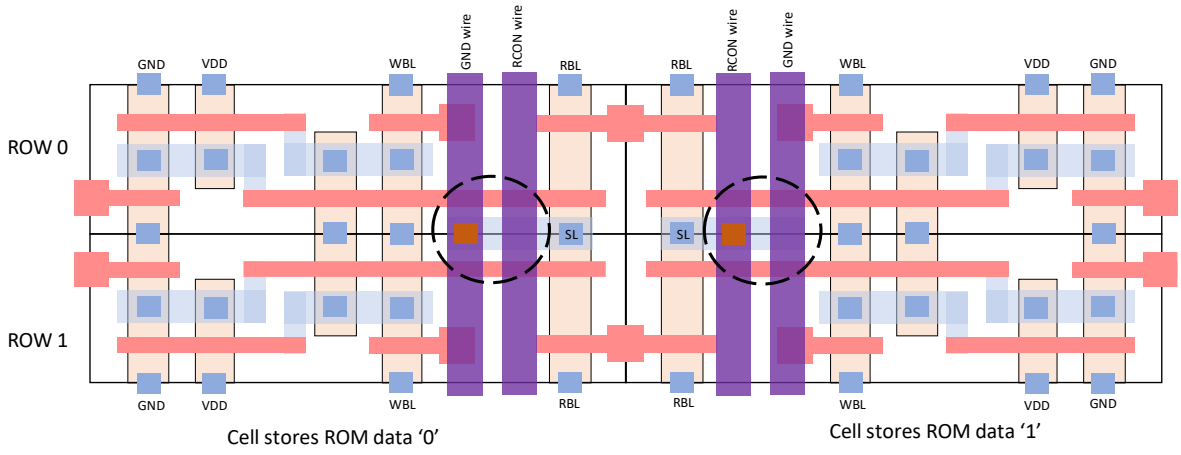


Figure 2.3: Layout of 2x2 array of 8-T cells storing '0' and '1'. Encircled section demonstrates how vias connect SL to RCON or GND

Let us discuss the layout aspects of this special 8T array. In a conventional 8T cell, the read and the write ports are isolated, which aids in tolerance to process variations and noise. The thin cell layout of an 8T cell is shown in figure. Due to the limited cell height, we observe that there is no space to accommodate a horizontal RCON wire. However, since 8T cells are wide, there is adequate space to accommodate a vertical RCON wire without expanding the dimensions of the cell. After we draw this wire, we selectively connect the SL terminal of each 8T cell to either RCON or to GND. Note that cells in two adjacent rows share the same connection of SL due to layout constraints. Hence, we can store 1 ROM bit per 2 SRAM cells. Our final ROM capacity is half of the SRAM array size.

2.3 Table based approximate evaluation of exponential function

Several neural networks such as RNNs and SNNs make use of transcendental functions as a decay factor for past inputs. The basic function in most of the applications involve the computation on exp(x). One way to evaluate exp(x) is by using the Maclaurin series expansion. However, when x is not a small number, it is easy to see that we need to evaluate several terms from the series to maintain reasonable accuracy. Additionally, as x grows larger, calculating each term in the series also gets progressively more expensive. Thus, due to the nature of transcendental functions, it is not feasible to use Maclaurin series in practice. Look-up-table based methods using range reduction prove more practical in this regard. Intel's math library uses three key steps to evaluate exp(x): reduction, approximation, and reconstruction. Let us illustrate this with an example:

1. Range reduction: We can express x as,

$$x = \frac{N \cdot \ln(2)}{2^K} + r$$

For some integer parameter K, which we will discuss later. We choose N such that $N = \left\lfloor \frac{x \cdot 2^K}{\ln(2)} \right\rfloor$. This guarantees us that x never exceeds $\ln(2)/2^K$. We can rewrite the expression as,

$$e^x = 2^{N/2^K} \cdot e^r$$

2. Approximation: Now that we have a reduced range for r, it is straightforward to calculate e^r by use of Chebychev's approximation or Remez's algorithm.
3. Reconstruction: After we have e^r, we need to multiply it by $2^{N/2^K}$ to reconstruct our final result. If we take an integer M to be $\lfloor N/2^K \rfloor$, we can rewrite the equation as,

$$e^x = 2^{N/2^K} \cdot e^r = 2^M \cdot 2^{d/2^K} \cdot e^r$$

Where $d = 0, 1, 2, \dots, 2^{K-1}$. All combinations of $2^{d/2^K}$ can be pre-computed and stored as look-up table entries. The number of entries needed is 2^K. Calculation of 2^M can be done as bitshift operation in binary systems.

By examining the above method, we make the following observations. Exploiting these will help in a more efficient circuit implementation.

1. For reconstruction, three terms need to be multiplied to get the result, which again needs to be converted to 32-bit FP representation. Multipliers are expensive, both in terms of area and power.
2. As K grows, the range of r reduces. Subsequently, the range of e^r also reduces and gets closer and closer to 1.
3. As K grows, the number of entries required for $2^{d/2^K}$ grow exponentially.

Most neural networks are tolerant to minor variation in its parameters. The observations made above together pose an interesting opportunity to trade accuracy with speed. If we choose an optimal K such that the range of e^r is sufficiently small, we can replace e^r by a constant (say 1), trading accuracy for reduced circuit complexity. Since the reconstructed result is a function of e^r, we choose $e^{r_{opt}}$ such that our maximum error is minimized. For this, $e^{r_{opt}}$ needs to be the average over the range of e^r. Thus,

$$e^{r_{opt}} = \frac{e^{r_{min}} + e^{r_{max}}}{2} = \frac{1 + e^{r_{max}}}{2}$$

This simplifies our implementation greatly due to the following reasons.

1. Replacing e^r by a constant (say $e^{r_{opt}}$) means we no longer need to dedicate hardware for calculating e^r by using polynomial approximations.
2. For reconstruction, the two remaining terms are: 2^M and $2^{d/2^K}$. Each term only affects the exponent and mantissa in the 32-bit FP representation respectively. Thus, both exponent and mantissa computations can be done independently without the need for a multiplier.
3. Instead of tabulating entries for $2^{d/2^K}$, we now tabulate entries for $2^{d/2^K} \cdot e^{r_{opt}}$. We can go one step further directly tabulate the equivalent mantissa in the table. This incurs no additional circuit complexity.

