Accelerating Analog Compute-in-Memory Simulation with FPGAs

James Read
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA
jread6@gatech.edu

Vaidehi Garg
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA
vgarg34@gatech.edu

***Abstract*—Analog compute-in-memory (CIM) is a research field gaining interest for accelerating deep neural networks due to its promise of computing highly energy efficient vector-matrix multiplications in parallel. Due to the analog nature of this computation, the outputs are prone to errors caused by noise in the circuits. Thus it is important to have accurate simulation of such systems and their sources of noise to predict the impact on inference accuracy these chips would have before the arduous tapeout process. We aim to develop such a simulator and accelerate it with an FPGA to allow for full-scale simulations of large DNNs such as ResNet-50. For the scope of this project, we will implement this simulation for the first layer of ResNet-50 on an FPGA and compare it to baseline CPU and GPU implementations in Python.**

I. Introduction

Modern computing algorithms such as deep neural networks (DNNs) require large quantities of data and vector-matrix multiplications (VMMs). This necessitates frequent movement of data between the compute units and the memory, making it difficult to efficiently implement DNN models. In recent years, compute-in-memory has been proposed as a promising solution to this problem, since it allows for the acceleration of VMM operations by exploiting the crossbar nature of memory sub-arrays.

In analog CIM, the weights of the DNN model are typically encoded as the conductances of the memory cells, while the input vector is represented by the voltages applied to the rows. The multiplication is performed in parallel in the analog domain using Ohm's Law, i.e., each memory cell has an associated current as a result of the input voltage being multiplied with the cell conductance. Following Kirchhoff's Current Law, the current summation along the columns represents the output vector. An illustration of this is shown in Figure 1. Although analog CIM has the capability to provide huge improvements in power, performance, and area, it encounters analog non-idealities that impact the inference accuracy. These include factors such as IR drop, RC delay, and intrinsic noise amongst others.

A simulator for analog CIM systems and their associated non-idealities can be useful in predicting the impact of these non-idealities on inference accuracy prior to fabricating these designs in hardware. In this project, we aim to develop such a simulator and accelerate it with an FPGA to allow for full-scale simulations of large DNNs such as ResNet-50. For the scope of this project, we will implement this simulation for the first layer of ResNet-50 on an FPGA and compare it to a baseline CPU and GPU implementations.

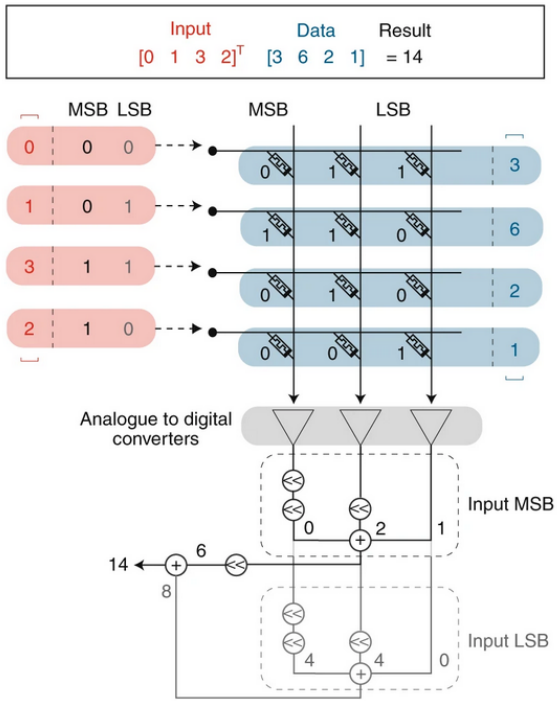

Fig. 1. Analog CIM illustration, originally from [4]

