2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED) | 978-1-6654-3922-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISLPED52811.2021.9502484

Statistical Optimization of Compute In-Memory Performance Under Device Variation

Brian Crafton[1], Samuel Spetalnick[1], Jong-Hyeok Yoon[2], and Arijit Raychowdhury[1]

[1]Georgia Institute of Technology
[2]Daegu Gyeongbuk Institute of Science and Technology
arijit.raychowdhury@ece.gatech.edu

***Abstract*—Compute in-memory (CIM) is a promising technique that minimizes data transport, maximizes memory throughput, and performs computation on the bitline of memory sub-arrays. Utilizing embedded non-volatile memories (eNVM) such as resistive random access memory (RRAM), various forms of neural networks can be implemented. Unfortunately, CIM faces new challenges traditional CMOS architectures have avoided. In this work, we explore the impact of device variation (calibrated with measured data on foundry RRAM arrays) and propose a new algorithm based on device variation to increase both performance and accuracy for CIM designs. We demonstrate up to 36% power improvement and 44% performance improvement, while satisfying any error constraint.**

I. Introduction

Over the last decade, tremendous progress towards accelerating machine learning workloads has been made at all levels of the computing hierarchy, enabling orders of magnitude improvement in energy efficiency. At the software level, models are compressed, pruned, and quantized to minimize the total size of the model and cost of a single inference [1]. At the hardware level, prior work focuses on maximizing the reuse of all data such that expensive memory accesses and total data movement is minimized [1]. Both of these strategies focus on minimizing the cost of data movement and memory accesses, while maximizing the utility of available on-chip memory capacity. While these techniques yield strong results, they still face the fundamental technological limitations of CMOS. In particular, the large size of the SRAM bitcell ($\approx 100 - 150F^2$) results in limited on-die capacity, which necessitates movement of data from an external DRAM to the on-die SRAM at more energy per bit.

Fortunately a new class of high density eNVM is positioned to minimize data movement and increase memory throughput by performing CIM. CIM performs vector matrix multiplication (VMM) ($\vec{y} = W\vec{x}$) by mapping the computation to the analog domain. Weights (W) are programmed to the non-volatile resistive state of eNVM, input vectors ($\vec{x}$) are applied as voltages, and the sum of currents along the bitline serves as the result of VMM, $\vec{y}$. Under this implementation, the only data transport required for VMM is the input vector ($\vec{x}$) from memory and result ($\vec{y}$) to the memory. Despite these benefits, a key obstacle in designing a reliable CIM accelerator with eNVM is the inherent cell-to-cell variation in the device's resistive state. These variations are not specific to eNVM, and occur due to process and temperature or write-to-write (cycle-to-cycle) variations. Conventional digital memory such as SRAM overcomes this challenge using differential sensing and a large ratio between the '0' and '1' states. However, when reading multiple memory cells at the same time with an analog-to-digital converter (ADC), high variation between resistive states results in sum-of-products errors accumulated on the bitline. Given that these operations are used to implement VMM, and thus neural networks, we find that device level variation results in erroneous computation. While neural networks can tolerate these errors to some extent, inference performance degrades as a function of the these errors.

Recent work has attempted to mitigate the impact of these errors in several different ways. Training a network to be robust to device variation induced error can be done both off-chip and on-chip. Off-chip training attempts to train a neural network to tolerate device variation induced errors [2], however this technique still results in accuracy degradation. On-chip training [3] can be done to minimize error for a specific chip, however this is expensive since each chip must be trained based on its own specific devices. Lastly, write-verify methods have been proposed to reduce the cell-to-cell variation [4]. However these methods require high write energy and latency, and greatly reduce the endurance of the device.

In this work, we present a new technique to minimize variation-induced errors by controlling the number of wordlines read in parallel. Ideally, all wordlines for an array (128, 256) can be read to enable massive thoughput. However, the more wordlines we read, the higher the error due to the accumulation of device variation. Recent work [5], [6] explore this tradeoff to maximize performance for a given inference performance target. Building upon this idea, we construct an accurate error model for VMM based on RRAM device

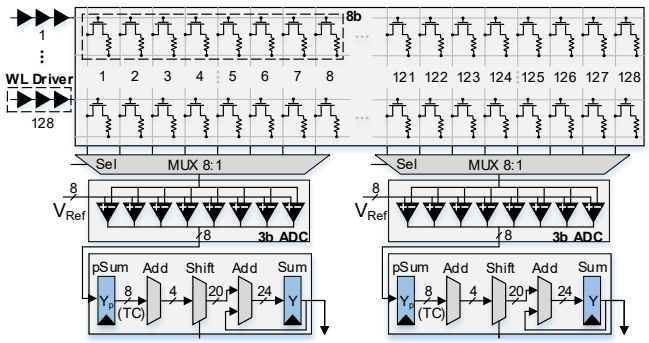


Fig. 1. 128×128 RRAM array architecture. 8 adjacent cells form an 8-bit weight and share a 3-bit ADC through a 8-to-1 multiplexer. Shift and add logic accumulate partial sums and apply corresponding magnitude.