Now that we have explained the algorithm and modifications to it, let us compute the parameters we need to populate the e-ROM entries. Our macro uses a 64x64 array of 8T cells. We have previously discussed that two adjacent rows in 8T array must store the same ROM bit for layout related concerns. Hence, we effectively have 32 rows of 64 bits each for ROM entries. The mantissa in a 32-bit FP number is 23-bits long. Let us consider three options:

1. We can fit at most two whole mantissa entries in one 64-bit row. The maximum quantization error due to binary conversion in 23-bit mantissa is $Q_{23-bit} = 2^{-23}$. This gives us a total of $2 \times 32 = 64$ entries for our look-up table. The appropriate K parameter for this case is 6. The max range of r for $K = 6$ is given by,

$$(r_{max})_{K=6} = \ln(2)/2^6 \simeq 0.0108304 \ldots$$

Subsequently, the max value e^r can take is given by,

$$(e^{r_{max}})_{K=6} = e^{0.005415\ldots} \simeq 1.0108892 \ldots$$

We also calculate $e^{r_{opt}}$ as,

$$(e^{r_{opt}})_{K=6} = \frac{1 + (e^{r_{max}})_{K=6}}{2} \simeq 1.0054446 \ldots$$

For our approximation, the maximum error is given by,

$$error\ (\%) = 100 \times \left\{\frac{e^x - e^x_{approx}}{e^x}\right\}\%$$

$$= 100 \times \left\{\frac{\left(2^M \cdot 2^{d_{max}/2^K} \cdot e^{r_{max}}\right) - \left(2^M \cdot \left(2^{d_{max}/2^K} - Q_{23-bit}\right) \cdot \left(e^{r_{opt}}\right)_{K=6}\right)}{\left(2^M \cdot 2^{d_{max}/2^K} \cdot e^{r_{max}}\right)}\right\}\%$$

$$= 100 \times \left\{1 - \frac{\left(2^{d_{max}/2^K} - Q_{23-bit}\right) \cdot \left(e^{r_{opt}}\right)_{K=6}}{2^{d_{max}/2^K} \cdot e^{r_{max}}}\right\}\%$$

$$\approx 100 \times \left\{1 - \frac{\left(2^{63/64} - 1.19 \times 10^{-7}\right) \cdot 1.0054446}{2^{63/64} \times 1.0108892}\right\}\%$$

$$\approx 0.5386\%$$

Although this is still a fair compromise, we see that 23-bit precision in mantissa is an overkill if error due to approximating e^r term is large. We are also losing $64 - 46 = 18$ bits in each row as overhead.

2. Since 23-bit for mantissa precision is not beneficial, let us try 16-bit precision in mantissa. We can immediately see that we can fit 4 mantissa entries in one row without any overhead bits. This arrangement gives a us 128 entries for the look-up table and a maximum quantization error of $Q_{16-bit} = 2^{-16}$ due to binary conversion in mantissa. The appropriate value of parameter K in this case is 7. Going through the same error calculation as above,

$$error(\%) = 100 \times \left\{1 - \frac{\left(2^{d_{max}/2^K} - Q_{16-bit}\right) \cdot \left(e^{r_{opt}}\right)_{K=7}}{2^{d_{max}/2^K} \cdot e^{r_{max}}}\right\}\%$$

$$\approx 100 \times \left\{1 - \frac{\left(2^{127/128} - 1.52 \times 10^{-5}\right) \cdot 1.0027149}{2^{127/128} \times 1.0054299}\right\}\%$$

$$\approx 0.2708\%$$

3. The next higher K parameter is 8. For this case, the number of entries needed for the look-up table is 256, which allows only 8 bits for each entry of the mantissa. Here, mantissa incurs a maximum quantization error of $Q_{8-bit} = 2^{-8}$ due to binary conversion. The maximum error in our approximation is given by,

$$error(\%) = 100 \times \left\{1 - \frac{\left(2^{d_{max}/2^K} - Q_{8-bit}\right) \cdot \left(e^{r_{opt}}\right)_{K=8}}{2^{d_{max}/2^K} \cdot e^{r_{max}}}\right\}\%$$

$$\approx 100 \times \left\{1 - \frac{\left(2^{255/256} - 3.90 \times 10^{-3}\right) \cdot 1.0013556}{2^{255/256} \times 1.0027113}\right\}\%$$

$$\approx 0.3305\%$$

We proceed with $K = 7$ since it gives us the best combination of accuracy and precision.

2.4 Mixed-signal Matrix Vector Multiplication Engine (MVME)

The MVM engine in this work is taken from [20]. The multiplication is done in bit-serial fashion as shown in figure 2.4. External circuit is required for shift-and-add function. This is done to give flexibility for input word-width.

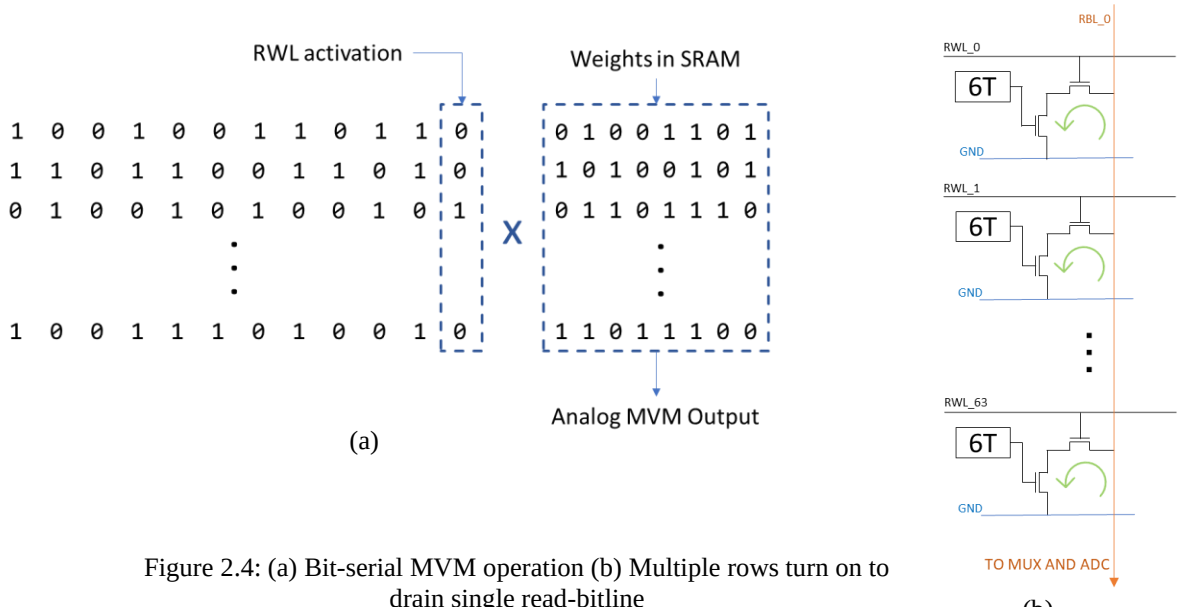


Figure 2.4: (a) Bit-serial MVM operation (b) Multiple rows turn on to drain single read-bitline

The engine comprises of a 64x64 array of 8T cells, 4 analog muxes and 2 ADCs. The multiplication is done in analog domain by turning on multiple read word lines simultaneously. The amount of discharge produced by each SRAM cell (storing a 1) is proportional to the RWL pulse height. In order to maintain sufficient dynamic range in the bitline voltage for higher ADC precision modes, the RWL pulse height voltage (VRWL) needs to be adjusted for each precision. Hence this voltage is provided external to the macro.

