Flex-PIM: A Ferroelectric FET based Vector Matrix Multiplication Engine with Dynamical Bitwidth and Floating Point Precision

Yun Long, Edward Lee, Daehyun Kim, Saibal Mukhopadhyay
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, USA
yunlong, elee539, daehyun.kim, saibal.mukhopadhyay@gatech.edu

***Abstract*—This paper presents Flex-PIM, a ferroelectric FET (FeFET) based processing-in-memory (PIM) engine for vector-matrix-multiplication (VMM). With FeFET as the basic memory cell, Flex-PIM features low read latency/programming energy, non-volatility and high density. The core of Flex-PIM micro-architecture is an all-digital VMM engine integrated with innovative memory array peripherals to realize dynamically controllable bitwidth and floating point precision. The Flex-PIM architecture is simulated in 28nm CMOS technology and shows multiplication-accumulation (MAC) operations from 32-bit floating point (99 GMACS/W) to 4 bit fixed-point (3.3 TMACS/W). A system level design with specialized instruction set is presented to acclerate training and inference of deep neural networks (DNN) using Flex-PIM. The full-chip simulations show that Flex-PIM can increase computing efficiency of training and inference by 32x and 120x, respectively, over desktop GPUs while maintaining high accuracy over a wide-range of DNN models using flexible precision.**

***Index Terms*—Processing in memory, ReRAM, FeFET, DNN acceleration, digital VMM engine.**

I. Introduction

Processing-in-memory (PIM) architectures [1]–[7] have been extensively explored to accelerate vector-matrix-multiplications (VMM) to facilitate deep neural network (DNN) computation. The PIM design eliminates the separation between memory and computing, providing high computational efficiency (throughput/watt). Moreover, the PIM design can leverage emerging non-volatile memory devices, such as ReRAM [1], [4] or Ferroelectric FET (FeFET) [3], leading to non-volatile computing platforms.

However, the adoption of PIM-based VMM engine design faces many critical challenges. In prior works [1], [3], [4], the high throughput is achieved by using internally analog computation, leading to overheads of analog to digital conversions and making the computation error-prone. Consequently, most past PIM designs employ reduced bitwidth (such as 8-bit) fixed point data representation to manage data conversion cost. More recently, an all-digital PIM design is proposed by leveraging the bit-wise AND for multiplication and row-by-row read and accumulation operation [8]. While eliminating the analog computation, this design still limits to 8-bit precision and can only accelerate DNN inference. Though 8-bit precision provides good inference accuracy for DNN models on image classification tasks [9], we observe that more complex DNN applications like object detection [10] require higher precision (see section III-C). Moreover, DNN training typically requires floating point precision to ensure good accuracy since gradient calculation and back-propagation are very error-sensitive.

Recent developed general purpose processors (e.g. GPU) or digital accelerators [11] can support multiple bit-precision. More recently, a SRAM based bit-serial integer/floating point vector computing engine is proposed [2], but the bit-serial operation limits performance. A flexible and computationally efficient PIM engine supporting dynamical bitwidth and floating point precision is highly desired but still missing.

Towards this end, this paper presents Flex-PIM, a FeFET based all-digital PIM architecture with flexible bit-precision including 32-bit floating-point support. The core design of Flex-PIM is an all-digital VMM engine that leverages bit-wise AND and row-by-row accumulation operation. The key benefits of Flex-PIM comes from the novel peripheral circuits which realize dynamical bit-precision and floating point operation. With FeFET as the memory cell, we achieve lower read latency and programming energy over ReRAM while keeping the benefits of high density and non-volatility. We demonstrate the application of Flex-PIM for acclerating training and inference of DNN engines. This paper makes following key contributions:

- We present the micro-architecture of FeFET-based VMM engine with flexible bit-precision and floating point support. The Flex-PIM architecture is simulated in 28nm CMOS technology to show multiplication-accumulation (MAC) operations ranging from 32-bit floating point to 4 bit fixed-point with computing efficiency of 99 GMACS/W to 3.3 TMACS/W.
- We present a system design with associated instruction set to accelerate training and inference of various DNN models for image classification and object detection applications. The simulations show increased computing efficiency in training (32x) and inference (120x) of DNN

This work is supported by National Science Foundation (NSF) (181005). All authors are with School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332, USA.