978-1-6654-3922-0/21/$31.00 ©2021 IEEE

variation. We calibrate our variation models with experimental data collected from a 40nm foundry RRAM test-chip array, whose circuit details appear in [7]. Next, we formulate an optimization problem to achieve optimal performance for a given error constraint. We benchmark on ImageNet using ResNet18 and demonstrate a 36% power and 44% performance improvement, while satisfying any error constraint.

II. Background and Motivation

A. Compute In-Memory (CIM)

To implement VMM ($\vec{y} = W\vec{x}$), CIM systems encode the input vector $\vec{x}$ as wordline voltages and the weight matrix W as conductance states in a memory cell. The current through each cell is proportional to the product of the programmed conductance (W_{ij}) and applied voltage ($\vec{x}_i$) (Ohm's Law). By Kirchhoff's current law (KCL), the resulting currents that are summed along the columns of the crossbar are proportional to the product of the matrix and vector, ($\vec{y}$).

Typically, each memory cell only stores 1 or 2 bits and thus weights (W_{ij}) requiring higher precision are encoded as several cells on the same wordline. Similarly, input voltages are limited to 1 bit and vector values ($\vec{x}_i$) are input to the wordline over several cycles. Therefore, to implement 8-bit VMM with 1-bit cells and 1-bit voltages, we must use 8 adjacent cells for each weight (Fig. 1) and 8 voltage pulses for each input. This encoding scheme results in 64 (8 cells $\times$ 8 cycles) partial sums that we must shift and sum to generate the full 8-bit VMM. In other words, 8-bit VMM is implemented using 1-bit inputs and 1-bit cells as the shifted sum of 64 (8×8) binary VMM.

If the number of wordlines in the array exceeds the precision of the ADC (i.e. 16 wordlines for a 4-bit ADC), the ADC overflows and results in erroneous VMM. To resolve this, the number of wordlines read should be limited to the maximum precision of the ADC. As a result, performance is reduced since the wordlines must be read over several cycles, where the partial sums from these cycles are summed together. However, it is important to note that only wordlines equal to '1' are read, and thus we can process more wordlines when '0's exist in the input data. This technique is called *zero-skipping* [8], and can be used to achieve significant speedup since the activations of neural networks are typically very sparse.

To implement signed VMM using CIM, there are two common implementations. For signed weights and unsigned inputs, two's compliment can be used. For each binary VMM involving the most significant bit (MSB) of the weight, we simply scale by -2^7 instead of 2^7. For signed weights and signed inputs, an offset representation is used for both inputs and weights, where a bias is subtracted after the VMM. For signed 8-bit VMM, 2^7 is added to both inputs and weights, and after VMM is performed the bias is computed and subtracted from the result. The downside of this approach is that the sum of the input vector must be computed and then subtracted from each output value.

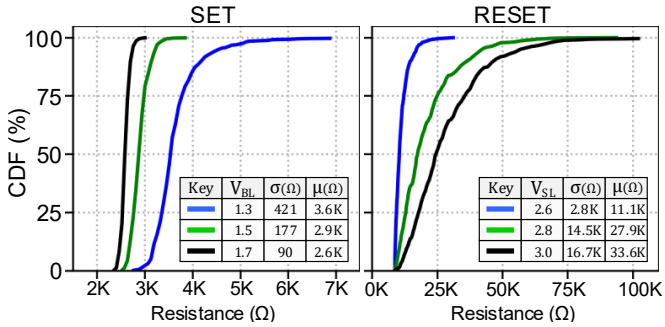


Fig. 2. CDF of measured resistance values for various write voltages for (A) set operation and (B) reset operation [7].

B. Quantifying the Impact of Device Variations

CIM seeks to read and accumulate several states on a bitline at once, and therefore a key obstacle to enabling CIM is cell-to-cell variation. These variations are typically normally distributed [4], [7] and measured as the standard deviation (σ) from the mean (μ) resistance value. For binary (2-level) cells, a digital '0' is encoded as the high resistance state (HRS) and a digital '1' is encoded as the low resistance state (LRS). Ideally, current from reading a cell in the HRS could be ignored if the difference between between the HRS and LRS (on/off ratio) were several orders of magnitude. Unfortunately, this is not the case as recent RRAM [7] and PCM [4] demonstrate an on/off ratio between $10\times$ to $100\times$. Thus to accurately quantify device variation we model the standard deviation as a percentage of the mean for LRS and HRS (σ_L/μ_L, σ_H/μ_H) and the on/off ratio between HRS and LRS (μ_H/μ_L).