The bitlines are connected to 4 analog muxes. Depending on the selected precision, the analog muxes route the bitline voltages to the inputs of the ADC. Each ADC has reconfigurable precision of 2/3/4/5/6-bits. When operating in 2/3/5-bit precision, each ADC can sample and convert two inputs simultaneously, thus doubling throughput. In this case, the 4 muxes select one bitline each and connect it to the ADCs. It takes 64/4=16 ADC cycles to complete one MVM operation for these precision modes. When operating in 4/6-bit precision, two muxes are turned off and the other two select one bitline each and connect it to the ADCs. It takes 64/2=32 ADC cycles to complete one MVM operation in these precision modes.

2.5 Macro with dual function

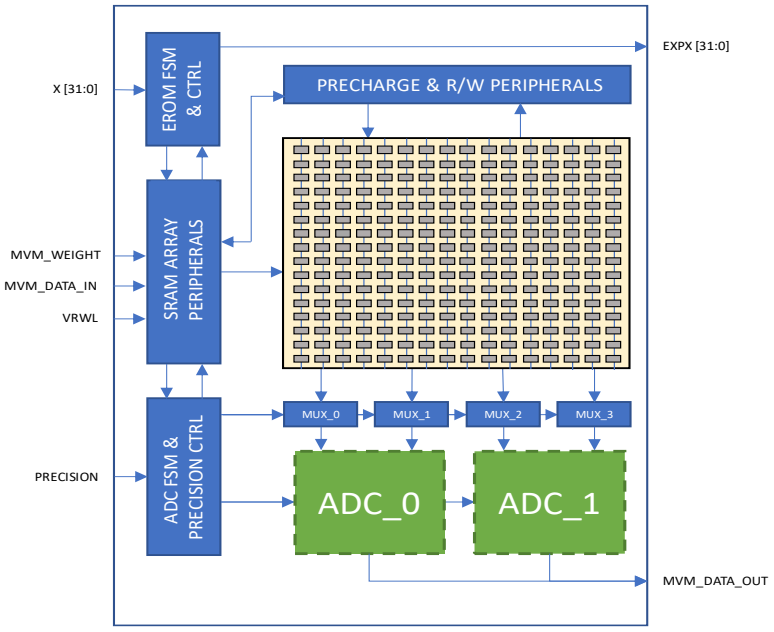
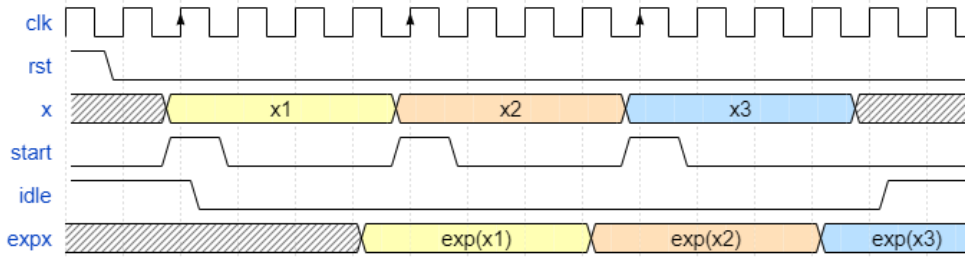


Figure 2.5: Block diagram of macro

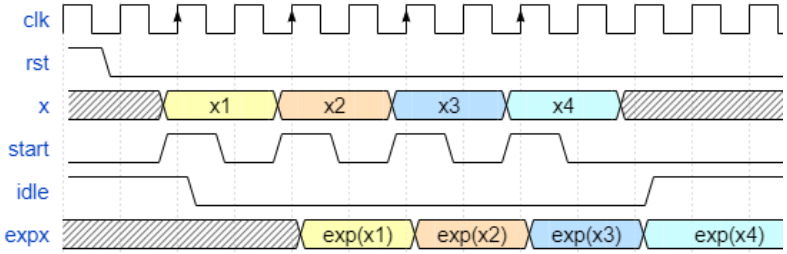
The proposed macro is designed in TSMC 65nm CMOS technology. It is able to do either exp(x) calculation or MVM operation at any one time. Clock gating is employed throughout the entire macro to ensure idle parts of the macro do not contribute to dynamic power. When operating in either mode, the macro raises a flag signaling busy state. To interrupt the any ongoing operation, the reset signal can be asserted at any time, which will immediately kill the operation and restore the macro to idle state. The footprint or the physical area taken by the entire macro is similar to the original MVM engine in [20].

The maximum frequency for MVM operation is 100Mhz. The throughput of the MVM engine is dependent on the precision of the ADC. For 2/3-bit operations, the access latency is 3 clock cycles and for 4/5/6-bit operations, it is 4 clock cycles. Since each ADC can sample two columns for 2/3-bit operations, the maximum throughput is achieved when the input sparsity is less than 12.5% (at most 8 RWLs on at a time) and 3-bit precision is used.

The exp(x) engine is designed for a maximum clock frequency of 250Mhz at 1.2v in TSMC 65nm technology. The supply voltage as well as the clock frequency can be lowered to reduce power consumption. The argument for exponential calculation (x) as well as the result are in 32-bit IEEE 754 format. There are two modes to run the exp(x) engine: normal and fast. For normal mode, the calculated result is available after 4 clock cycles from the output bus of the macro. The contents of the SRAM array are not disturbed in this mode. However, in situations where the contents of the SRAM are no longer needed, the engine can write to the array without needing to first save a back-up. In this ‘fast mode’, the calculated result is available after only 2 clock cycles. This mode is particularly useful if we are using the macro solely for exp(x) calculation. Timing diagram for both modes are shown in figure 2.6. Figure 2.7 plots the simulation results for energy vs throughput for exp(x) computation in the macro.