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

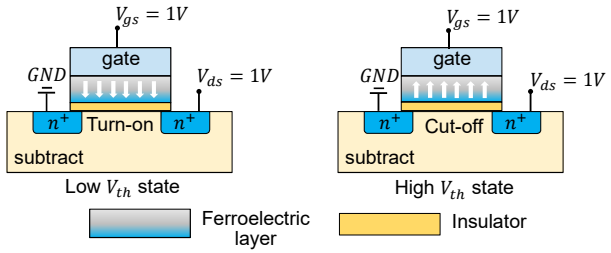


Fig. 1. FeFET structure with polarization downwards (i.e. low Vth state) and upwards (i.e. high Vth state).

over desktop GPU (Nvidia GTX1080Ti) while maintaining high accuracy using flexible precision.

II. Flex-PIM based VMM Engin

This section presents the micro-architecture and design of the Flex-PIM based vector-matrix-multiplication (VMM) engine.

A. All-digital FeFET based PIM design

1) Ferroelectric FET (FeFET): As shown in Figure 1, FeFET is a transistor type device with ferroelectric oxide sandwiched in the gate dielectric stack. With different ferroelectric polarization, FeFET exhibits switchable transistor threshold voltage, resulting to non-volatile memory. The low writing energy due to the unique field effect switching mechanism is the most prominent feature which distinguish FeFET from other emerging technologies. Further, FeFET eliminates the large RC delay in ReRAM case, therefore, reducing the read latency. Similar with prior works [2], [3], in this paper we use FeFET as binary memory device (i.e. 1-bit per cell).

2) Digital VMM engine: Figure 2(a) shows the configurations for analog/mix-signal based VMM engine design. Each memory cell stores 1-bit of the weight parameter. All wordlines are activated simultaneously and the current summed at BL results the multiplication-accumulation (MAC) operation. Figure 2(b) shows our digital VMM engine design, rather than activating all the wordlines simultaneously, the memory crossbar are accessed in a row-by-row (controlled by the one-hot-vector circuit) sensing and accumulation fashion. At each clock cycle, only one en_i signal is enabled. Then AND logic is performed between the enable signal (en_i) and the corresponding input value (a_i). Only when both en_i and a_i are high, the value (1-bit) stored in corresponding memory cells (b_i) are sensed out and accumulated in the counters. Essentially, For N-bit number MAC operation, it takes $N \times R$ cycles to perform the computation where N is the bitwidth and R is the number of rows in the memory array. One should note that while sacrificing the parallelism of analog computing, digital PIM configuration achieves similar performance over the analog design by eliminating the large delay introduce by the power/area hungry ADC [1]. Detailed comparison is discussed in section III-E.

We argue that the digital VMM engine design is orthogonal to the memory techniques (SRAM, ReRAM, and FeFET). In this work, we focus on FeFET and leverage its non-volatility, high density, low read latency/write energy to achieve the optimal computing efficiency.

B. Support for dynamical bit-precision

The flexible bit precision is achieved by a set of hierarchical organized *shifter & adder* units, as shown in Figure 3 [12]. Assuming weight parameters are 4-bit numbers and the weights are programmed to 4 adjacent cells (because the memory cell is binary), the first level accumulators (the smallest trapezoid in Figure 3) accept results from 4 BLs. The result can be directly sent to the output buffer or nearest router via the bypass connections (blue line in Figure 3).

Additional shifter & adder units are appended at the bottom for computation of high precision. For example, for 8-bit precision (parameters are 8-bits number), the results from two adjacent 4-bit shifter & adder are routed to the next level shifter & adder which can perform the shifting and adding operation over 8 bitlines. With more hierarchical shifter & adder units, our design supports fixed point MAC operation with different bit precision (up to 32-bit in our design) in the same place.

Compared with ASIC based designs which either reconfigure floating point unit (FPU) for fixed point multiplication or have separated computing resources for different type of operations, our PIM design can support flexible bit-precision in the same place seamlessly.

C. Support for floating point operation

Upon the support for dynamical fixed point computation, our PIM design also supports floating point operation which is compliant with IEEE 754 floating point standard. The single precision format of IEEE 754 has 32-bit data width with 1-bit for sign, 8-bit for exponent, and 23-bit for mantissa (Figure 4(a)). To clearly illustrate our approach, we use a simplified version of floating point representation as an example which has the first 2-bit as exponent and the rest 2-bit for mantissa (mantissa only has integer, no fraction bit). We also assume all the numbers are positive (there is no need for sign bit). As a example, 1011 (first 2-bits are exponent, equals to 2; last 2-bits are mantissa, equals to 3) in decimal representation is $3 << 2 = 12$ where $<<$ is the left shift operator.