Recent demonstrations of RRAM [7] and PCM [4] show low LRS variation (3.5%) and high HRS (50%) variation with an on/off ratio of $10\times$ to $100\times$ depending on how the cells are written [4], [7]. Using higher write voltages and an iterative write verify protocol, lower variation and higher on/off ratio can be achieved. To quantify the variation, we measure the resistance values of RRAM cells from a recent RRAM test-chip prototype on a 40nm foundry RRAM array [7]. The array contains 256×256 RRAM cells (64Kb). The details of the read and write circuit of the array are beyond the scope of this paper and interested readers are pointed to [7] for further discussions. Instead, in this paper, we use the measured data to calibrate our RRAM models. To understand the impact of the write voltages, we used three different configurations for both the set (LRS) and reset (HRS) operations. The CDF of these measurements as well as the corresponding μ and σ are shown in Figure 2. While higher write voltages and iterative write verify can reduce device variation and increase on/off ratio, there are several drawbacks. First, these techniques increase write energy and latency. Second, they greatly reduce the endurance of RRAM and most other eNVM [7]. Therefore, in this work we consider a range of device models with different variation and on/off ratio parameters. These models are consistent with other measured data from existing literature [4] and provide high confidence on LRS and HRS distributions.

III. Error Model and Optimization

The simplest way to control accumulated variation-induced error is to reduce the number of word lines enabled at one time. Of course, this will unfortunately compromise the performance and energy efficiency of using CIM. However, we have little choice in this trade-off because target applications require certain accuracy to be useful. Therefore, our objective function becomes achieving a target accuracy while maximizing performance. In this way, the variation of the device should ultimately dictate the precision of the ADC used.

As we discussed in Section II, VMMs are mapped to CIM arrays by dividing the VMM into multiple lower precision VMMs that can be mapped to an array's wordlines (1-bit) and memory cells (1-bit, 2-bit). If we again assume 8-bit VMM, 1-bit wordlines, and 1-bit cells, we perform 64 binary VMMs that are then shifted by the corresponding magnitude of the input and weight values mapped to it. For the binary VMM between the least significant bit (LSB) of both the inputs and weights, we shift by 0 (or multiply by 1). For the binary VMM between the most significant bits (MSB), we shift by 14 (or multiply by 16384). Therefore, variation induced errors during the binary VMMs are also shifted by this magnitude. Thus errors that occur for the most significant bit (MSB), are exponentially more destructive to the result of the VMM.

To illustrate this idea, we can extract the VMM performed by a convolutional layer and identify how much error each binary VMM contributes. We use mean absolute error (MAE) where if we let the true result of VMM be: $y = W\vec{x}$, and the CIM result be: $\hat{y} = \hat{W}\vec{x}$, then MAE is $\sum|\hat{y} - y|$. As an example, we extract all the 8-bit VMMs performed during a single inference of 2 different layers in 8-bit ResNet18 on ImageNet. Using the simulator detailed in Section V, we simulate the VMM using devices based on Figure 2 with 6.2% LRS variation and 50% HRS variation and 3-bit ADCs.

We show the average MAE breakdown across the 64 binary VMMs that occurs when performing an 8-bit VMM in Figure 3. The rows of the table correspond to the magnitude of the input bit (X_0-X_7) and columns of the table correspond to the magnitude of the weight bit (W_0-W_7). Each bin of the table contains the percentage of total MAE that can be attributed to the corresponding binary VMM. It is clear that the majority of MAE comes from high magnitude VMMs, while low magnitude VMMs contribute near-zero MAE to the final output. Thus, if our design allows we can operate these binary VMM at different speed (number of wordlines) as a function of their MAE and magnitude. This idea is the basis of our work, and in the following sections we discuss how to model VMM error and optimize CIM performance.

A. Modeling VMM Error

In this work we model VMM error as mean absolute error (MAE). Therefore, if we let the true result of VMM be: $\hat{y} = W\vec{x}$, and the CIM result be: $\hat{y} = \hat{W}\vec{x}$, then MAE is $\sum|\hat{y} - y|$. To compute the expected MAE for an 8-bit VMM (assuming 1-bit inputs & weights), we must model the MAE in a bottom-up fashion starting with the the ADC. From the ADC MAE model, we can compute the expected MAE for a single binary VMM, and then lastly the 8-bit VMM. In the following sections, we break down the MAE computation into these three levels of abstraction.