(a)



(b)

Figure 2.6 : (a) Timing diagram for normal mode of operation (b) Timing diagram for fast mode of operation

Table 2.1 : Latency of exp(x) operation at different voltages

Mode	Voltage (Volts)	Latency (ns)
Normal	0.9	22
	1.0	20
	1.1	18
	1.2	16
Fast	0.9	11
	1.0	10
	1.1	9
	1.2	8

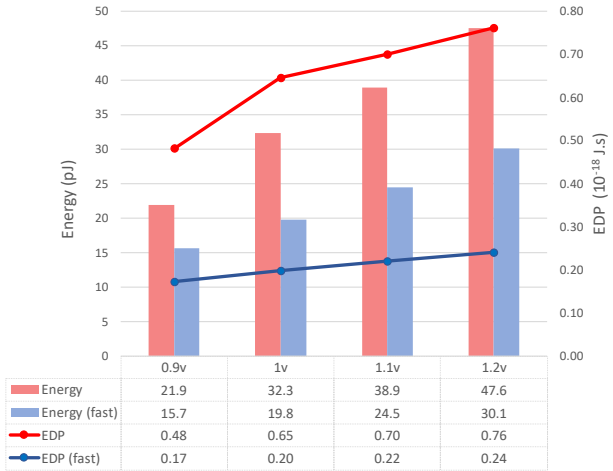


Figure 2.7: Energy and E-D product at different voltages

Table 2.2: Macro summary

Technology	TSMC 65nm CMOS
Macro Area	0.04 mm^2
Supply Voltage	0.9-1.2v (expx) / 1.2v (MVM)
Max. Frequency	250Mhz (expx) / 100Mhz (MVM)
SRAM size and type	64x64 8T cells
ADC precision	2/3/4/5/6-bit
MVM throughput (GOPS)	3.2 – 8.53 @ 100Mhz

2.6 Simulating an LSTM workload

One popular NN workload that requires frequent evaluation of transcendental function is LSTM networks. To overcome the vanishing gradient problem in LSTM, we need a function with a property such that its second derivative can sustain for a long range before going to zero. Tanh is one such function that is commonly used in this case.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

The evaluation of tanh(x) requires the evaluation of e^{2x} followed by a few algebraic computations. Let us assume that the hardware accelerator implementing this LSTM has dedicated logic to do this algebraic computation in one cycle. Thus, using the exp(x) engine, we get the result for tan(x) in only five cycles (three cycles if we use fast-mode). With this assumption, let us use the PUMA simulator [9] to run an LSTM workload in two cases. In the first case, we use our dual-purpose macro for evaluating exp(x) as well as for MVM operations. In the second case, we fetch the ROM table from DRAM and use an ALU to do the exp(x) evaluation. Note that in the second case, there is an energy and latency penalty due to accessing off-chip memory.

Using PUMA simulator [9], we can estimate the benefits of using the exp(x) engine in the dual function macro. Our implementation architecture assumes 1 tile containing 1 core. The core is further divided into 8 MVMUs and a common register file of 1kB. Each MVMU consists of 8 dual-function macros. The size of the inputs and hidden state of the LSTM workload in consideration is 64 and 64 respectively and the bit-width of each element is 8-bits.

Table 2.3: Comparing LSTM workload.

	Clock cycles	Energy (uJ)
Math table in DRAM	4138	4.72
Dual-function macro	3320	3.99
Improvement %	**19.8%**	**15.6%**

2.7 Conclusion

It is evident from the above simulation experiments that replacing an MVM engine with our proposed dual-purpose macro improves overall performance and efficiency. This comes at little to no additional cost to silicon area or power. In most neural network applications, where the accuracy of transcendental functions is not of paramount importance, this macro can be a drop-in replacement for bitwise-serial matrix vector multiplication engines, while providing the capability to fast evaluate those functions.

3. COMPUTE-IN-MEMORY BASED RECONFIGUREABLE SPIKING NEURAL NETWORK ACCELERATOR

3.1 Introduction

Spiking Neural Networks (SNNs) have gained increasing attention in recent years due to their ability to model the behavior of biological neurons in a more realistic way compared to traditional Artificial Neural Networks (ANNs). Unlike ANNs, SNNs process information using event-driven spikes which is more akin to how the biological neuron works. Where conventional NNs use multiply-and-accumulate operation, SNN only needs addition, which is a great deal simpler and efficient, both in terms of algorithms and hardware. Subsequently, the event-driven approach to inputs presents a unique opportunity to extract even more efficiency when there is adequate sparsity in incoming spikes [14]. The membrane potential in an SNN is crucial for the defining the present state of the neuron, computation of output spikes and the transmission of information between neurons. This makes SNNs more suitable for sequential learning tasks as compared to recurrent neural networks such as LSTMs. In LSTMs, memory effect is achieved by the hidden state, which needs to be fed back as input to the network [15]. In SNNs, the inherent membrane potential acts as the memory element, keeping track of all previous inputs, thereby avoiding the need for feedback paths.

Due to the nature of SNNs, the network needs to access and update memory elements storing the membrane potential (Vmem) and weights (Wmem) at each timestep. This poses a performance bottleneck as conventional memory architectures are orders of magnitude slower than compute units [16]. The repeated shuffling of data from memory to compute and then back to memory incurs significant energy consumption as well. Authors of [17] discusses custom compute-in-memory macro that addresses many of these concerns. The macro supports a maximum input fan-in of 128 and a total of 12 output channels. In this work, we explore modifications to the design that enable certain features such as chaining several macros together to handle a larger fan-in. We build reconfigurability in bit-precision of weights as well as show how we can distribute larger fan-in between several macros for parallel computation. In addition, we introduce zero-skipping FSM, which exploits the inherent sparsity in input spikes to reduce latency and improve energy efficiency.