II. Problem Description

Simulating a compute-in-memory system that accelerates CNNs adds a significant amount of compute time to inference. This is because many additional steps must be added to the convolution operation including mapping the convolution filter to a tile of crossbars, quantizing inputs and weights into integers, solving the circuits, injecting noise, simulating the ADC operation, shifting and adding ADC outputs, and de-quantizing the final output. Because these operations are not natively supported with PyTorch functions, they cannot benefit fully from the parallel compute in GPUs with a Python

TABLE I
COMPARISON OF CPU AND GPU PERFORMANCE FOR CROSSSIM

Dataset	Neural Network	CPU runtime	GPU runtime
ImageNet	ResNet50	20.58 s/image	13.59 s/image
ImageNet	VGG-19	82.07 s/image	35.50 s/image
ImageNet	ResNet50	1.61 s/image	5.68 s/image
ImageNet	VGG-19	0.23 s/image	0.45 s/image

implementation. Because of this, we want to use an FPGA to accelerate each of these steps, and try to achieve a faster runtime than a PyTorch implementation.

The main limitation of compute-in-memory is the question of its impact on network accuracy. With a fast inference simulation tool, circuit designers can rapidly prototype their designs with a focus on accuracy, and CIM can overcome its most significant drawback at a faster pace.

III. RELATED WORK

The main two CIM simulation tools used are NeuroSim [1] and CrossSim [2]. NeuroSim is primarily a compilation tool that generates power, performance, and area estimates of a DNN accelerator if it is implemented using CIM. There is an inference simulation included that supports limited evaluation of non-idealities, however, it does not solve analog circuits, leading to several noise components being neglected. On the other hand, CrossSim does include circuit simulations, allowing for the inclusion of parasitic resistance on the accuracy. However their detailed simulations result in a slower inference time as shown in Table I.

IV. PROPOSED APPROACH

Implementing analog CIM using frameworks such as PyTorch involves the integration of multiple libraries and APIs. Although iterating on this software is relatively fast and straightforward, the runtime is not efficient for larger simulations, and significant portions of the codebase remain abstracted away from the end user, preventing fine-grained control and error injection into the circuit solvers.

We plan to implement lean circuit solvers using C for HLS. This will allow us to compute the CIM circuits more efficiently using the FPGA resources, while also achieving greater control and transparency into the computation itself. Implemented successfully, this would reduce the reliance on external circuit solvers and enable error injection from measured CIM devices and circuits at various levels of the simulation.

A. Simulation of Analog CIM

Because CIM computes vector-matrix multiplications, we need to use the image-to-column method to accelerate a convolution. While computing the convolution this way, the simulation will include the following steps:

1) **Unfolding Input Feature Map.** CIM computes vector-matrix multiplications, thus we need to unfold the input feature map from a multi-dimensional array into a series of vectors that can be applied to the memory arrays, i.e., a 2D array. Unfolding the input feature map of the first layer of ResNet-50 results in a 12544×147 matrix. We perform this step in Python, and exclude it from our FPGA implementation, since we intend to mainly focus on accelerating the computation. A visual of the unfolded is provided in Fig. 3
2) **Partitioning matrix multiplication amongst processing elements (PEs).** Since memory arrays are 2D in nature, the 4D convolution filter must be unrolled to a 2D matrix. See Fig. 3 for a visual. In the 2D matrix, each column corresponds to each channel in the filter. After unrolling the filter, the weight matrix is of size 147×520 for the first layer of ResNet-50. Additionally, because memory array sizes are fixed and large memory arrays have more analog noise, we must partition the matrix multiplication into a series of processing elements. A processing element is composed of an array of RRAM memory cells of fixed size. The total weight corresponding to the section of the matrix multiplication are applied to the array. Each PE computes a partial sum of the full matrix multiplication. After each PE has finished its computation, partial sums are added together digitally to compile the result of the entire VMM. For our project, we are going to be accelerating the first layer of ResNet-50 in HLS. Because of this, we choose our memory array size to fit within the buffer size limits in the FPGA and be perfect divisors of the total weight matrix size.
3) **Quantize inputs to PEs.** In the majority of CIM designs, the inputs are binary numbers since the memory cells are most commonly single level cells. Therefore, to compute MACs with higher input precision, one bit of the input is applied each cycle and the outputs are shifted and added to account for the significance of each input bit. To avoid excessive computation time, inputs are normally quantized to 8-bits or fewer. In our project we will use 8-bit inputs.
4) **Quantize weights in PEs.** The memory devices used in CIM designs can either represent one bit of the weight in the case of a single-level cell and more than one bit in the case of multi-level cells. Because emerging non-volatile memory devices have yet to be extremely reliable for multi-level cells, the majority of designs use single level cells. Because of this we will assume single level cells in our simulation. Again, to avoid excessive computation time the weights will be quantized to 8-bits in our simulation. Outputs of each PE must also be scaled properly to account for the significance of the weight bits. The quantization of both the inputs and weights are done externally using TensorRT, a DNN quantization framework developed by NVIDIA.
5) **Map weights to conductance states.** Once the weights are quantized, we map the bits to the conductance states