The embedded table in Figure 4(a) shows the MAC operation between two vectors $\langle a_1, a_2, a_3 \rangle$ and $\langle b_1, b_2, b_3 \rangle$. The final result equals to 52. Figure 4(b) shows how to map this computation to our PIM design, the first step is to map the $\langle b_i \rangle$ vector to the memory crossbar. We check the maximum exponent of the three numbers in $\langle b_i \rangle$, which is 01. Then, values with maximum exponent (b_1, b_2) are mapped to the crossbar from the leftmost bitline (i.e. MSB); for value with smaller exponent (b_3), mapping is performed after right shifting. Similar with mapping vector $\langle b_i \rangle$ to the crossbar, we implement a input vector buffer to temporarily store vector $\langle a_i \rangle$. Following the same rules, for element with the maximum exponent (i.e. a_3), the mantissa is stored from the leftmost column of the input vector buffer. For element with smaller

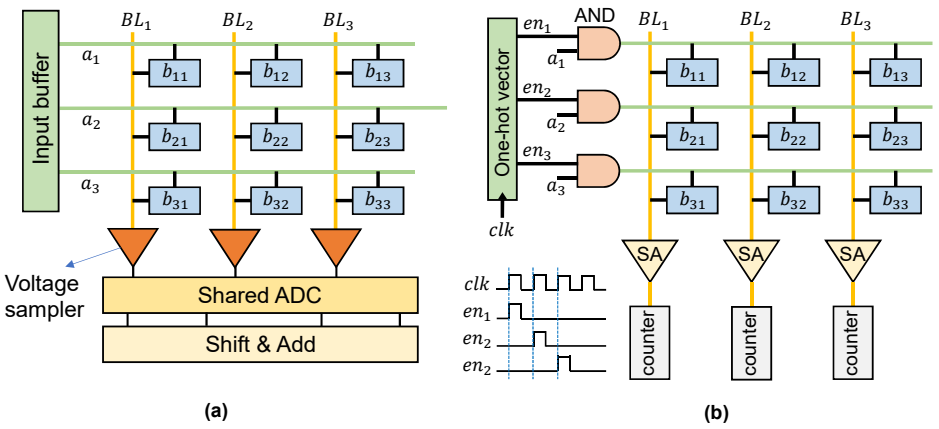


Fig. 2. Different PIM configurations to compute $a_1b_1 + a_2b_2 + a_3b_3$ where $\langle a_i \rangle$ are 1-bit numbers and $\langle b_i \rangle$ are 3-bit numbers. (a) Analog configuration. (b) Digital configuration.

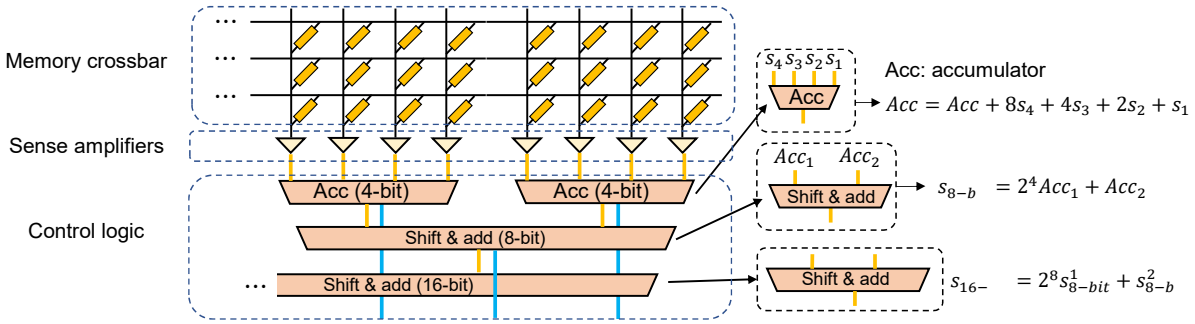
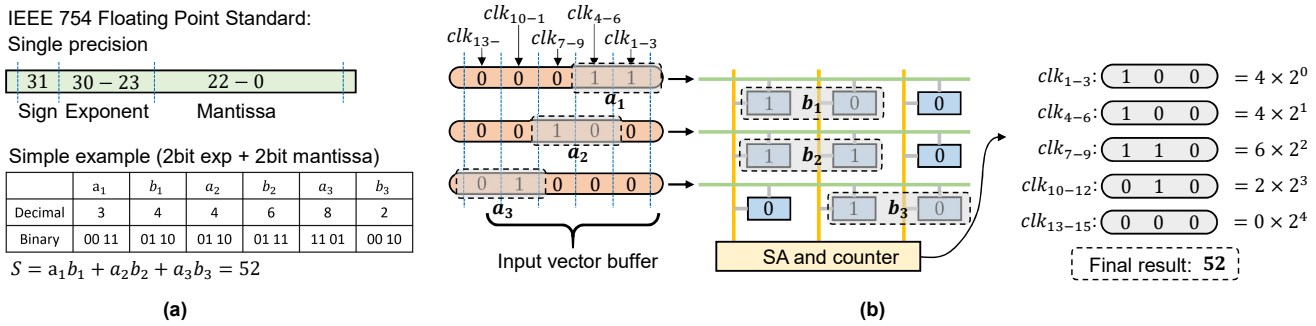


Fig. 3. Hierarchical *shifter & adder* used as multiplexer for dynamical bit precision [12].