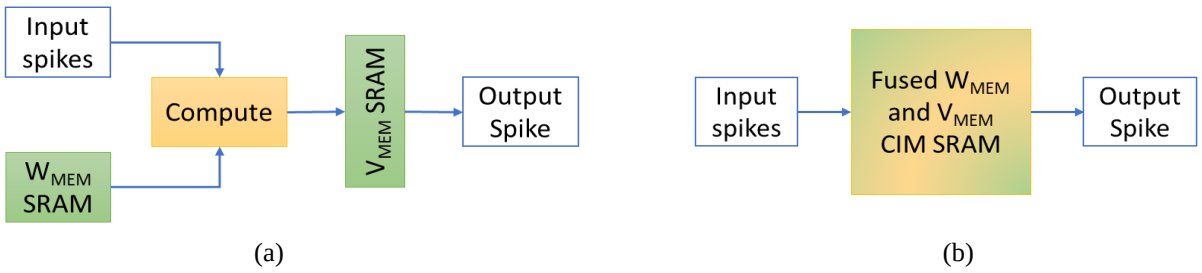


Figure 3.1: (a) Conventional approach for SNN implementation incurs data transfer bottleneck. (b) Proposed approach using fused compute and memory eliminates repeated data movements.

3.2 Architecture and Organization

The accelerator consists primarily of two core logical blocks: compute and neuron. We shall refer to them as compute_macro and neuron_macro. The compute_macro is the heart of the accelerator and is responsible for all SNN instruction processing. However, in a multi-macro implementation, neuron_macro is employed to accumulate the partial outputs from several compute_macro and generate the final output.

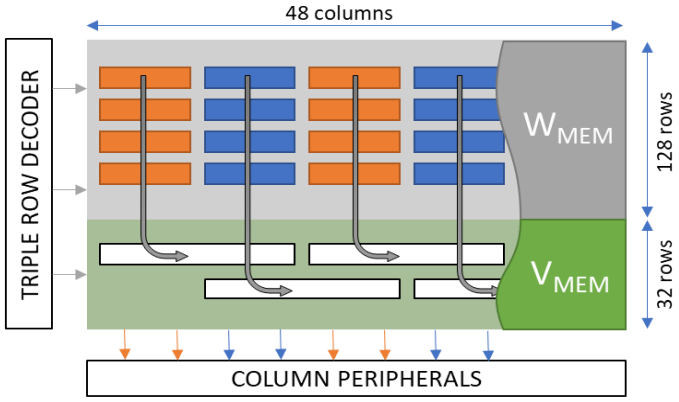


Figure 3.2: Organization of weights and membrane potentials in the SRAM in compute_macro along with other components

3.2.1 Compute macro

The compute_macro comprises an SRAM array, a triple-row decoder, and column peripheral circuits, which work together to process custom SNN instructions discussed later. The triple-row decoder has the capability to turn on two read wordlines (RWLs) and one write wordline (WWL) independently. When two RWLs are turned on simultaneously, the RBL gives the bitwise NOR/OR while the RBLB gives the bitwise NAND/AND of the data from the enabled rows. The SRAM array consists of two subarrays of 10T cells: the W_{MEM} subarray and the V_{MEM} subarray. The W_{MEM} subarray has 128 rows, each corresponding to an input neuron. Each row is 48-bits and depending on weight precision (discussed later) can store between 6-12 weights. The V_{MEM} subarray has 32 rows to store partial Vmems, each of which can store between 3-6 partial Vmems. The partial Vmems are distributed between two rows in a staggered odd/even fashion as shown in fig to correctly utilize the column peripheral circuits. The two subarrays are connected by a row of switches in the read bitlines (RBLs), which allow for reconfigurable weight precision as discussed below.

3.2.2 Reconfigurable Weight Precision

Each row in the V_{MEM} or W_{MEM} subarray is 48-bit wide. With 8-bit weight and 15-bit Vmem, we only get 6 output channels. To gain flexibility in the number of output channels, we can lower the weight precision. With 4-bit or 6-bit weights, we can get 12 or 8 output channels respectively. Reconfigurability in weight precision is primarily achieved by inserting switches in the read-bitlines between the V_{MEM} and the W_{MEM} subarray in the compute_macro. The RBL switches can be reprogrammed to support the selected precision. For example, in the case of 6-bit weight and 11-bit Vmem in odd cycle, the RBL switches connect the RBLs in the first 6 columns and disconnect the RBLs in next 6 columns and so on. The polarity of the switches is flipped in the even cycle. For 4-bit or 8-bit weights, sets of 4 or 8 switches connect/disconnect the columns respectively. The muxes in the column peripheral circuit are also reconfigured appropriately to support the selected weight precision.

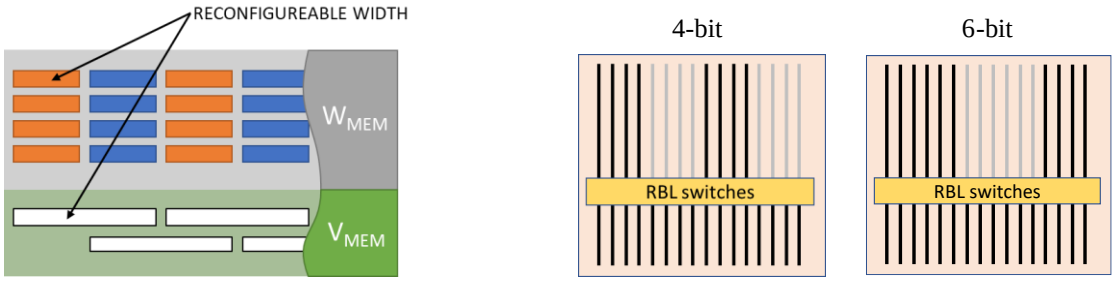
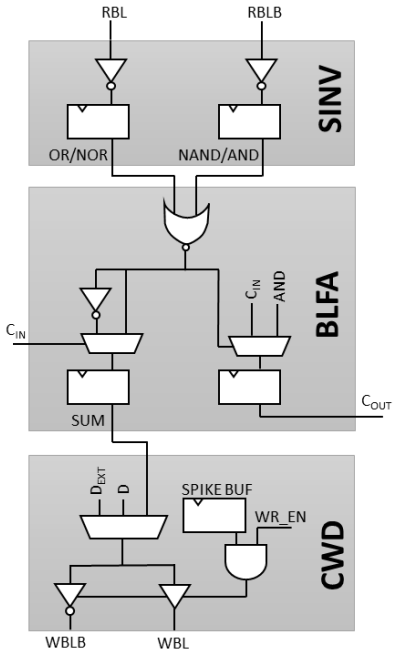


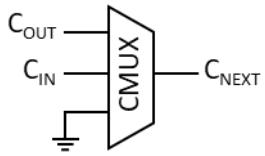
Figure 3.3: Supporting reconfigurable precision by decoupling RBLs in W_{MEM} and V_{MEM} subarray by incorporating switches.