of the desired memory device. A "1" bit in the weight is represented by the high conductance state of the device and a "0" bit is represented by the low conductance state. This is because a higher current will flow through the device in the high conductance state compared to the low conductance state, contributing more to the accumulated output current. We will use the conductance values of the two states based on [3]. In reality these conductances have some variations in them which we would like to simulate, however due to time constraints we will leave the inclusion of this non-ideality to future work.

6) **Circuit simulation.** In this step we will compute the analog currents flowing through each device, and accumulate them along each column in the crossbar circuit. We will do this using ideal circuit laws, namely Ohm's law ($I_{cell} = V_{in}G_{cell}$) and Kirchoff's current law ($I_{sum} = \sum_{col} I_{cell}$). See Fig. 5 for a depiction of the ideal circuit. Most CIM designs use voltage-mode ADCs to sense the analog outputs, thus we must convert the output currents into voltages. This is commonly done with a voltage divider circuit. We use the following equation in our simulations:

$$V_{out} = \frac{V_{dd}}{1 + R_{divider} \times I_{sum}} \tag{1}$$

where V_{dd} is the supply voltage for which we use 1V in our simulations. Similarly to the conductances these output voltages have variations mostly due to thermal noise. To include these variations in conductances and voltages in our FPGA simulations we would need a way to randomly sample Gaussian distributions which is likely to be out of the scope for this project. Therefore, we will compute the outputs in the ideal scenario, but we will design the simulation such that it will be easy to include these variations in the future.

7) **ADC sensing.** After the analog outputs are calculated, they must be converted to digital outputs. This step will be simulated by placing the analog outputs into discrete bins, the number of which is determined by the ADC precision. This consists of a for loop that compares the output voltage to reference voltages, setting the digital output to the index of the reference voltage closest to the output voltage. In our project we use an ADC precision of 5 bits.

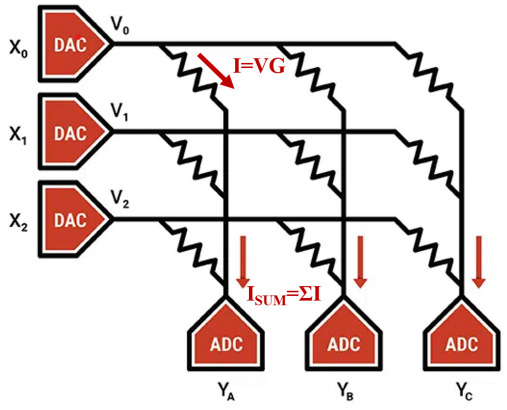

Fig. 2. Ideal CIM crossbar circuit.

