LAcc: Exploiting Lookup Table-based Fast and Accurate Vector Multiplication in DRAM-based CNN Accelerator

Quan Deng
College of Computer
National University of Defense Technology
dengquan12@nudt.edu.cn

Youtao Zhang
Computer Science Department
University of Pittsburgh
zhangyt@cs.pitt.edu

Minxuan Zhang
College of Computer
National University of Defense Technology
mxzhang@nudt.edu.cn

Jun Yang
Electrical and Computer Engineering Department
University of Pittsburgh
juy9@pitt.edu

ABSTRACT

PIM (Processing-in-memory)-based CNN (Convolutional neural network) accelerators leverage the characteristics of basic memory cells to enable simple logic and arithmetic operations so that the bandwidth constraint can be effectively alleviated. However, it remains a major challenge to support multiplication operations efficiently on PIM accelerators, in particular, DRAM-based PIM accelerators. This has prevented PIM-based accelerators from being immediately adopted for accurate CNN inference.

In this paper, we propose LAcc, a DRAM-based PIM accelerator to support LUT- (lookup table) based fast and accurate multiplication. By enabling LUT based vector multiplication in DRAM, LAcc effectively decreases LUT size and improve its reuse. LAcc further adopts a hybrid mapping of weights and inputs to improve the hardware utilization rate. LAcc achieves 95 FPS at 5.3 W for Alexnet and 6.3 × efficiency improvement over the state-of-the-art.

1 INTRODUCTION

The proliferation of Convolutional Neural Networks (CNNs) has motivated the development of novel ASIC CNN accelerators to improve their inference as well as training performance. Most such accelerators focus on achieving a good tradeoff between improving computation efficiency and alleviating memory constraints. When exploiting mature integrated circuit designs to construct high performance computing blocks, CNN accelerators often face tight memory bandwidth and capacity constraints. For example, Dadiannao [4] chooses eDRAM instead of SRAM as the on-chip memory to improve the on-chip memory capacity, which helps to store weights and activation parameters and decrease the off-chip data movement. EIE [11] leverages the sparsity of neural networks to compress CNN data, which improves the effective off-chip memory bandwidth.

Alternatively, memory-centric designs, e.g., Neural Cache [9], Drisa [16], DrAcc[7], Neurocube [14] and PRIME [5], prioritize the memory subsystem to construct processing in memory (PIM) or processing near memory (PNM) designs. By avoiding massive data movement via integrating the processing unit and the memory together, PIM designs strive to achieve a balance between computation efficiency and memory performance.

However, it is challenging to support multiplication operation in PIM designs. Most memory-centric designs choose quantized neural networks that use binary or ternary weights, which eliminate most multiplication operations. Neural Cache[9] and SCOPE [17] are two PIM designs that support multiplication operation. The former leverages natural accumulation to implement multiplication while the latter adopts stochastic computing to implement approximate multiplication. Lacking high performance multiplication support has prevented PIM-based accelerators from being immediately adopted for accurate CNN inference.

In this paper, we propose LAcc, a DRAM-based PIM accelerator that supports LUT (lookup table) based fast and accurate vector multiplication for CNNs. Our contributions are as follows:

- We propose an LUT-based vector multiplication approach. LAcc leverages decomposed multiplication to decrease the LUT size and makes a tradeoff between LUT reuse and pre-calculation. LAcc studies the hardware utilization and the page parallelism when adopting different addends and batch sizes, and proposes a hybrid mapping of weights and inputs.
- LAcc implements the LUT-based vector multiplication in DRAM. It gives the corresponding data and hardware mapping methods and proposes an optimized XOR operation to further accelerate addition operations in DRAM.
- We simulate our proposed design and compare it with other PIM-based accelerator designs. Our experimental results show that LAcc improves the multiplication performance 6.8× over the ideal baseline, and achieves 6.3× efficiency improvement over the state-of-the-art without accuracy loss.

2 MULTIPLICATION IN CNNS

Multiplication is an important operation in CNNs. For example, in the inference stage of Alexnet, the MAC (Multiplication-and-accumulation) operations account for 85%-90% of the total operations [8]. While it is viable to adopt quantized CNNs to eliminate multiplication for high efficiency in mobile and IoT devices, there

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317845

is a non-negligible inference error, e.g., 7% error of ImageNet top-1 inference, between binary neural network [6] and the best case [24]. Considering the increasing complexity of future tasks, e.g., object detection and generative adversarial network (GAN), multiplication is necessary for accurate CNN inference.