Layer 1

	W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7
X_0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.3
X_1	0.0	0.0	0.0	0.1	0.1	0.2	0.4	0.8
X_2	0.0	0.0	0.1	0.1	0.2	0.2	0.5	0.9
X_3	0.0	0.1	0.1	0.1	0.3	0.4	0.8	1.9
X_4	0.1	0.1	0.1	0.2	0.6	0.8	0.8	3.7
X_5	0.1	0.2	0.3	0.5	1.4	1.9	4.4	9.3
X_6	0.2	0.3	0.5	0.9	2.6	3.5	7.9	14.8
X_7	0.2	0.4	0.6	1.1	2.9	3.9	8.8	19.1

Layer 4

	W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7
X_0	0.0	0.0	0.0	0.1	0.1	0.3	0.7	1.0
X_1	0.0	0.0	0.1	0.1	0.3	0.5	1.3	2.0
X_2	0.0	0.1	0.1	0.3	0.5	1.0	2.5	4.0
X_3	0.0	0.1	0.3	0.5	1.0	2.0	4.8	7.5
X_4	0.1	0.2	0.4	0.9	1.7	3.3	8.0	12.3
X_5	0.1	0.2	0.5	0.9	1.8	3.6	8.9	14.1
X_6	0.0	0.1	0.2	0.3	0.6	1.3	3.4	5.4
X_7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Fig. 3. Each of the 64 binary VMM contribution to total MAE for layers 1 and 4 of ResNet18 on ImageNet.

*1) **ADC**:* The expected error at each ADC can be computed as a function of the device variation and the number of wordlines enabled. Ideally, the output of the ADC is the number of LRS cells read since an LRS cell corresponds to a digital '1'. However, the cumulative current through the memory cells can produce erroneous ADC output codes due to cell-to-cell variation. Given the standard deviation (σ_L, σ_H) of the resistive states (LRS and HRS) of the memory cells, we can compute a PMF for the ADC output codes. The probability of observing an ADC code, C, given the actual number of LRS cells, N_L, can be computed using the normal CDF (Φ) function. Equation 1 demonstrates this, and models $P(C \mid N_L)$, the probability of the cumulative current from N_L LRS cells being read by the ADC as C.

$$= \Phi\left(\frac{C - N_L + 0.5}{\sqrt{\sigma_L^2 N_L + \sigma_H^2 N_H}}\right) - \Phi\left(\frac{C - N_L - 0.5}{\sqrt{\sigma_L^2 N_L + \sigma_H^2 N_H}}\right) \quad (1)$$

*2) **Binary VMM**:* Equation 1 provides the probability of an ADC code given that N_L LRS cells are read. Given that each vector-vector product in the binary VMM is the sum of ADC output codes along the bitline, we can compute the expected MAE of a binary VMM by identifying the PMF of these ADC codes, $P(N_L)$, and applying Equation 1. $P(N_L)$ depends on the inputs and weights of the binary VMM under consideration. To acquire $P(N_L)$, we can profile our neural network in simulation *assuming ideal devices*, and perform the 64 binary VMMs, in place of a 8-bit VMM, while counting the number of times each ADC output value occurs. This gives as an approximation for $P(N_L)$, and as an example, we profile layer 1 of 8-bit ResNet18 on ImageNet. We visualize this in Figure 4, showing different distributions for the different binary VMM in the 8-bit VMM when reading 16 wordlines.

So far we have obtained $P(N_L)$, the PMF of ideal (zero device varition) ADC output codes, and $P(C \mid N_L)$, the PMF of obtaining an ADC code, C, given the actual number of LRS cells, N_L. Using these PMFs, we can compute the expected MAE for a binary VMM by summing the MAE for all possible outcomes. We implement this by first iterating over the set of possible N_L, the number of LRS enabled along a bitline. This set ranges from 0 to the number of wordlines enabled, N_{WL}. For each value of N_L, we iterate over all possible ADC

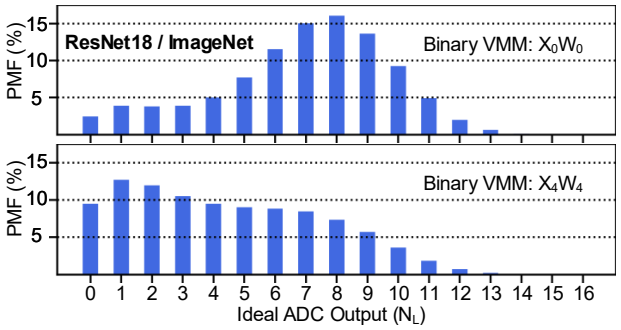


Fig. 4. $P(N_L)$ for layer 1 of ResNet18 on ImageNet.