3.2.4 Column Peripheral Circuit

The compute_macro includes column peripheral circuits that connect to sets of bitlines (RBL, RBLB, WBL, and WBLB). The sensing inverters (SINV) sense and latch the data on RBL/RBLB, which is then used by the bitwise logic full adder (BLFA) to generate SUM and C_{OUT} signals. SUM is written into the enabled WWL destination row using the conditional write-driver (CWD), while C_{OUT} is forwarded to the next column for accumulate operation, forming a ripple-carry adder. The modular design of the adder allows for reconfigurability during odd/even cycles using BLFAs from each column peripheral. To account for the staggered data mapping, each column peripheral can be reconfigured in carry forward (CF), carry skip (CS), LSB, and MSB modes using Carry-MUXes (CMUX). For example, in the case of 6-bit weight precision, during odd cycle, columns 0-11 form one adder, columns 12-23 form the next adder, and so on. During even cycle, columns 6-17 form one adder, columns 18-29 form the next adder, and so on. Additionally, the sixth bit of Vmem aligns with the MSB of the weight (Wsign) and needs to be kept ‘0’ to correctly read Wsign, which is accomplished by the CS block forwarding Wsign to the next six column peripherals for performing the full 11-bit accumulate operation. This arrangement of column peripherals is similarly followed for 4/6/8-bit weight precision modes.



(a)



(b)

Figure 3.4: (a) Block diagram of column peripheral circuit (b) Carry mux between each column's peripheral circuit

Due to the nature of the ripple carry adder so formed, the combinational logic depth of the circuit grows linearly with number of columns. In order to relax the timing constraints, we need to add pipeline stages as shown. These flip-flops serve the purpose of splitting up the circuit into read, compute and store stages as in a conventional microprocessor architecture. Once the pipeline gets filled, the SRAM is accessed by three addresses simultaneously (two RWLs for reading the operands and one WWL for writing back the result).

3.2.5 In-Memory Instructions

The compute_macro supports four SNN instructions: AccW2V, AccV2V, SpikeCheck and ResetV. Let us go over the operation of these instructions.

1. AccW2V: Accumulate weight to Vmem

This instruction is used to read the weight corresponding to the input spike and add it to the Vmem. To achieve this, one RWL corresponding to the input in the W_{MEM} subarray and one RWL and one WWL from the V_{MEM} subarray is turned on simultaneously. The column peripherals generate the final sum that eventually gets written to the V_{MEM} subarray.

2. AccV2V: Accumulate Vmem to Vmem

This instruction is similar to AccW2V but acts on two rows in the V_{MEM} subarray. It is used to add a predefined parameter (say, a leak factor or residual membrane potential) to the Vmem.

3. SpikeCheck: Compare Vmem to threshold

This instruction is used to compare Vmem to a predefined threshold value. To achieve this, two RWLs from the V_{MEM} subarray (one corresponding to the Vmem and one corresponding to the threshold) are turned on simultaneously. The adders in the column peripheral circuit checks if the sum is greater than zero. The spike buffer is updated if the membrane potential exceeds the threshold.

4. ResetV: Reset membrane potential

Following SpikeCheck instruction, depending on which spike buffers are populated, ResetV is used to write a predefined reset value to the membrane potential. This is achieved by turning on one RWL and one WWL from the V_{MEM} subarray. The column peripheral circuits are effectively bypassed in this instruction.

3.3 Zero Skipping

Energy efficiency in SNNs arise from the inherent sparsity that is present in the input spikes which enables us to skip computation cycles for substantial number of timesteps. The sparsity of input spikes is seen as zeros in our input scratchpad (IF spad). In order to implement zero-skipping, we must only forward entries that are ‘1’. However, checking all entries one by one consumes clock cycles, leaving the accelerator starved. A better approach is to implement a leading-one detector as illustrated in fig. One 16-bit row is fetched from the IF spad and is subjected to a series of bitwise operations. The resultant vector after passing through an encoder gives us the number of indexes to skip to get the next ‘1’ for the CIM queue. This process is repeated till the vector is zero, ensuring fastest possible processing for all input spikes.

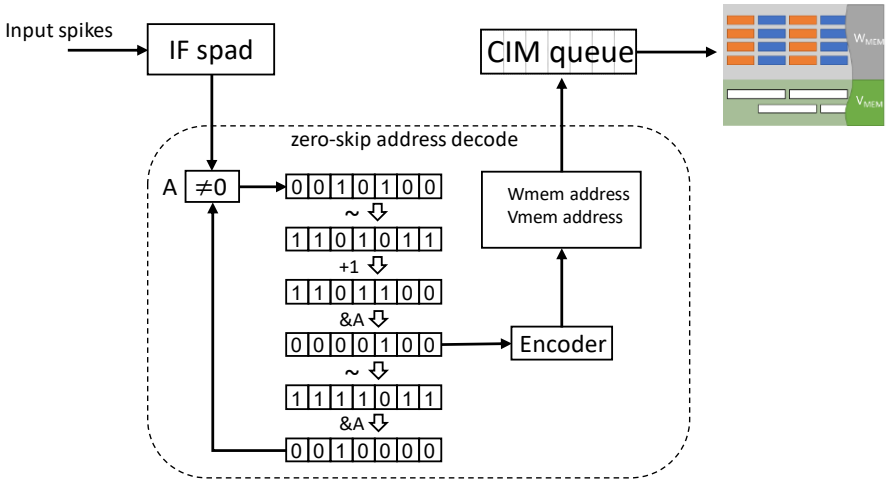


Figure 3.5 Illustration of zero-skipping using leading-one detector.

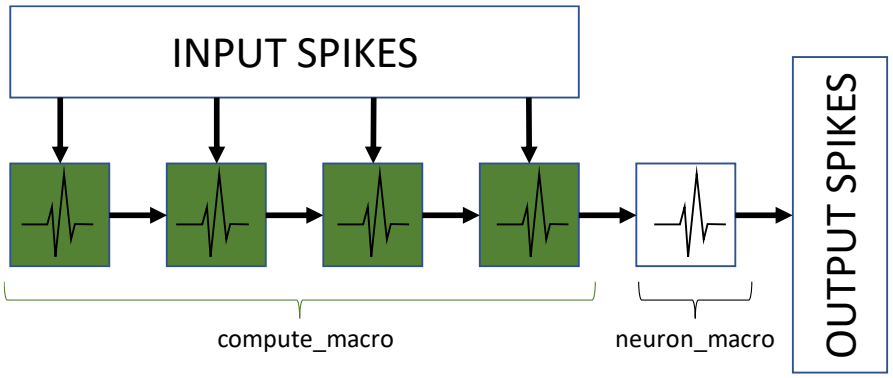


Figure 3.6: Multi-macro implementation for greater input fan-in.