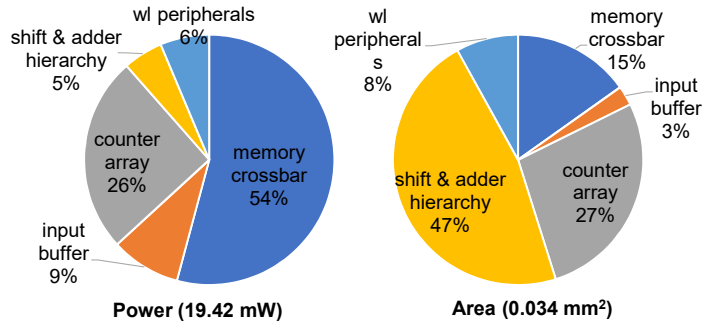
	a_1	b_1	a_2	b_2	a_3	b_3
Decimal	3	4	4	6	8	2
Binary	00 11	01 10	01 10	01 11	11 01	00 10

Fig. 4. (a) IEEE floating point standard and simplified version of floating point data representation. (b) Mapping and computing of floating point MAC inside memory.

exponent (i.e. a_1 and a_2), we perform right shift and append 0 to the left end. Figure 4(b) shows how we map $\langle a_i \rangle$ and $\langle b_i \rangle$ to the input vector buffer and memory array, respectively.

After mapping is done, computation process is similar as fixed point operations. At the first clock cycle, we fetch one value from the input vector buffer (start from the top-right corner) and activate the first WL accordingly. Since we only have three WLs, it takes 3 cycles to get the first part of the partial sum (Figure 4(b)). This process is repeated until all the values are fetched from the input vector buffer. After proper shifting, these partial results are summed together to get the final result, which is 52.

One big challenge is that for single precision floating point number, the range of exponent is -127 to 128, indicating that the length of the input vector buffer and number of columns in memory array to represent one number should be at least 255 bits/cells, to accommodate all possible values. This is apparently not practical and will significantly slow down the



VMM engine throughput and computing efficiency

	4-bit	8-bit	16-bit	32-bit	Floating-point
Throughput	64 GOPs	16 GOPs	4 GOPs	1 GOPs	1.9 GOPs
Density	1.88 TMACs/mm²	470 GMACs/mm²	117 GMACs/mm²	29 GMACs/mm²	56 GMACs/mm²
Efficiency	3.3 TMACs/W	824 GMACs/W	206 GMACs/W	52 GMACs/W	99 GMACs/W

VMM engine layout view:

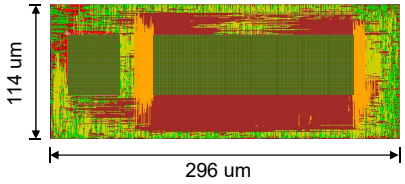


Fig. 5. Top: Distributed power and area for one VMM engine with a 256×256 FeFET crossbar. Middle: VMM engine throughput, computing density and computing efficiency with different bit-precision (TMACs/GMACs: Tera or Giga multiplication accumulation operation). Bottom: Layout view for a VMM engine (the area is $0.034mm^2$).

computing speed. In our implementation, we set the length of input vector buffer (also the number of cells to store one mantissa) to be 23-bit (same with the matissa's length in single precision floating point format). We argue it can provide the same precision as CMOS logic based floating point unit (FPU) design, where rounding errors also happen when the difference of the exponents of two operands are larger than 23-bit. For example, with single precision notation, $1.5\times2^{-1}+1.5\times2^{-32}$ still equals to 1.5×2^{-1} for a standard FPU.

D. Design and Simulations of VMM engine

The physical design of Flex-PIM based VMM engine is performed considering 28nm technology. For FeFET crossbar, WL driver and sense amplifier, we perform detailed SPICE simulation using the extracted netlist of the custom layout to estimate the energy/latency for data sensing. Then the SPICE simulation is coupled with synthesized digital blocks (such as the one-hot vector, counter array, *shifter & adder* hierarchy, etc) to form the VMM engine modeling. Synopsys Design Compiler and PrimeTime are used to model the power and area of the synthesized components.