output codes: N_C. Next, we compute the probability for each pair (N_L, C) and the associated error. Naturally, the error for each pair is simply the difference: $C - N_L$. The final part of our model comes from the fact that a binary VMM contains columns where we may read more than N_{WL} wordlines. As discussed in Section II, we break down these columns into several cycles to not overflow the ADC. Therefore, we scale our MAE calculation by N_{tot}/N_{WL}, where N_{tot} is the expected number of wordlines enabled along a column. Our complete model for a binary VMM is given in Equation 2.

$$E_{xw} = \sum_{N_L}^{N_{WL}} \sum_{C}^{N_C} \frac{N_{tot}}{N_{WL}} \cdot P(C \mid N_L) \cdot P(N_L) \cdot |C - N_L| \quad (2)$$

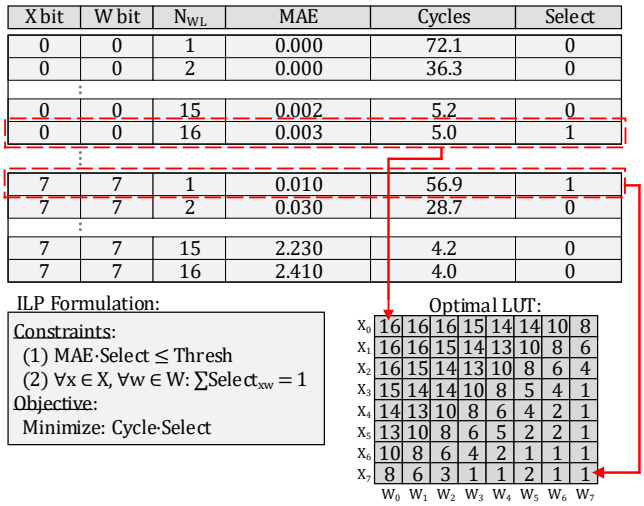
*3) **8-bit VMM**:* Building off of the binary VMM model, we can compute the expected MAE for the 8-bit VMM as the sum of MAE across the 64 binary VMM scaled by their corresponding magnitude. Our final MAE approximation for a matrix multiplication is given in Equation 3.

$$E_{VMM} = \sum_{x}^{X} \sum_{w}^{W} 2^x 2^w \cdot E_{xw} \quad (3)$$

B. Optimizing Operation Speed

After establishing a methodology for computing the expected MAE for 8-bit VMM, we can now formulate an optimization problem. In our optimization problem, we consider the relationship between MAE and performance and we attempt to optimize CIM performance for a given MAE constraint. Since each of the 64 binary VMMs will yield different error rates, we choose to operate each one at a different number of wordlines per cycle. Thus, the solution to our optimization problem will be a lookup table (LUT) containing 64 (8×8) values indicating the number of wordlines per cycle to be performed by the corresponding binary VMM.

To find the optimal LUT, we compute the expected MAE and performance for all 64 binary VMM operating at various numbers of wordlines per cycle. We then compile these results into a table. To illustrate this, we provide an example table in Figure 5. In this example, we perform 8-bit matrix multiplication and we read up to 16 wordlines at a time, thus our table will be $8 \times 8 \times 16$. For each of the 64 binary VMM, we must select between 1 wordline and 16 wordlines in a single cycle. This is visualized in the far right column of Figure 5, where one of the 16 conditions will be chosen for



X bit	W bit	N_{WL}	MAE	Cycles	Select
0	0	1	0.000	72.1	0
0	0	2	0.000	36.3	0
⋮					
0	0	15	0.002	5.2	0
0	0	16	0.003	5.0	1
⋮					
7	7	1	0.010	56.9	1
7	7	2	0.030	28.7	0
⋮					
7	7	15	2.230	4.2	0
7	7	16	2.410	4.0	0

ILP Formulation:

Constraints:
(1) MAE·Select ≤ Thresh
(2) ∀x ∈ X, ∀w ∈ W: ∑Select$_{xw}$ = 1
Objective:
Minimize: Cycle·Select

Optimal LUT:

	W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7
X_0	16	16	16	15	14	14	10	8
X_1	16	16	15	14	13	10	8	6
X_2	16	15	14	13	10	8	6	4
X_3	15	14	14	10	8	5	4	1
X_4	14	13	10	8	6	4	2	1
X_5	13	10	8	6	5	2	2	1
X_6	10	8	6	4	2	1	1	1
X_7	8	6	3	1	1	2	1	1

Fig. 5. Example LUT optimization problem formulation.

each binary VMM. Although it is just an example, we show two cases where N_{WL} is chosen. For the lowest magnitude binary VMM (X_0-W_0), N_{WL}=16 is chosen since the MAE is relatively low for the performance gain. For the highest magnitude binary VMM (X_7-W_7), N_{WL}=1 is chosen because of the high impact a single error has on the VMM. The objective of this optimization problem is to satisfy an MAE constraint, while maximizing performance.