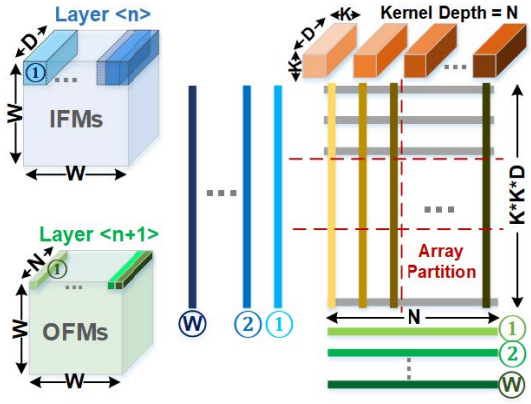

Fig. 3. Visual of input feature map unfolding and mapping filters to CIM arrays. Taken from [1].

8) **Shifting and adding ADC outputs.** After the analog-to-digital conversion, the digital outputs must be scaled depending on the significance of the input bit applied to the memory array as well as the significance of the weight bit of each column in the memory array. Once these scalars are applied, the result is accumulated to the partial sum.

9) **De-quantize the digital outputs.** After the shifting and adding, the integer outputs must be converted back to floating-point numbers for future operations in the DNNs. To accurately de-quantized the outputs two new scaling factors need to be calculated. One related to the scaling factor used for the input quantization and one related to the scaling factor used for the weight quantization. We can calculate these using an additional input vector and by adding an additional column of RRAM to the memory array. The increases the input matrix to 12545×147 and the weight matrix to size 147×520. The details for these calculations are excluded from this report but are explained in [1]. One drawback of adding the extra input row and columns to the weight matrix are that they become more difficult to partition evenly into tiles for computation on the FPGA. 12545 and 520 have only a few perfect divisors and thus limit the potential tile dimensions significantly.

V. Execution Plan

A timeline with milestones is provided in Table II. We began by writing a baseline analog CIM model in PyTorch and converting it to a C model using the LibTorch library. Next, we extended our work to an HLS implementation that was synthesizable for the Pynq-Z2 FPGA board, and further implemented tiling to meet the resource constraints. Finally, we worked on accelerating the HLS implementation while

staying within the resource constraints of the board, and implemented the baseline code on-board. Since we are a group of two people, both group members contributed to most tasks.

TABLE II
GROUP MEMBER CONTRIBUTIONS

Milestone	Group Member(s)	Date
Baseline PyTorch CIM Simulation	James	03/28
Baseline C CIM Simulation	James, Vaidehi	04/04
Baseline HLS for FPGA	James, Vaidehi	04/11
Baseline HLS with tiling	James, Vaidehi	04/18
Optimized HLS for FPGA	James, Vaidehi	04/25
On-board demo	Vaidehi	05/02
Final report	James, Vaidehi	05/02

VI. MILESTONES ACHIEVED

At the time of writing this report we have completed the baseline PyTorch CIM simulation for both CPU and GPU, the baseline C implementation using LibTorch, baseline HLS with tiling, an optimized HLS simulation, and the on-board demo of the baseline HLS version.

VII. CIRCUIT SIMULATION DETAILS

For our project, we chose to simulate a CIM circuit with RRAM memory devices as the memory cells. These devices are the most commonly used in CIM designs and have the most prior simulation work. We chose a memory array size of 21 rows by 104 columns. We picked this size because it evenly divides the total memory array allowing us to create our weight buffers with the same dimensions. Activating 21 rows in parallel allows the output of each column to range from 0-21, which means we need at least a 5-bit ADC to sense all the potential outputs. We chose an on-off ratio of 147, as it would allow for accurate sensing of all output states had we activated all 147 rows of the weight matrix in parallel. We technically do not need this high of a ratio to avoid sensing errors with only 21 rows in parallel but we left it this way as the on-off ratio won't have any negative effects because we aren't including the circuit non-idealities in this project. Finally, we pre-calculate the value of the resistive divider and reference voltages for the ADC externally from the PyTorch implementation.