We explored various crossbar size and aspect ratio for the design space optimization. Crossbar with 256 rows/columns produces the best overall performance in terms of efficiency and computing density. Top of Figure 5 shows the power and area distribution for a VMM engine design with a 256×256 FeFET crossbar. The VMM engine are divided into 5 blocks, namely, memory crossbar, WL peripherals (one-hot-vector unit), counter array for data accumulation, input buffer to temporary input vector storage, and shifter & adder hierarchy to realize flexible bit-precision.

The memory crossbar and counters occupy the majority of the power consumption (80%) because we use a higher internal memory clock (4GHz) to reduce the computation latency. On the other side, shifter & adder hierarchy takes 47% area but only consumes 5% power. This is because we only use it when the row-by-row data sensing is finished and the data in the counter are ready. Most of the time, the shifter & adder unit are idle. The insert table in Figure 5 illustrates the VMM engine throughput, computing density and computing efficiency under different bit-precision. The layout view of one VMM engine is shown at the bottom of Figure 5.

E. Design Overhead

Enabling the dynamical bit-precision and floating point computing is expected to introduce additional hardware design cost and increase computation latency. The overhead mainly comes from two aspect: *additional clock cycles for data shifting/adding* and *mantissa shifting for floating point operation*.

For the first design overhead, assuming we perform 16-bit fixed point operation using a 256×256 memory array, it takes $256 \times 16 = 4096$ cycles to accumulate data (i.e. the 256 rows are iterated 16 times). Then, another 3 clock cycles are required for the data going through the shifter & adder hierarchy.

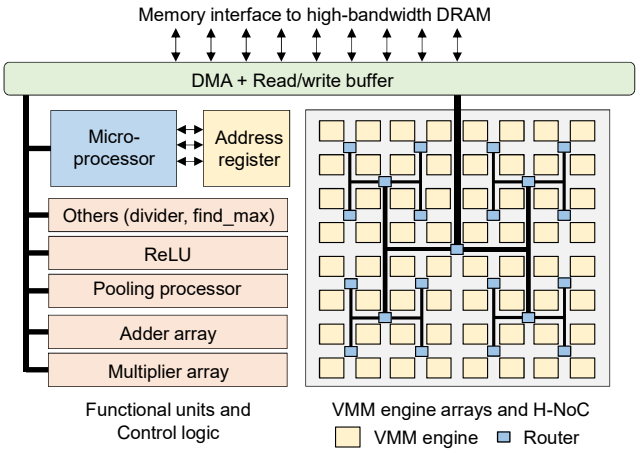


Fig. 6. System Architecture. There are 2048 VMM engines in total (16 MB) with each VMM engine contains one 256×256 FeFET crossbar.

In another word, the additional cost to realize dynamical bit-precision is negligible. On the other side, for floating point operation, mantissa must be shifted based on its exponent value accordingly. We argue that bit shifting is a much cheaper operation than multiplication and can be handled externally. Therefore, the hardware cost is also insignificant. Overall, the cost for dynamical bit-precision and floating point are trivial in terms of latency ($< 1\%$) and chip power (5%). We also synthesize a baseline digital VMM engine which only support 8-bit fixed point computation by elinimating the shifter & adder units, resulting to 896 GMACs/W efficiency. Meanwhile, Flex-PIM demonstrates 824 GMACs/W at 8-bit precision, less than 8% overhead in terms of computing efficiency.

On the other side, our design introduces large area overhead (47%) due to the fact that there are significant amount of registers for intermediate data storage inside shifter & adder.

III. Flex-PIM application: DNN Acceleration

This section discusses one application of Flex-PIM: The acceleration of training and inference of various DNN models.

A. System Architecture

The system design largely follows existing PIM based DNN accelerator architecture [1], [3] with several modifications to accommodate our VMM engine design and instruction set decoding procedure. As shown in Figure 6, VMM engines are placed in a 2-D plate, interconnected with a hierarchical network-on-chip (H-NoC). An accumulator is implemented inside the router to realize on-the-fly partial results summation (in the case that single matrix operation is mapped to multiple VMM engines). Experiments indicate that such NoC design can significantly improve the data transmission efficiency (both input data dispatching and result collection) [3]. Additional to the original design, in this work we design the H-NoC to support flexible bit-precisions and our implementation is fully parameterized. To be more specific, with different parameters, our H-NoC can be reconfigured to achieve different levels of hierarchy and data bandwidth to fit various application scenarios. In our implementation, we end up with a 7 level NoC hierarchy, containing 2048 VMM engines with the total memory capacity 16MB. In addition to the VMM engines array, several functional units are implemented to support the computation which can't be accelerated inside memory, such as Pooling, ReLU, random number generator, to name a few.