Unfortunately, while it is highly efficient to support bitwise logic and arithmetic operations inside memory devices, it is challenging to support multiplication with PIM designs. We next present two alternative multiplication implementations.

2.1 Multiplication Decomposition

It is well known that a multiplication operation can be decomposed into a chain of addition operations. In recent studies, Stripes[13] and Bit-pragmatic[1] leverage the decomposition to mitigate the bit-level operation sparsity. Bit fusion[22] decompose a 16-bit multiplication into multiple 2-bit multiplications to achieve a flexible accelerator, concerning the varying operand length in different layers of CNN.

Assume there are four unsigned operands A=‘01’, B=‘11’, C=‘11’ and D=‘01’. X_n refers to the n_{th} bit of X. To calculate Y=A*B+C*D, we may perform two multiplications and then the sum, as shown in Equation 1. However, we may also decompose a 2-bit multiplication as shown in Equations 2 and 3, and then sum them up as shown in Equation 4. In the following discussion, the bit of the decomposed operand is referred to as **select bit (SB)** while the original operand is referred to as **addend**.

$$Y = A * B + C * D; \tag{1}$$

$$A * B = A * B_0 * 2^0 + A * B_1 * 2^1; \tag{2}$$

$$C * D = C * D_0 * 2^0 + C * D_1 * 2^1; \tag{3}$$

$$Y = (A * B_0 + C * D_0) * 2^0 + (A * B_1 + C * D_1) * 2^1; \tag{4}$$

Given a CNN convolutional layer that has (1) an H*W*N input matrix (H, W, N are the height, the width, and the depth of the matrix); (2) an R*C*M output matrix (M, R, C being the height, the width, and the depth of the matrix); and (3) a set of M kernels with one kernel being an K*K*N weight matrix (N, K are the depth and the height/width of the weight matrix). Assume L is the bit length of operands and S is the stride, we may transform the convolution using multiplication decomposition, as shown in Figure 1.

Comparing to the original CNN algorithm, the one in Figure 1 has an extra *L* loop to accumulate the results of different bit offset, and an additional accumulation in the *M* loop. The computation in the box refers to the equation of the *weight addend*, and the outside one refers to the equation of the *input addend.*

```
For(r=0;r<R;r++)
 For(c=0;c<C;c++)
  For(m=0;m<M;m++)
   O[m][r][c]= $\Sigma_{i=0}^{L-1}$ O[m][r][c][i]*$2^i$
   For(n=0;n<N;n++)
    For(l=0;l<L;l++)
      O[m][r][c][n][l]=$\Sigma_{i=0}^{k-1}\Sigma_{j=0}^{k-1}$W[m][n][i][j]* I[n][r*S+i][c*S+j][l]
      $\boxed{\text{O[m][r][c][n][l]=}\Sigma_{i=0}^{k-1}\Sigma_{j=0}^{k-1}\text{I[n][r*S+i][c*S+j]* W[m][n][i][j][l]}}$
      O[m][r][c][l]+= O[m][r][c][n][l]
```

Figure 1: The Addition-based Algorithm in LAcc.

2.2 LUT-Based Multiplication

Another multiplication implementation method is to use LUTs (lookup tables). Table 1 shows an ideal comparison among different implementations of addition and multiplication. All the algorithms are implemented in one platform for fair comparison and the results are normalized by the latency of one memory WT/RD. The required input operands are ready in the function units. A multiplication operation in Neural Cache needs 16 addition, 15 data copy and 1 data restore operations. A multiplication operation in SCOPE needs one AND operation. Implementing a multiplication operation using LUT costs only one memory WT/RD and achieves the best performance. However, it is challenging to implement LUT-based multiplication inside DRAM.

- **The setup time and the memory overhead of the lookup table are costly.** Ideally, LUT needs to contain all the result candidates of the multiplication. However, the space of LUT increases rapidly with the data length. For example, a 16-bit fixed point multiplication needs to cover a result space of 2^{32} possibilities. The storage overhead of LUT makes it non-preferable in DRAM. An 8Gb DRAM can hold only 2 such LUTs. Furthermore, the timing overhead of pre-calculation is unaffordable, concerning the limited computation resources in DRAM.
- **The hint variety of LUT decreases the hardware parallelism and utilization rate.** The process of accumulation-based multiplications and additions are hardware lockstep. Threads with the same operations can be perfectly distributed on the same DRAM page. However, it is hard for LUT-based multiplications to fill up the DRAM page. Because the input operands decide the entry of LUT. Only threads with identical inputs can be allocated on the same DRAM page, which brings unavoidable waste operations and induces a poor page utilization and system performance loss.