VIII. C SIMULATION USING LIBTORCH

We first started with a direct translation of the PyTorch circuit simulation in C using LibTorch, which is an API that exposes the C++ backend used by PyTorch. For the C model, we replaced any PyTorch function calls in the circuit simulation section of the baseline with the equivalent LibTorch function calls. This allowed us a quick way to set up a structure for the C code, verify the functional correctness of our C implementation, and resolve any intermediate issues such as reading in files, passing arguments etc. We had originally intended to use the LibTorch implementation as our synthesizable baseline, but during the course of the project, we realized that LibTorch functions would not be synthesizable using HLS due to the size and complexity of the library. Additionally, using an external library such as LibTorch would add overhead and abstract away computationally intensive loops that may be good candidates for optimization. Therefore, we decided to re-implement the LibTorch functions within our code using simple C constructs such as static arrays and for loops.

IX. BASELINE HLS SIMULATION

We initially implemented the baseline without considering the utilization of the FPGA resources, and used a C simulation to guarantee functional correctness. Since we don't perform the input and weight matrix unrolling on the FGPA, we mainly needed to compute the circuit simulation step outlined in section IV. This step is summarized in algorithm 1.

Algorithm 1 Circuit Simulation

Input: $\mathbf{precision}, \mathbf{adc\_levels}, \mathbf{inputs}, \mathbf{weights}, \mathbf{R_{divider}}$
Output: $\mathbf{outputs}$

outputs $= 0$
for i in $input\_rows$ **do**
  for j in $output\_cols$ **do**
    for m in **precision do**
      $input\_mask = 1 << m$
      for k in **precision do**
        $weight\_mask = 1 << (\mathbf{precision} - k - 1)$
        for r in $weight\_rows$ **do**
          $w\_col = j * \mathbf{precision} + k$
          $input\_bit = input[i][r] \& input\_mask$
          $equiv\_cond$ += $input\_bit * weight[r][w\_col]$
        end for
        $v\_out = V_{dd}/1 + equiv\_cond * \mathbf{R_{divider}}$
        $adc\_out = \mathbf{adc\_levels}$
        for l in **adc_levels do**
          if $(v\_out >= v\_ref[l])$
          $adc\_out = l$
          break
        end for
        $output[i][j]$ += $adc\_out * weight\_mask * input\_mask$
      end for
    end for
  end for
end for
return $\mathbf{outputs}$

In the above algorithm, both 'inputs' and 'weights' are 2D matrices, and 'precision' is the bit-precision of the weights and inputs (we use 8b for both). In the HLS implementation the size of 'inputs' and 'weights' are determined by the buffer sizes. For our HLS baseline and the optimized HLS we use an

input tile of size 193×21 and a weight tile of size 21×104. To represent the weights and voltage values to a reasonable degree of precision, we used a fixed point format of 32 bits with 1 sign bit, 6 mantissa bits, and 25 fractional bits. The baseline HLS implementation latency and resource utilization reported from the Vitis HLS synthesis tool are shown in Table III.

TABLE III
BASELINE HLS SYNTHESIS REPORT

Latency (s)	BRAM	DSP	FF	LUT
11.142	31%	11%	11%	30%

X. OPTIMIZED HLS SIMULATION

Since the analog outputs of the columns are independent of each other, ideally we intended to calculate them in parallel. However, due to FPGA resource constraints, pipelining the main matrix multiplication loop over the input rows and weight columns was not possible. Therefore, we focused our efforts on achieving latency gains by parallelizing the inner loops within the main matrix multiplication loop.

First, we ensured that the two loops that compute the equivalent conductance of a set of weights and perform ADC sensing were fully unrolled. Since these are 'perfect' loops by HLS standards (i.e. have no dependencies outside of the loop) and iterate a relatively small number of times compared to the main computation loop, they were automatically unrolled by the synthesis tool. We also made sure that the voltage reference array, which contains 16 fixed-point values, was completely partitioned.