TABLE I
Flex-PIM instruction set.

Instruction type		Operands	Execution flow
Control		Precision, Train/inference mode, Batch size, registers	Reset control variables and regs
Layers	Conv layer	Output feature map, Input feature map, Convolution kernels	Load data ↓ dispatch data ↓ computing ↓ collect result ↓ store data
	FC layer	Output vector, Input vector, Weight matrix	
	Pooling layer	Output matrix, Input matrix	
	BN layer	Output matrix, Input matrix	
	Activation	Output vector, Input vector	
Parameter specification		Input size, Kernel size, Output size	Determine load/restore pattern

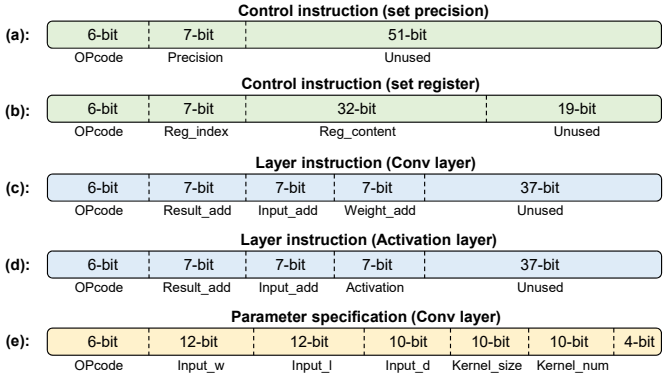


Fig. 7. Examples for Flex-PIM instruction set.

B. Instruction set and programming model

A key step to enhance the system flexibility is to design a dedicated domain-specific instruction set. Our instruction set is designed based on high-level (i.e. DNN layers) abstraction, making the code more readable and informative.

As shown in Table I, there are three types of instructions, control, layers, and parameter specification. Instructions are 64-bit with the first 6-bit as Opcode. The first 2-bit of Opcode specify the instruction types (00 for control, 01/10 for layers and 11 for parameter specification). Control instruction is used to define computing precision, set running mode (inference or training) and write address register. Examples for control instruction are shown in Figure 7(a, b). The second type of instruction defines the layer and where the weight/activation

should be fetched from. For layers containing weight parameters (e.g. Conv layer in Figure 7(c)), there are three operands (each has 7-bit, corresponding to the 128 address registers) for computing result address $Result_{add}$, activation address $Input_{add}$, and weights address $Weight_{add}$. For layers without weights (e.g. activation layer in Figure 7(d)), the third operand is to indicate the computing type.

However, we are still facing an issue that 64-bit instructions can not include all the information for a layer definition, such as input feature map depth, convolution kernel size, etc. This leads to the implementation of the parameter specification instruction. During execution, once an instruction is decoded and identified as layer-type instruction, the processor immediately fetches another instruction (i.e. a parameter specification instruction) from the instruction buffer. After getting all the necessary knowledge, the processor then asks the DMAC (direct memory access controller) to fetch data from the right location and perform data partition and dispatching. Figure 7(e) shows an example of a parameter specification instruction which provides definitions for a Conv layer.

A software/hardware interface is designed to bridge the gap between software and hardware, letting users easily deploy their applications without specific hardware knowledge. With the well-defined instruction set, the implementation of the software/hardware interface becomes straightforward. The runtime system takes DNN definition file (and pre-trained model if available) as input, sets the computing model and running precision, and performs layer-wise interpretation to translate the high-level python script to the developed instructions.

C. Benefits of flexible bit-precision

The motivation for flexible bit-precision design is to explore the optimal tradeoff between accuracy and computing efficiency. To be more specific, we can use low precision (e.g. 8-bit) for DNN models which are robust towards quantization while high precision (e.g. 16-bit or floating point) for models which are sensitive. Even more, we can have different precision inside a single network. For example, our experiment with Faster-RCNN (an object detection network) [10] shows that the accuracy (mAP) drops 30% for uniform 8-bit quantization even after re-training. The solution to compensate the accuracy drop is called adaptive/dynamical quantization where only part of the DNN model is quantized. As shown in Figure 8, we reduce the bitwidth of the backbone feature extractor (ResNet-101) to 8-bit and keep the rest intact (the first and last layer of ResNet-101 are not quantized). The accuracy is largely recovered. Since the backbone occupies the majority (more than 90%) of the parameter size and computation, we still achieve significant gain from the dynamical quantization.