Upon formulating this problem we note that it is, by definition, a variant of the knapsack problem called multi-class knapsack. The weight (or cost) of each choice in the table is the expected MAE from operating the binary VMM at N_{WL} per cycle. The value of each choice is the latency required for the binary VMM, making our objectice function minimization because we seek to minimize latency. We compute this latency as the expected number of cycles required to perform the binary VMM. For example, in Figure 5, the average N_{tot} for the first binary VMM (X_0, W_0) is 72.1 cycles. Thus operating at 1 wordline per cycle (N_{WL}=1), we can expect to finish the binary VMM in 72.1 cycles. However, at 16 wordlines per cycle (N_{WL}=16), we can expect to finish in just 5 cycles.

The constraints to our optimization problem are two fold. First, the sum of the MAE of all the selected binary VMM must be less than or equal to our MAE constraint. Second, exactly 1 N_{WL} must be chosen for each of the 64 binary VMM, hence multi-class knapsack. The solution to this optimization problem provides us with the optimal operation speed (N_{WL}) for each binary VMM that maximizes performance while satisfying our MAE constraint. This can be solved with a greedy heuristic or a common optimization technique like branch and bound or integer linear programming (ILP). Despite being an NP-hard problem, this instance of the multi-class knapsack is quite small and is solved very fast (<10ms) using ILP implemented with the *CVXOPT* package in Python.

IV. CIM-based Architecture

For our experiments we adopted a similar architecture to previous work [9]. Our basic processing element (PE) contains

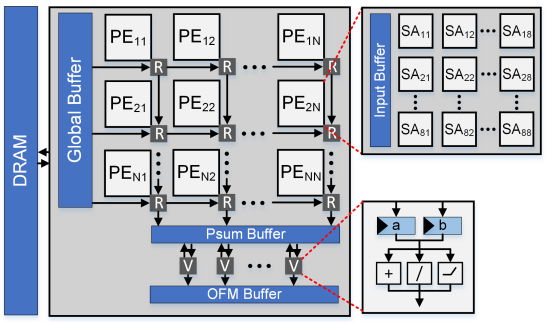


Fig. 6. System architecture with 1 router (R) per PE.

32 256×256 RRAM arrays. Each array has 1 ADC for every 8 columns. To better understand how our algorithm applies to different ADCs we try both Flash and SAR designs. For the Flash ADC, we use 3-bit precision where a single bitline is pitch-matched with a comparator to minimize peripheral overhead and maximize memory density. For the SAR ADC, we find that an area optimized 6-bit SAR ADC takes approximately the same area as a 3-bit Flash ADC.

To cache input data, each PE contains a 4096KB SRAM cache with a 128-bit bus. To feed the 32 RRAM arrays with data, a CMOS-based memory controller reads from the SRAM cache and provides the RRAM arrays with input data. This controller contains a state machine that iterates through all 64 binary VMM, and issues the correct input data (X_0-X_7) based on the corresponding binary VMM. In a standard design, like ISAAC [9], this memory controller would issue a input vectors with a fixed number of '1's based on the precision of the ADC. However in our design, we control the number of wordlines based on the corresponding binary VMM. Hence, we add a small programmable LUT to this controller containing 64 entries for the 64 binary VMM. To set the speed we require $\log_2(N_{WL})$ bits per binary VMM. Therefore if we enable up to 64 wordlines per cycle, we require a 48 byte LUT, which is negligible compared the large SRAM and RRAM banks.

The activation inputs to the RRAM sub-arrays are stored in on-chip SRAM, while the input images are read in from external DRAM. Matrix multiplication is performed by the PEs, while custom vector units are used to perform vector-wise accumulation, bias addition, quantization, and ReLU. Because all inputs in a CNN are positive valued, we can use two's complement representation for our weights and avoid offset format (Section II). We use a $N \times N$ mesh network for communication between PEs, memory, and vector units shown in Figure 6. To avoid reprogramming the RRAM

Component	# Unit	Specification	Energy
Processing Element (PE)			
Input Cache	1	128b SRAM (64KB)	64 fJ / bit
Output Cache	1	64b SRAM (32KB)	62 fJ / bit
Vector Unit	1	ReLU, Quantize	81 fJ / op
Sub-Array	16	See Below	See Below
Sub-Array (SA)			
RRAM Array	1	1-bit/cell 256×256	1.1 fJ / bit
ADC	32	Flash 3-bit / 1 cycle	45 fJ / conv-step
		SAR 6-bit / 6 cycle	22 fJ / conv-step
Shift + Add	32	24-bit Shift, 24-bit Add	101 fJ / op
DFF	32×24	24-bit	85 fJ / bit

Fig. 7. Simulation parameters used for hardware components.