3.4 Neuron Macro

A single compute_macro can be compared dendrites in a biological neuron. Just as dendrites collect the input stimulation and forwards them to the cell body, compute_macro collects the input spikes and generates a partial Vmem, which it eventually forwards to the neuron_macro. One compute_macro has 128 weight rows, which correspond to 128 input fan-in. To increase the input fan-in, several compute_macro can be chained together, where each forwards its partial Vmem to the next compute_macro. At the end of the chain, one neuron_macro collects and accumulates the partial Vmems into final Vmems and generates the output spike.

To efficiently manage and distribute the input spikes among the chain of compute_macros, we implement logic for each macro that accomplishes the following: flattening input spikes before staging them in the scratchpad, implement zero-skipping and finally queuing of input spikes.

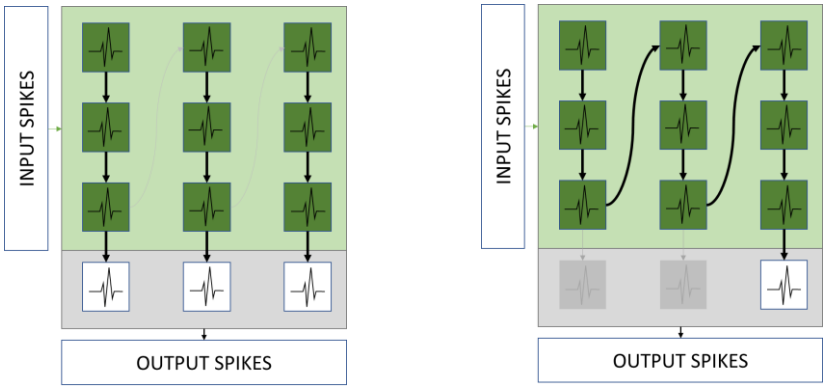
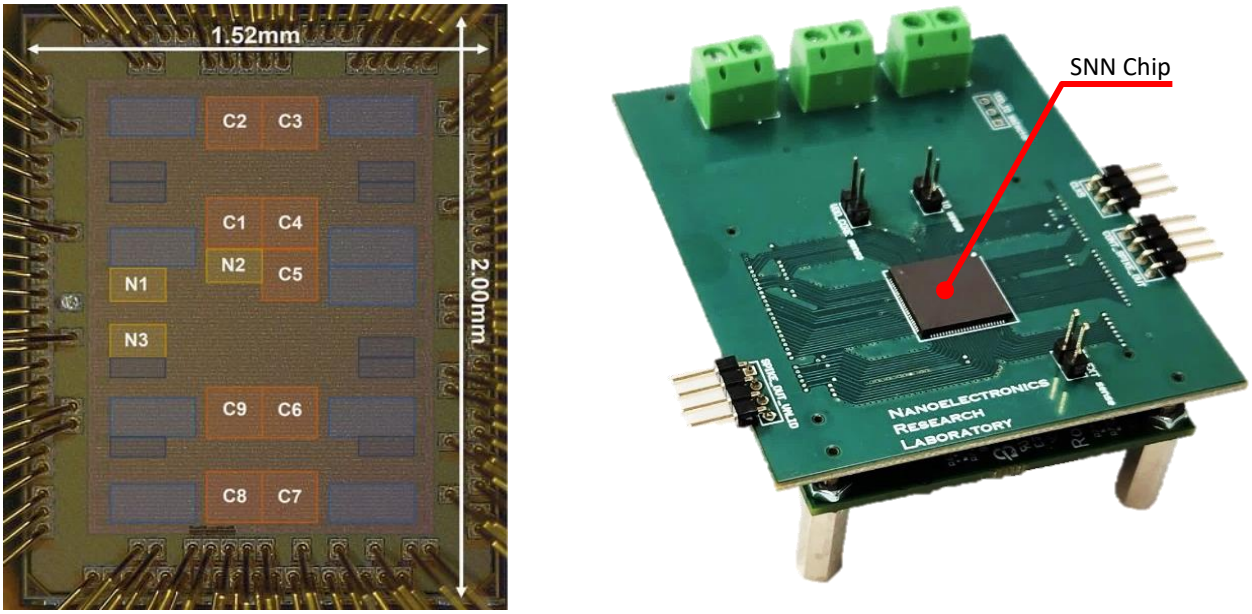


Figure 3.7: (a) Config A: Three compute and one neuron for parallel implementation (b) Config B: Nine compute and one neuron for maximum available fan-in.

3.5 Implementing the SNN accelerator and results

As detailed in the previous sections, the modular approach to the macros allows for great flexibility in implementation. Our accelerator is configured with nine compute_macros and three neuron_macros. This allows us to either have three chains of three compute_macros or a single chain of nine compute_macros. In either case, one neuron_macro forms the last link in the chain,

which is responsible for generating the output. The test chip also incorporates input spike buffers for each compute_macro that keeps it well fed with inputs. The output spike generated by the neuron_macro is written to an output buffer as well as to the output bus.



(a) (b)

Figure 3.8: (a) Die Micrograph; C=compute_macro, N=neuron_macro (b) SNN Accelerator test PCB stacked on top of Opal Kelly XEM 7310 FPGA board

Table 3.2: Chip idle power and leakage power

Freq @ VDD_CORE	**Idle Power (mW)**	**Leakage Power (mW)**
200Mhz @ 1.2v	34.8	0.0732
100Mhz @ 1.0v	11.9	0.0391

Table 3.1: Chip summary

Technology	TSMC 65nm CMOS
Full-chip dimension	2mm x 1.52mm
Supply Voltage	0.8-1.2v
Max Frequency	200Mhz
Weight precision	4/6/8-bit
Output Channels	18-36 (Config A); 6-12 (Config B)

3.6 Conclusion

SNNs are a class of networks that closely mimic how biological neurons function. The simplicity in computation (addition instead of MAC) leads to great energy efficiency. Incorporating fused Vmem and Wmem allows us to avoid majority of the data movement, further lowering energy costs. We presented an SNN accelerator based on the design of IMPULSE macro. The accelerator uses modified compute_macro to support reconfigurable weight precision and number of output channels. Chaining multiple compute_macro allows for a larger fan-in. The neuron_macro forms the end of the chain that generates the final output spike.