D. Performance of benchmark DNN models

Our benchmark includes three image classification DNN models (AlexNet [13], VGG [14], and ResNet-50 [15]) using ImangeNet dataset [16] and one object detection model (Faster-RCNN with ResNet-101 as the feature extractor) using MS-COCO dataset [17].

TABLE II
COMPARISON WITH OTHER IN/NEAR MEMORY DESIGNS.

	Memory solution	Precision	Configs	Computing efficiency (GOPs/W)
SRAM PIM [2]	SRAM	Fixed point Floating point	Digital	560 (8-bit) N/A (floating point)
ReRAM PIM [1]	ReRAM	Fixed point	Analog	381 (16-bit)
FeFET PIM [3]	FeFET	Fixed point	Mixed-signal	443 (8-bit)
Flex-PIM	FeFET	Fixed point Floating point	Digital	824 (8-bit) 206 (16-bit) 99 (floating point)

During inference phase, we use 8-bit for both the activation and weight parameters for the image classification DNN models (AlexNet, VGG, and ResNet-50). For Faster-RCNN, we apply the dynamical quantization methodology where only the backbone network is quantized to 8-bit. For training, 32-bit floating point is applied for all the benchmark models.

Figure 9 shows the speed comparison of desktop GPU (Nvidia GTX1080Ti) and Flex-PIM for training and inference across the benchmark DNN models under varying batch sizes. For complex DNN models (such as VGG-16, ResNet-50, and Faster-RCNN), we choose smaller batch sizes to avoid the *run out of GPU memory* error. The throughput (i.e. images per second) for each DNN models are normalize towards the GPU throughput with minimum batch size for better visualization. We observe that Flex-PIM outperforms GPU solution by 6.1x and 23.2x in terms of training and inference speed, respectively. Additionally, desktop GPU's power is 5.2x higher than Flex-PIM (47W for 2048 VMM engines and corresponding function units), resulting up to 32x - 120x computing efficiency (GMACs/W) improvement.

E. Comparison with prior works

Compared with prior mixed-signal FeFET based PIM architecture (443 GMACs/W with 8-bit precision) [3], we achieve 2x performance improvement benefiting from the all-digital VMM engine design and the high internal memory clock frequency. Also, our design is more flexible with the support for dynamical bitwidth and floating point precision. On the other hand, ISAAC [1], a ReRAM based analog PIM architecture supports 16-bit fixed point computing, illustrates 384 GMACs/W efficiency, which is 1.8x higher than our approach. However, it ignores the large RC delay of ReRAM sensing and is error-prone due to the analog computing. We also compare with recent SRAM based in/near memory design [2], which gives 560 GMACs/W performance for 8-bit integer multiplication. Our design outperforms it by 50% improvements. The key results and design configurations for these PIM architectures are summarized in Table II.

As a conclusion, Flex-PIM demonstrates state-of-the-art performance while achieves both accuracy and flexibility.

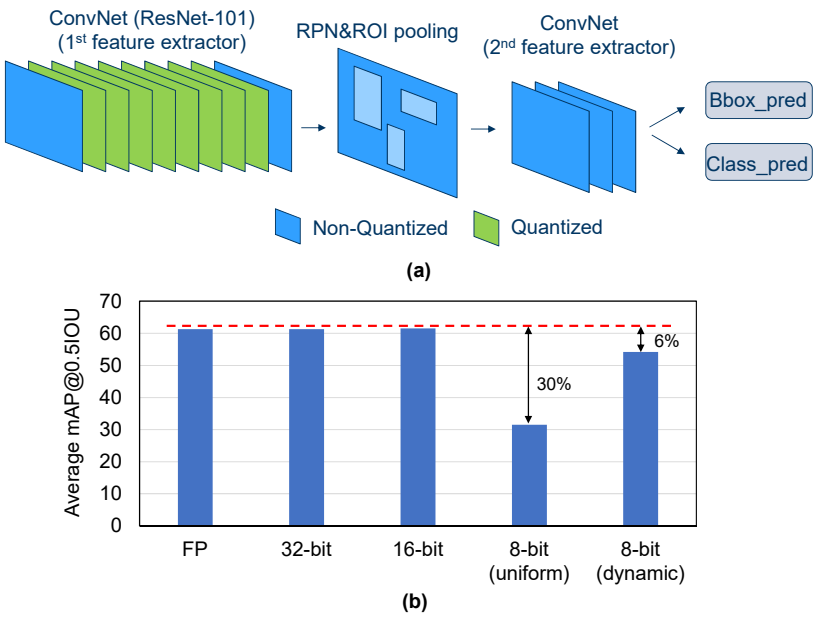


Fig. 8. (a) Adaptive/dynamical quantization for Faster-RCNN, only the middle layers of the ResNet-101 (first stage feature extractor) are quantized to 8-bit. (b) Accuracy (mAP @ 0.5 IOU) under different bit-precision and dynamical quantization.