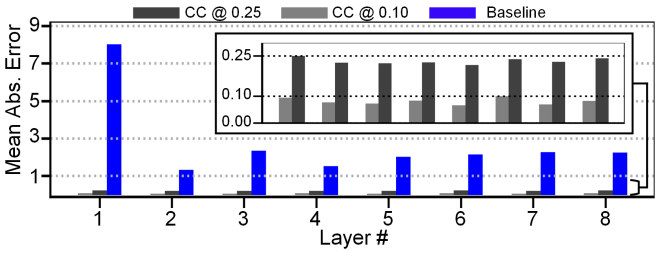


Fig. 8. MAE by layer for the first 8 layers of ResNet18 on ImageNet.

arrays due to high write energy and latency, we store all layers on chip like prior work [9]. To maximize throughput, we pipeline the layers to that all RRAM arrays are active. Furthermore, Given that total RRAM capacity nearly doubles the size of 8-bit ResNet18 (33.5MB vs 17.6MB), we apply weight duplication [9]. To choose which layers are duplicated, we apply performance-based array allocation [8] to maximize throughput. This technique profiles the workload and evenly allocates arrays to each layer to prevent a bottleneck.

V. Results

To benchmark our algorithm (from Section III) we call *counting cards*, we compare against a baseline design. For both *counting cards* and baseline, we use the same architecture described in Section IV. The only difference is *counting cards* sets the N_{WL} for each binary VMM to optimize for a given MAE constraint, while baseline operates based on the precision of the ADC. We evaluate performance, power, and accuracy on ImageNet using ResNet18.

A. Simulation Framework

Our simulator performs cycle-accurate implementations of convolutional and fully connected layers. It is based in Python, but runs array level operations in C for faster evaluation. We model components in the design in object oriented fashion, iterating through all components in all PEs each cycle. We embed performance counters in our ADC and sub-array objects to track metrics like stalls so we can calculate utilization. As input, the simulator takes the network weights, input images, PE level configuration, and chip-level configuration. The PE-level configuration includes details like the precision of each ADC and size of the sub-array. The chip-level configuration contains the number of PEs and details about array allocation and mapping. As output, the simulator produces a table with all desired performance counters and all intermediate layer activations that are verified against a TensorFlow implementation for correctness. We evaluate power and performance using 32nm CMOS and ADC models adopted from NeuroSim [10] and displayed in Figure 7. All code and simulation parameters to recreate results is available at: https://github.com/bcrafton/speed_read.

B. Simulation Results

To generate the 64 entry LUT for each layer of ResNet18, we first acquire the distribution of ADC output codes and then optimize performance for a target MAE constraint. Initially, we perform inference on 100 images and profile the distribution

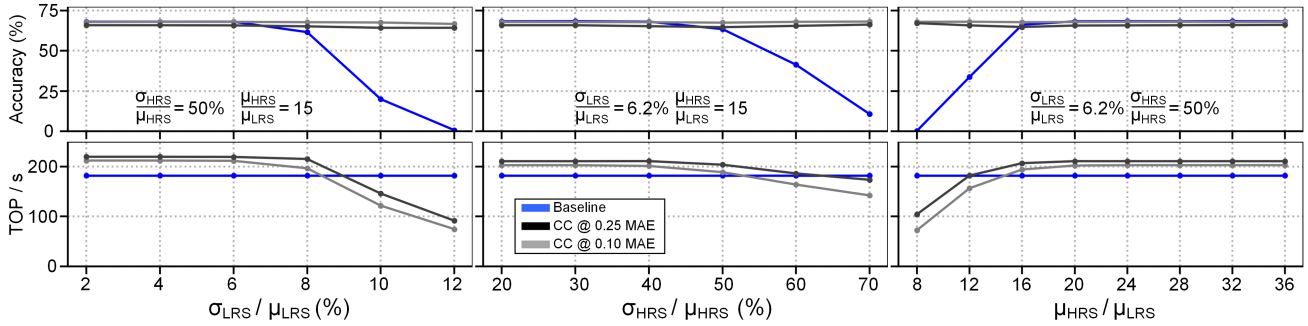


Fig. 9. Performance and Accuracy (ResNet18) of *counting cards* versus the baseline design.

of the ADC output codes. Because this distribution depends on N_{WL}, we perform each of the 100 inferences using between 1 and 128 wordlines, for a total of 12,800 inferences. Using an Nvidia RTX 2080 GPU, this process took 84 seconds. While this significantly longer than simply running 12,800 inferences, it is only done once per model and representative dataset. After acquiring the distribution, we compute the optimal LUT for each layer based on device models shown in Figure 2 and an MAE constraint which we sweep to explore the accuracy-performance tradeoff.