Table 1: Different Addition Implementations Comparison

	Addition		Multiplication	
Scheme	Delay	Accuracy	Delay	Accuracy
Neural Cache-like Case	13	100%	224	100%
SCOPE-like Case	13	100%	4	99%
LUT-based Case	13	100%	1	100%

3 LACC OPTIMIZATION ON VECTOR MULTIPLICATION

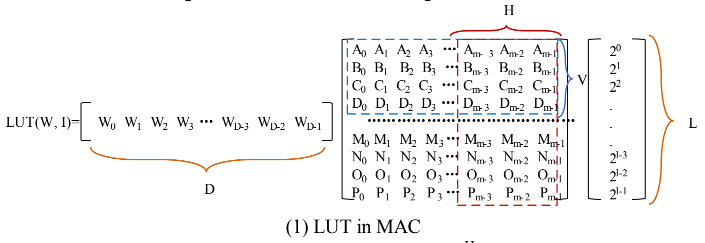
Most multiplication operations in CNNs are vector dot product, i.e., given two vectors $\vec{X}$ and $\vec{Y}$, we need to compute their dot product as shown in Equation 5. We assume each vector had D items and each item is of L bits. A 16-bit scalar multiplication is considered as a special case of vector multiplication. We next elaborate our optimization on vector multiplication.

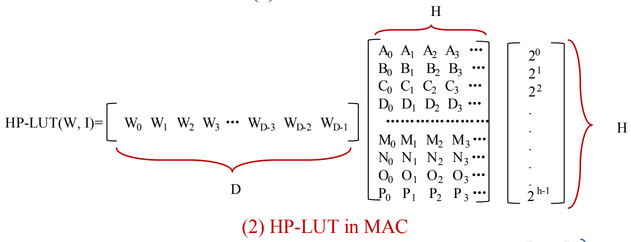
$$P(\vec{X}, \vec{Y}) = [X_1, X_2, ..., X_D] \cdot [Y_1, Y_2, ..., Y_D]^T = \sum_{i=1}^{D} X_i Y_i \tag{5}$$

3.1 Reducing LUT Size with Decomposition

Given the vector multiplication in Equation 5, the LUT table size can be greatly reduced if we fix some bits in the vectors. As a special case, if we fix $\vec{X}$, we only need to store 2^{D*L} results, a great reduction from the original size. Figure 2 illustrates how to reduce LUT table size through decomposition. We represent the second operand at the bit level where H and V are the vertical and horizontal partition units, respectively.

There are two decomposition methods. A *horizontal partition* (HP) decomposes the second operand to $\lceil L/H \rceil$ segments horizontally. The first vector is left untouched. $P(\vec{X}, \vec{Y})$ can be computed from the sum of $\lceil L/H \rceil$ sub-vector dot products (with shifts). Each sub-vector dot product can be looked up from an HP-LUT table that stores $2^{D*L} \times 2^{D*H}$=$2^{D*(L+H)}$ results. Figure 2 (2) presents the first sub-vector product after horizontal partition.





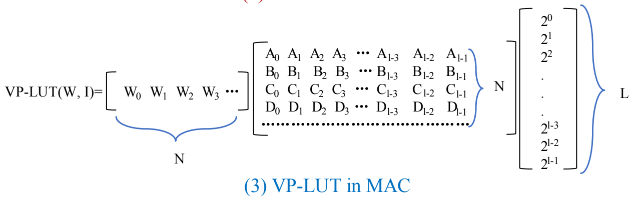


Figure 2: Two Basic MAC Partition Methods.

A *vertical partition* (VP) decomposes the second operand to $\lceil D/V \rceil$ segments vertically. The first operand is also partitioned to $\lceil D/V \rceil$ sub-vectors. $P(\vec{X}, \vec{Y})$ can be computed from the sum of $\lceil D/V \rceil$ sub-vector dot products. The result of each sub-vector dot product can be looked up from a VP-LUT table that stores $2^{V*L} \times 2^{V*L}$=2^{2*V*L} results. Figure 2 (3) presents the first sub-vector product after vertical partition.

A tradeoff between reuse and pre-calculation. In this paper, we are to fix the first vector $\vec{X}$ in the vector multiplication, we can further reduce the LUT table sizes to 2^{D*H} and 2^{V*L} entries for HP and VP, respectively. The tradeoff is, we need to pre-calculate the LUT table when the first vector changes. This overhead is amortized if the table can be reused multiple times. Clearly, the more the table is reused, the lower the amortized overhead is.