SUMMARY

In this thesis, we have established that the functionality of SRAM cells can be extended beyond merely storage of data. In our first work, by employing ROM embedded SRAM cells, we were able store math tables in a pre-existing array. This allowed us to evaluate transcendental functions which form the basis for many neural network applications. The same SRAM array was also used as a bit-serial matrix-vector-multiplication engine, which is another dominant operation in neural network workloads. By combining the two circuits in the same macro, we have shown that it benefits certain classes of workloads. Our analysis with PUMA simulator running an LSTM workload resulted in a speed-up of 1.25x. This came at negligible cost to silicon area as the footprint of the dual-function macro is similar to the footprint of the standalone MVM engine. Additionally, this lowered our energy penalty of fetching data from DRAM.

Our second work deals with accelerating and harnessing energy efficiency of SNNs. SNNs by design mimic the biological neuron through the use of spikes as a method of communication. This binary nature of SNNs makes implementation of neural networks simpler. Repeated shuffling of data between compute and memory structures in accelerators poses a limitation on the maximum throughput and efficiency. From our discussions, it is established that compute-in-memory architectures are a promising alternative to conventional computing as it resolves this memory wall bottleneck. By leveraging the high bandwidth of data movement within SRAMs, we not only gain massive parallelism in dominant operations but also reap the benefits of resulting lower energy consumption. The proposed architecture scales up the modular compute-in-memory enabled IMPULSE macro and as well as implements some key modifications. By incorporating switches in the bitlines, we were able to make weight precision reconfigurable. Also, implementing zero-skip FSM allowed us to extract efficiency out of the inherent sparsity in inputs. We achieved encouraging results from post-silicon test of the accelerator.

REFERENCES

[1] WA Wulf, SA McKee, "Hitting the memory wall: implications of the obvious", ACM SIGARCH Computer Architecture News (Volume 23, Issue 1, March 1995), pp 20–24, DOI: 10.1145/216585.216588

[2] Shien-Yang Wu, et al., “A 3nm CMOS FinFlex™ Platform Technology with Enhanced Power Efficiency and Performance for Mobile SoC and High Performance Computing Applications”, 2022 International Electron Devices Meeting (IEDM), DOI: 10.1109/IEDM45625.2022.10019498

[3] M. F. Ali, R. Andrawis, and K. Roy, "Dynamic read current sensing with amplified bit- line voltage for stt-mrams", IEEE Transactions on Circuits and Systems II: Express Briefs (Volume: 67, Issue: 3, March 2020), DOI: 10.1109/TCSII.2019.2915822

[4] A. Agrawal, A. Jaiswal, C. Lee, and K. Roy, "X-SRAM: Enabling In-Memory Boolean Computations in CMOS Static Random Access Memories", IEEE Transactions on Circuits and Systems I: Regular Papers (Volume: 65, Issue: 12, December 2018), DOI: 10.1109/TCSI.2018.2848999

[5] A. Agrawal, A. Jaiswal, D. Roy, et al., " Xcel-RAM: Accelerating Binary Neural Networks in High-Throughput SRAM Compute Arrays", IEEE Transactions on Circuits and Systems I: Regular Papers (Volume: 66, Issue: 8, August 2019), DOI: 10.1109/TCSI.2019.2907488

[6] A. Jaiswal, I. Chakraborty, A. Agrawal, and K. Roy, " 8T SRAM Cell as a Multibit Dot-Product Engine for Beyond Von Neumann Computing" IEEE Transactions on Very Large Scale Integration (VLSI) Systems (Volume: 27, Issue: 11, November 2019), DOI: 10.1109/TVLSI.2019.2929245

[7] C. Eckert, X. Wang, J. Wang, et al., "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks ", 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), DOI: 10.1109/ISCA.2018.00040

[8] J. Von Neumann, “The computer and the brain”, Yale University Press, 2012.

[9] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, et al., " PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference”, in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 2019 https://doi.org/10.1145/3297858.3304049

[10] I. Chakraborty, M. Fayez Ali, D. Eun Kim, A. Ankit, and K. Roy, " GENIEx: A Generalized Approach to Emulating Non-Ideality in Memristive Xbars using Neural Networks" in 2020 57th ACM/IEEE Design Automation Conference (DAC), DOI: 10.1109/DAC18072.2020.9218688

[11] J. Hennesy and D. Patterson, “Computer Architecture: A Quantitative Approach”. San Mateo, CA: Morgan Kaufman, 2007.

[12] L. Chang, R. K. Montoye, Y. Nakamura, K. A. Batson, R. J. Eickemeyer, R. H. Dennard, W. Haensch, and D. Jamsek, “An 8T-SRAM for Variability Tolerance and Low-Voltage Operation in High-Performance Caches” IEEE Journal of Solid-State Circuits (Volume: 43, Issue: 4, April 2008), DOI: 10.1109/JSSC.2007.917509

[13] J. Harrison, T. Kubaska, S. Story, and P. Tang, “The computation of transcendental functions on the IA-64 architecture,” Intel Technology Journal Q4, Nov. 1999.

[14] M. Davies et al., “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning” IEEE Micro (Volume: 38, Issue: 1, January/February 2018), DOI: 10.1109/MM.2018.112130359

[15] J. S. P. Giraldo et al., “Laika: A 5uW Programmable LSTM Accelerator for Always-on Keyword Spotting in 65nm CMOS” in ESSCIRC 2018 - IEEE 44th European Solid State Circuits Conference (ESSCIRC), DOI: 10.1109/ESSCIRC.2018.8494342

[16] S. Narayanan et al., “Spinalflow: An architecture and dataflow tailored for spiking neural networks,” in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 349–362.

[17] A. Agrawal, M. Ali, M. Koo, N. Rathi, A. Jaiswal, and K. Roy, “IMPULSE: A 65-nm Digital Compute-in-Memory Macro With Fused Weights and Membrane Potential for Spike-Based Sequential Learning Tasks” IEEE Solid-State Circuits Letters (Volume 4), DOI: 10.1109/LSSC.2021.3092727

[18] B. Han et al., “RMP-SNN: Residual Membrane Potential Neuron for Enabling Deeper High-Accuracy and Low-Latency Spiking Neural Network,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, pp. 13558-13567.

[19] D. Lee, K. Roy, "Area Efficient ROM-Embedded SRAM Cache", IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 21, NO. 9, SEPTEMBER 2013

[20] M. Ali, K. Roy "A 65 nm 1.4-6.7 TOPS/W Adaptive-SNR Sparsity-Aware CIM Core with Load Balancing Support for DL workloads " IEEE Custom Integrated Circuits Conference (CICC), 23 - 26 Apr 2023