In Figure 9, we show the accuracy and performance of *counting cards* and our baseline design, for various device models and target MAE constraints (0.1, 0.25). Our MAE constraints are chosen as 0.1 and 0.25 because we find that this roughly translates to $< 0.25\%$ and $< 1\%$ accuracy degradation on ResNet18. For this simulation we use the 3-bit Flash ADC. For σ_L, σ_H, and the on/off ratio, we sweep a range of parameters determined by our data collected from the 40nm RRAM array (Fig. 2). For Figures 9A and 9B, we show accuracy and performance versus σ_L. Over this range, we find that counting cards maintains a near constant accuracy independent of σ_L. For low values of σ_L, a performance speedup is achieved ($1.21\times$), while for higher values performance slows down to maintain low MAE. While sweeping σ_H and on/off ratio, we observe similar behavior. However it is clear that in this parameters given by Fig. 2, σ_L has the highest impact. In Figure 8, we show MAE by layer for σ_L = 12%. This figure demonstrates how counting cards keeps precise control of MAE at each layer based on the MAE constraint, while the baseline design exhibits high MAE.

To compare power and performance of counting cards using a 6-bit SAR ADC and 3-bit Flash ADC, we plot results versus

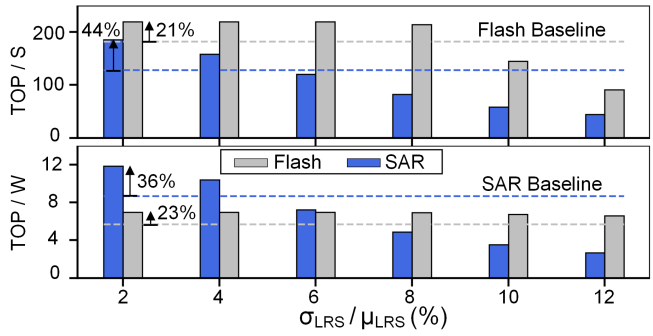


Fig. 10. Power & performance of counting cards using Flash and SAR ADC.

σ_L in Fig. 10. It should be noted that baseline performance and power is presented as a dotted reference line because it does not change as a function of device variation. As expected the SAR ADC yields significantly better energy efficiency over the Flash, while achieving less performance. At low σ_L, counting cards enables 44% improved TOP/s and 36% better TOP/W for the SAR ADC. For the Flash ADC we observe 21% improved TOP/s and 23% improved TOP/W. Interestingly, for the SAR ADC the TOP/W & TOP/s decreases greatly as σ_L increases while the Flash remains near constant. This is because the Flash operates at 8 wordlines per cycle instead of 64, and thus it is far less sensitive to higher σ_L.

VI. Acknowledgement

This work was funded by the U.S. Department of Defense's Multidisciplinary University Research Initiatives (MURI) Program under grant number FOA: N00014-16-R-FO05 and the Semiconductor Research Corporation under the Center for Brain Inspired Computing (C-BRIC).

References

[1] B. Crafton *et al.*, "Merged logic and memory fabrics for accelerating machine learning workloads," *IEEE Design & Test*, 2020.

[2] Y. Long *et al.*, "Design of reliable dnn accelerator with un-reliable reram," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1769–1774, IEEE, 2019.

[3] X. Sun and S. Yu, "Impact of non-ideal characteristics of resistive synaptic devices on implementing convolutional neural networks," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 570–579, 2019.

[4] J. Wu *et al.*, "A 40nm low-power logic compatible phase change memory technology," in *2018 IEEE International Electron Devices Meeting (IEDM)*, pp. 27–6, IEEE, 2018.

[5] A. S. Rekhi *et al.*, "Analog/mixed-signal hardware error modeling for deep learning inference," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.

[6] Y. Park *et al.*, "Unlocking wordline-level parallelism for fast inference on rram-based dnn accelerator," in *Proceedings of the 39th International Conference on Computer-Aided Design*, pp. 1–9, 2020.

[7] J.-H. Yoon *et al.*, "29.1 a 40nm 64kb 56.67 tops/w read-disturb-tolerant compute-in-memory/digital rram macro with active-feedback-based read and in-situ write verification," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 404–406, IEEE, 2021.

[8] B. Crafton *et al.*, "Breaking barriers: Maximizing array utilization for compute in-memory fabrics," in *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SoC)*, IEEE, 2020.

[9] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[10] P.-Y. Chen *et al.*, "Neurosim+: An integrated device-to-algorithm framework for benchmarking synaptic devices and array architectures," in *2017 IEDM*, pp. 6–1, IEEE, 2017.