For HP, $\vec{X}$ needs to multiply with $\lceil L/H \rceil$ sub-vectors of $\vec{Y}$. The LUT table, after the pre-calculation for the first sub-vector multiplication, shall be used $\lceil L/H \rceil$ times. As a comparison, for VP, each sub-vector of $\vec{X}$ needs to one sub-vector of $\vec{Y}$. The LUT table is used only once after pre-calculation.

3.2 Dynamically Mapping the Weights or Inputs for Performance Optimization

Since a vector multiplication can change the order of two operands, i.e., $P(\vec{X}, \vec{Y}) = P(\vec{Y}, \vec{X})$, we may fix either vector to reduce the LUT table size. For the CNN discussed in the paper, one vector is the vector representation of a subset of items from the input matrix while the other vector is the vector representation of the weight matrix. The lengths of both vectors are K*K*N. We next reuse the notations in Section 2 and compare two data mapping approaches to find the best tradeoff between reuse and page utilization of LUT for PIM-based CNN.

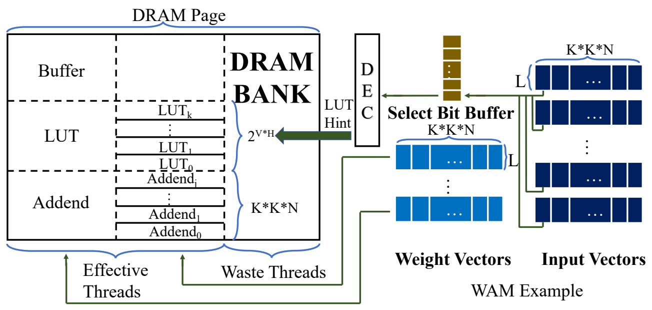


Figure 3: An Weight Addend Mapping(WAM) Example.

The first mapping approach is to fix the weight vector while decomposing the input vector, referred to as Weight Addend Mapping (WAM). Figure 3 illustrates how the weights and LUTs are mapped in the DRAM cell arrays. For each of the M weight vectors, we calculate a different LUT table. In the figure, the different entries of one weight vector are placed in different memory rows, which simplifies the pre-calculation of the LUT tables with in-memory operations. We place the aligned items of M weight vectors (or LUT entries) in one memory row to improve the page utilization during lookup. Considering the regular operations of LUT pre-calculation, multiple sets of weight vectors are mapped into one page to improve the page utilization during LUT pre-calculation under the premise of the page utilization of LUT. Furthermore, VP partitions the original LUT into sub-LUTs, which helps to fill up the page to improve the page utilization during LUT pre-calculation.

After adopting both VP and HP with VP size V and HP size H, each input vector is partitioned to K*K*N*L/(V*H) sub-vectors. As such, we need to pre-calculate K*K*N/V LUT tables for each weight vectors. Each LUT table has 2^{V*H} entries with each entry being L bits, and will be searched R*C*B times with B being the batch size, i.e., the number of images that perform inference simultaneously. As the weights are the same for different inference tasks, the LUT table reuse increases with B. Since one memory row stores only the aligned entries from M LUT tables and each entry is of L bits. The page utilization of LUT, i.e., the percentage of memory bits to be used after fetching one memory row for lookup, is computed as $M * L/P$ where P is the row size. We choose a 4KB row size in this paper.

The second mapping approach is to fix the input vector while decomposing the weight vector, referred to as Input Addend Mapping (IAW). We need to pre-calculate K*K*N/V LUT tables for each input vectors. Since the R*C*B input vectors may potentially be processed together, the page utilization of LUT is R*C*B*L/P. Since there are M weight vectors only, each LUT table is to be reused only M times.

Clearly, with different CNN parameters and VP/HP partition sizes, we may get different page utilization and LUT reuse values. We use an example to show the effect on performance. We use the third convolutional layer of Alexnet, which uses 3x3x128 weight