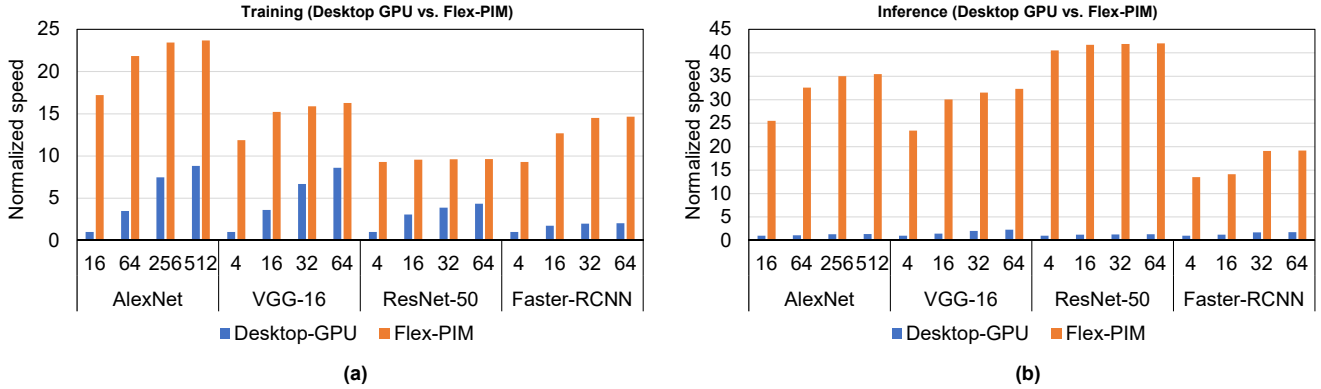


Fig. 9. Normalized training/inference speed of desktop GPU (Nvidia GTX1080Ti) and Flex-PIM for benchmark DNN models with varying batch size. The batch sizes for AlexNet are 16, 64, 256, 512; for other DNNs, the batch sizes are 4, 16, 32, 64. For inference, 8-bit is used for AlexNet, VGG, and ResNet-50; dynamical precision (8-bit + floating point) is used for Faster-RCNN. For training, all benchmark DNNs use floating point precision.

IV. Conclusion

In this work, we propose FeFET based PIM architecture to accelerate both DNN inference and training. With FeFET as the basic memory cell, all-digital VMM engine configuration, and dedicated circuit design to support dynamical bit-precision and floating point operation, we realize a highly efficient, flexible and accurate PIM design. We perform detailed power/area analyses and the design overhead is carefully evaluated. As FeFET continues to mature towards a commercial technology, we show the pathway to a fully-fledged flexible in-memory DNN accelerator solution.

References

[1] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 380–392, IEEE, 2016.

[2] J. Wang *et al.*, "A compute sram with bit-serial integer/floating-point operations for programmable in-memory vector acceleration," in *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, pp. 1440–1448, 2019.

[3] Y. Long *et al.*, "A ferroelectric fet based power-efficient architecture for data-intensive computing," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, p. 32, ACM, 2018.

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

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

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

pp. 288–301, ACM, 2017.
[7] C. Eckert *et al.*, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," *arXiv preprint arXiv:1805.03718*, 2018.
[8] Y. Long *et al.*, "A ferroelectric fet based processing-in-memory architecture for dnn acceleration," *IEEE Transactions on Exploratory Solid-State Computational Devices and Circuits*, no. 99, pp. 1–14, 2019.
[9] E. Wang *et al.*, "Deep neural network approximation for custom hardware: Where we've been, where we're going," *arXiv preprint arXiv:1901.06955*, 2019.
[10] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision (ICCV)*, pp. 1440–1448, 2015.
[11] H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775, IEEE, 2018.
[12] Y. Long, E. Lee, D. Kim, and S. Mukhopadhyay, "Q-pim: A genetic algorithm based flexible dnn quantization method and application to processing-in-memory platform," in *Proceedings of the 57th Annual Design Automation Conference*, pp. 1–6, 2020.
[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.
[14] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[15] K. He *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.
[16] J. Deng *et al.*, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255, Ieee, 2009.
[17] T.-Y. Lin *et al.*, "Microsoft coco: Common objects in context," in *European conference on computer vision*, pp. 740–755, Springer, 2014.