Next, we pipelined the weight precision loop, since the loops contained within it were already independent of each other and fully unrolled. Since this loop accesses the weight buffer in each iteration, we partitioned the buffer along both dimensions (completely partitioned in dimension 1, and cyclically partitioned with a factor of 8 in dimension 2).

Implemented together, these optimizations allowed us to achieve a latency of 584ms. The resource utilization was under 100% utilization across all resources except LUTs, which were at 119%. Since Dr. Hao advised that LUT usage is often over-reported by the HLS tool, we did not attempt to bring this number down further in favor of keeping the latency low and the fixed-point precision high. The optimized latency achieves almost a 20x speedup over the baseline HLS latency, but is comparable to the computation portion of the PyTorch implementation that uses the CPU. However, it is 20x slower than the PyTorch implementation on GPU. The results are reported in Table IV and a comparison of the resource utilization is shown in Figure 4. An overall comparison against CPU and GPU implementation is shown in Figure 5.

XI. ON-BOARD DEMONSTRATION

Although the HLS simulation worked with a 32-bit fixed point format of (26, 6), the on-board implementation required a higher precision to achieve functional correctness. We are currently not sure why this is the case. Therefore, for the on-board demo, we used the baseline code with a 64-bit fixed point format of (64, 11). The on-board implementation completes in 31.442s, whereas the HLS latency is 17.364s when running with the same fixed-point format (with under 100% resource utilization across all resources), indicating a 1.8x slowdown. A comparison of the results is shown in Table V, and an image of the on-board implementation (on a Pynq-Z2 board, using the Xilinx Vivado tool) is shown in Figure 6. Similar to the results for the labs previously completed in this course, there is a significant discrepancy

TABLE IV
BASELINE VS. OPTIMIZED HLS SYNTHESIS REPORT

	Latency (s)	BRAM	DSP	FF	LUT
Baseline	11.142	31%	11%	11%	30%
Optimized	0.584	9%	81%	61%	119%

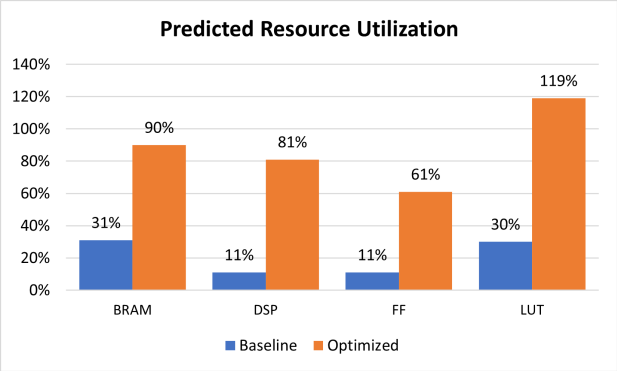

Fig. 4. A comparison of resource utilization between the baseline and optimized implementations.

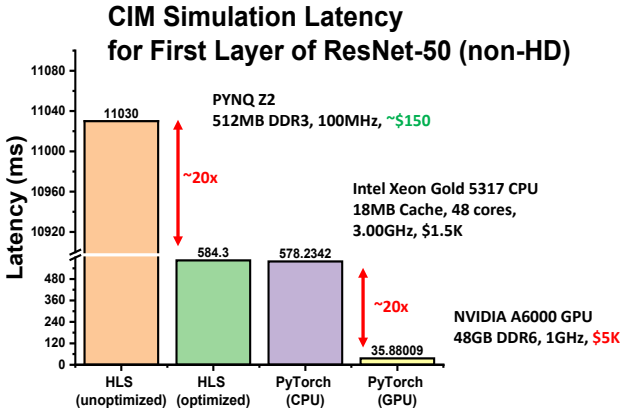

Fig. 5. Latency results from HLS synthesis. We achieved a 20x speedup compared to our HLS baseline which matched the PyTorch CPU version. There is still a $20\times$ difference between the optimized HLS and GPU version. However, the FPGA is $64\times$ cheaper than the GPU in terms of latency/price the FPGA wins.

between the HLS estimate for latency and the observed on-board latency. This is likely a result of Vitis HLS not taking into account physical delays due to wire delay and routing accurately, and underestimating the latency of AXI transfers from DRAM to local on-board buffers.