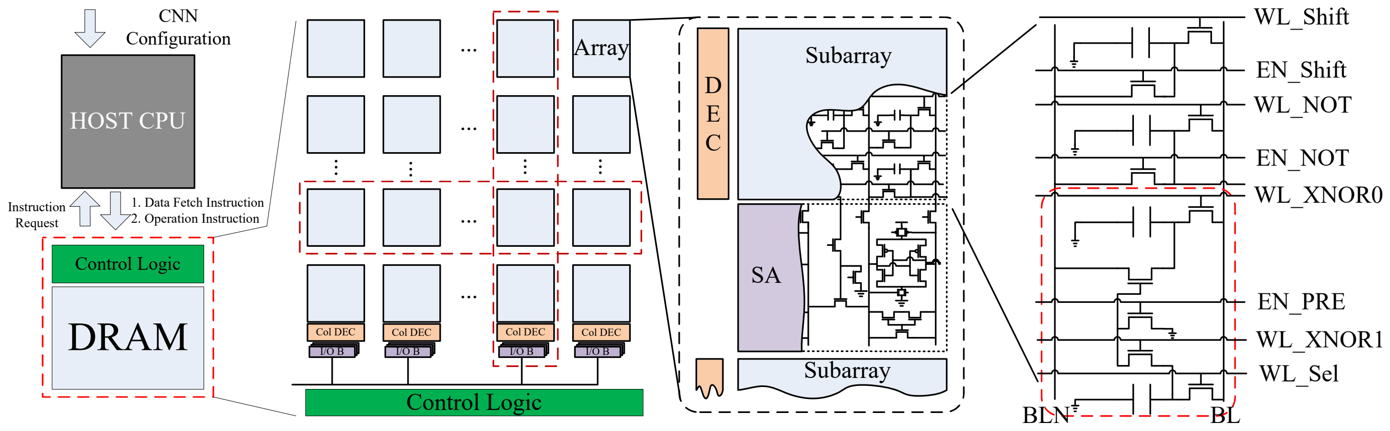


Figure 4: The Hierarchical Architecture of LAcc in DRAM.

Table 2: The Comparison of Different Addends

	Weight Addend			Input Addend		
BZ	PURL(%)	DRT	L(ms)	PURL(%)	DRT	L(ms)
1	18.75	169	5.6	8.25	384	10.8
2	18.75	338	10.7	16.50	384	11.1
4	18.75	676	20.8	33.00	384	11.6
8	18.75	1352	41.3	66.01	384	12.7

matrix, 13x13x384 output matrix. Table 2 compares the results of WAM and IAM with fixed VP and HP. BZ refers to batch size; DRT refers to data reuse time; PURL refers to page utilization rate during lookup. When the batch size is 1, WAM shows better performance than that of IAM. After the batch size increases, the performance of those two approaches reverses. With different HP and VP sizes, the performance difference is more significant, which indicates that we need to dynamically fix either the input or the weight matrices for the best performance improvement.

4 LACC ARCHITECTURE

LAcc is architected as an enhancement to a baseline DRAM main memory implementation that supports bitwise logic, fast data copy, and add operations [3, 7]. Figure 4 presents an overview of the LArcc accelerator. The arrays on the same vertical line share the same column decoder and input/output buffers while the arrays of the same horizontal line build a bank. LAcc uses the open bitline structure, where two adjacent subarrays share the same sense amplifiers.

LAcc slightly modifies on the control logic of DRAM[25] to decode the PICM commands from CPU. It adds a bypass line from the I/O buffer to the control logic and a 96B SB buffer. The control logic uses the table address and SB to generate the physical operation address. When all of the buffered commands finish, it sends instruction requests to the CPU side.

4.1 Hardware Mapping Method

Since LAcc is built as PIM DRAM accelerator, we map the processing units to banks, arrays, and subarrays. We treat each bank as a single instruction multiple data (SIMD) processing unit with active pages being the function units in operation. The input operands are stored in the same columns and different rows with the corresponding functional units. The data movement method within a subarray and across different subarrays leverage similar methods with LISA[3]. The page size of LAcc is 4KB, where there are 2048 16-bits function units in each page. LAcc preserves 4K rows of each bank for LUT, 1K rows for input data buffer and 3K rows for output buffers in each bank. The physical size of LUT decides the maximum length of VP is 12. LAcc keeps using an LUT until there are no operations within the LUT.

4.2 Data Mapping Method

LAcc uses different data mapping methods for addend, SB and output data. The addends of each thread are stored vertically in the reserved space of input buffer in the banks. For weight addend, LAcc directly uploads them from the lower level memory. For input addend, there is a data reforming step as the inputs are the output of the previous layer. The SBs and the output data are stored horizontally in the output buffer part, which is the same as original data in DRAM. When there is an SB read request, it first read the page out into the local buffer. Then the global column decoder sends the exact control bits with the same offset into control logic.

4.3 Optimized XOR Operation

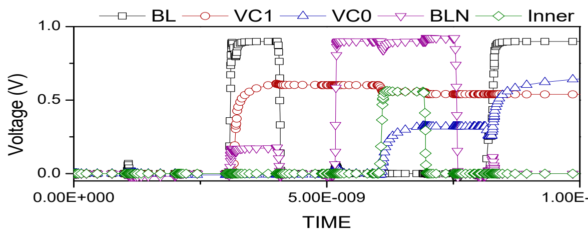
In addition to multiplication operations, a CNN needs to frequently perform add operations. The PIM based add operation implementation needs two XOR operations to implement the carry look-ahead algorithm [7]. The in-memory XOR, as being an expensive operation, account for 46% of the time of an addition operation. We next present an optimized XOR implementation to accelerate LAcc.

$$Y = A \oplus B = A\&!B + !A\&B \tag{6}$$

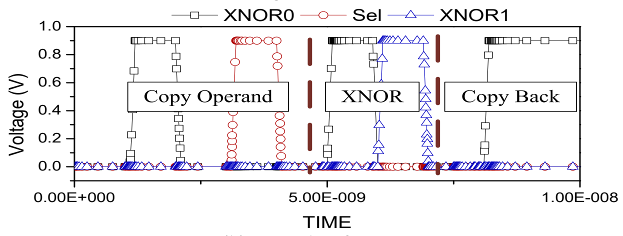
$$Y = \begin{cases} B & A = 0 \\ !B & A = 1 \end{cases} \tag{7}$$

Equation 6 and 7 illustrate two different ways to implement XOR. LAcc adopts the second one in DRAM while existing designs[7, 19] choose the first method. Figure 4 illustrates the hardware enhancement to realize this optimization. LAcc changes the connections of the last five DRAM rows. The 2nd row of the lines builds a bypass between the cell of the 1st row and BLN, and its gate connects the 4th row. The 3rd row is used to guarantee the connection between the 2nd row and the 4th row can be discharged to the ground.

To perform an optimized XOR operation, LAcc first activates the row that stores operand B and restores it into XNOR0. It then uploads operand A to Sel. Next, LAcc reads operand B into the sense amplifier and then activates the bypass of XNOR1. If A is '1', the complementary of B overwrites the original B. EN_PRE is enabled unless the WL_XNOR1 is enabled. Figure 5 shows the



(a) Signals of BLs



(b) Signals of WLs

Figure 5: Hspice Results of Optimized XNOR.

circuit simulation result with Hspice. The optimized XOR operation decreases the operating time of addition from 13 AAP operations to 10 AAP operation.

5 EVALUATION

To evaluate our proposed LAcc design, we use Hspice and CACTI to study the circuit modification and the DRAM structure, respectively. We assemble a set of benchmark programs — Lenet-5(MNIST), Alexnet, VGG16, VGG19, Resnet152, which cover both small and large input image sizes of CNNs. Lenet-5 uses 512 batch size while the other use 128 batch size. We perform fixed vertical partition (HP) of 1 but vary the granularity for horizontal partition (VP) to optimize the performance improvement. In the experiments, we evaluate the following schemes.

- `BW`. It denotes accumulation-based design with WAM.
- `BI`. It denotes accumulation-based design with IAM.
- `H`. It denotes accumulation-based design with hybrid addends.
- `RW`. It denotes LUT-based design with WAM.
- `RI`. It denotes LUT-based design with IAM.
- `HR`. It denotes LUT-based design with hybrid addends.

Table 3: The Configuration of LAcc

Bank Size	32	DRAM Tech.	25nm
Memory Capacity	8 Gb	Subarray Size	512x512
DRAM Bandwidth	4.15GBps	I/O interface	64b
Timing	ns	Energy	nJ
RAS	14.4	Activation	1.27
RP	5.8	Read/Write	0.63
RCD	8.7	Precharge	1.19

5.1 Performance

Figure 6 shows the normalized performance of those six benchmarks. On average, `HR` achieves 64.2×, 7.9× and 6.8× performance speedups over that of `BW`, `BI` and `H`, respectively. LAcc achieves the maximum performance improvement in Alexnet due to high page utilization. For this benchmark, `BI` and `RI` have low page utilization and performance for the fully connect layers of Alexnet — they spend about 50% of the total latency in these layers. Since the ratio (M/(R*C)) for the fully connected layers in Alexnet is 4096, `BI` and `RI` would need to increase their batch size to 2048 in order to achieve the same page utilization as those in `BW` and `RW`. However, their batch sizes are 128 only.

`RI` achieves better performance than `HR` in Lenet-5. This is because the sizes of its last two fully connected layers are 1*1*500 and 1*1*10, respectively. The weight addend mapping (WAM) loses its advantage on a large batch size.