TABLE V
VITIS HLS VS. ON-BOARD LATENCY USING (64,11) FP FORMAT

HLS Latency (s)	On-board Latency	Comparison (Board/HLS)
17.364	31.442	1.81

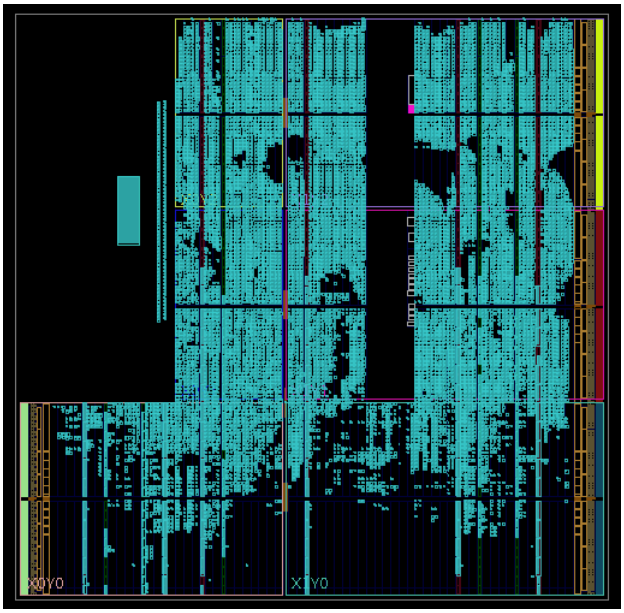

Fig. 6. On-board implementation.

XII. CONCLUSION

We were able to achieve significant speedup with our HLS solution and match the latency of a high-end CPU running the PyTorch implementation. Unfortunately the GPU still outperforms our FPGA implementation, albeit at a much higher price tag. We believe it is possible to further optimize our FPGA implementation and get closer to the GPU implementation but achieving a further 20x speedup may be out of reach. However after further profiling our code, it seems that the sections of the simulation that we didn't focus on in this project, namely the weight mapping and unfolding, take up a significant amount of time in the PyTorch simulations.

XIII. POTENTIAL FUTURE WORK

For the scope of this project, we focused mainly on implementing analog CIM circuit computation on the FPGA, and did not include circuit non-idealities. To implement this feature in the future, we would need to model each non-ideality (such as sampling Gaussian random variables), and inject them at requisite points of the computation. We could potentially collaborate with the team that included random sampling using FPGA in their project for this class. We also would like to implement the option for the user to define the number of rows computed in parallel, which may involve more complex tile handling. We would also like to debug the on-board demo further, and get it working with a lower fixed-point precision, so that we can implement our optimized design with lower latency and make better use of on-board resources. Finally, we would like to accelerate each layer of ResNet-50 to see the total simulation time for the entire network.

REFERENCES

[1] X. Peng, et al. "DNN+NeuroSim: An end-to-end benchmarking framework for compute-in-memory accelerators with versatile device technologies," IEDM, 2019. https://github.com/neurosim

[2] T. Xiao, et al. "On the Accuracy of Analog Neural Network Inference Accelerators," ArXiV, 2022.

[3] W. Li, et al. "A 40nm RRAM compute-in-memory macro featuring on-chip write-verify and offset-cancelling ADC references," ESSCIRC, 2021.

[4] Sebastian, A., Le Gallo, M., Khaddam-Aljameh, R. et al. Memory devices and applications for in-memory computing. Nat. Nanotechnol. 15, 529–544 (2020). https://doi.org/10.1038/s41565-020-0655-z