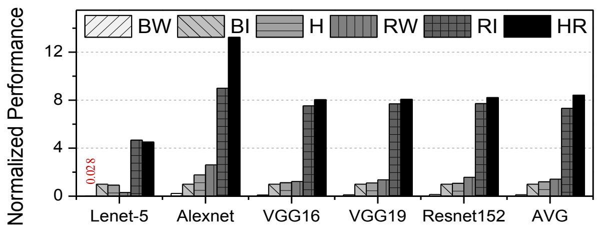


Figure 6: Comparing the performance of different schemes.

Figure 7 compares the system page utilization ratio, which equals the sum of the active page ratio multiplies the latency ratio of each layer. A 100% page utilization ratio means the operands in all active pages are effective through the entire processing time. The system page utilization ratios of `BW` and `RW` achieve the lowest, and that of `HR` achieves highest except Lenet-5. `RI` in Lenet-5 gets a 7% improvement over `BI`, while the average improvement is 3.8%. `HR` has an average improvement of 12.4% over `BI`. With a fixed batch size, the proposed design decreases the time ratio of the parts with low effective page utilization ratio.

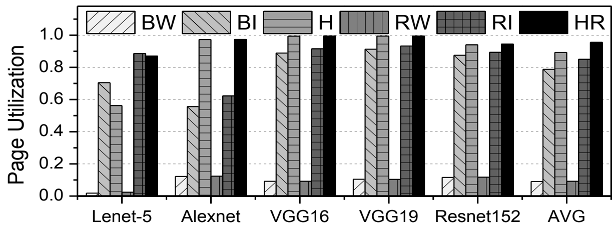


Figure 7: Comparing the page utilization.

5.2 Power and Area

Figure 8 shows the power of those six schemes, of which the difference is less than 4%. The power mainly contains two parts, i.e., the power of active pages and the power of data movement. In most of the time, LAcc leverages all the active pages in banks to do computing. Thus, the power of the first part in those schemes is very close. The difference of the power is caused by the latter one.

The overall extra area of LAcc is less than 1% compared with the baseline[7]. The optimized XNOR changes the connections of the last five DRAM rows of each subarray, which decreases 0.2% of DRAM capacity. The area of the added SB buffer in control logic is $132.2um^2$.

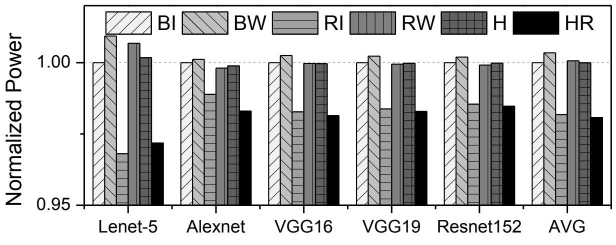


Figure 8: Power of LAcc

5.3 Batch Size Effects

Figure 9 and figure 10 show the performance and the page utilization ratio with different batch sizes in Alexnet. BW and RW show steady performance with increasing batch size, while their page utilization ratio shows the same tendency. HR shows the best performance improvement speed. The system page utilization ratio first meets a decrease with a batch size of 9. Because the parallel threads size exceeds the page size. There needs more page to hold all those threads. In return, the system page utilization ratio decreases and the speed of the performance improvement of HR slows down. The performance of RI keeps growing, which can be even better than HR with large batch size. The needed batch size can be thousands considering the fully connected layer size. However, the optional batch size of CNN is only 128/256/512. HR shows better applicability on CNN inference.

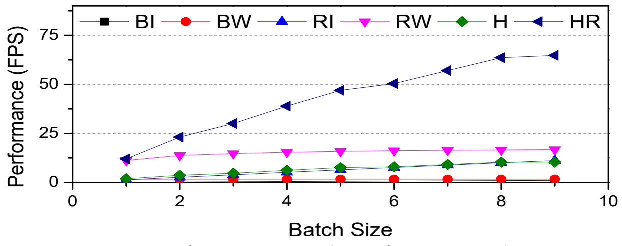


Figure 9: Performance with Different Batch Sizes

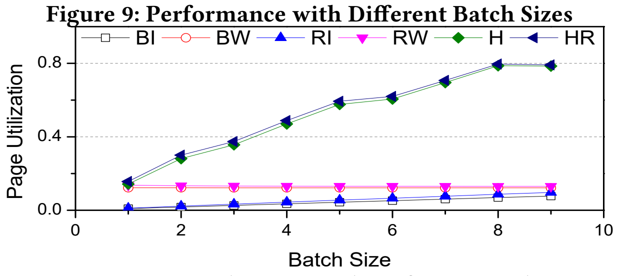


Figure 10: Page Utilization with Different Batch Sizes

5.4 Comparison with Other PIMs

We compare LAcc with an ideal scheme and three PIM designs, i.e. DRISA[16], SCOPE[17] and Neural Cache[9], to show the design efficiency of LAcc. The ideal scheme refers to the best scheme of LAcc without LUT-based vector multiplication, where we set an ideally page utilization of 100%. Neural Cache is implemented in SRAM, while the others are all implemented in DRAM. The benchmark is Alexnet, and the batch size is 64. The results of DRISA, SCOPE and Neural Cache are shrank from 8 bits to 16 bits.

LAcc achieves the best efficiency, which is 6.8× and 6.3× over the ideal scheme and SCOPE, respectively. The ideal scheme and LAcc utilize optimized XOR, which decreases the AAP number of addition from 13 to 10. The performance difference gives the credit to LUT-based vector multiplication and hybrid addend mapping. The efficiency of Neural Cache is limited by the large cell density and leakage power of SRAM. Although the efficiency of SCOPE is lower than that of LAcc, it achieves the best perf./area. Because the DRAM structures of LAcc and SCOPE are different. The maximum number of active pages of LAcc and SCOPE are 32 and 16K, and the page sizes are 4KB and 2Kb, respectively. Considering the similar contributions between SCOPE and LAcc, we make a comparison of the multiplication optimization effect. Compared with DRISA, SCOPE improves the perf./area by 4.11× and efficiency by 2.26×. Compared with the ideal scheme, LAcc improve the perf./area and efficiency by 6.82×. LAcc achieves 1.66× perf./area improvement and 3× efficiency over SCOPE on multiplication optimization.

Table 4: PIM design Comparison

Parameter	DRISA	SCOPE	Neural Cache	Ideal Scheme	LAcc
Performance	147	2528	7.2	14.0	95.6
Power(W)	98	176.4	53	5.3	5.3
Area(mm^2)	65.2	273	10.1	54.8	54.8
Capacity	2Gb	8Gb	35MB	8Gb	8Gb
Perf/Area (Fr./s/mm^2)	2.25	9.26	0.7	0.25	1.74
Efficiency (Fr./J/mm^2)	0.023	0.052	0.013	0.048	0.329

6 CONCLUSION

In this paper, we propose LAcc, a DRAM-based accelerator to support LUT-based fast and accurate vector multiplication. LAcc achieves 6.3× efficiency improvement over state-of-the-art and improves the multiplication performance 6.8× over the ideal baseline.

7 ACKNOWLEDGMENTS

This work is supported in part by NSF under Grant CCF-1718080 and CCF-1617071, in part by the National Natural Science Foundation of China under Grant 61802427 and HGJ National key project of China under Grant 2017ZX01028103. The authors thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] J, Albericio, *et al.* Bit-pragmatic deep neural network computing. *MICRO*, 2017.
[2] H, Bagherinezhad, *et al.* Lcnn: Lookup-based convolutional neural network. *CVPR*, 2017.
[3] K, Chang, *et al.* Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. *HPCA*, 2016.
[4] Y, Chen, *et al.* Dadiannao: A machine-learning supercomputer. *MICRO*, 2014.
[5] P, Chi, *et al.* PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. *ISCA*, 2016.
[6] M, Courbariaux, *et al.* Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv*, 2016.
[7] Q, Deng, *et al.* DrAcc: a DRAM based accelerator for accurate CNN inference. *DAC*, 2018.
[8] G, Desoli, *et al.* A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems. *ISSCC*, 2017.
[9] C, Eckert, *et al.* Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. *ISCA*, 2018.
[10] A, Fuchs, *et al.* Scaling Datacenter Accelerators With Compute-Reuse Architectures. *ISCA*, 2018.
[11] S, Han, *et al.* EIE: efficient inference engine on compressed deep neural network. *ISCA*, 2016.
[12] K, Hegde, *et al.* UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition. *ISCA*, 2018.
[13] P, Judd, *et al.* Stripes: Bit-serial deep neural network computing. *MICRO*, 2016.
[14] D, Kim, *et al.* Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. *ISCA*, 2016.
[15] A, Lavin, *et al.* Fast algorithms for convolutional neural networks. *CVPR*, 2016.
[16] S, Li, *et al.* Drisa: A dram-based reconfigurable in-situ accelerator. *MICRO*, 2017.
[17] S, Li, *et al.* SCOPE: A Stochastic Computing Engine for DRAM-based In-situ Accelerator. *MICRO*, 2018.
[18] M, Razlighi, *et al.* Looknn: Neural network with no multiplication. *DATE*, 2017.
[19] V, Seshadri, *et al.* Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. *MICRO*, 2017.
[20] V, Seshadri, *et al.* RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization. *MICRO*, 2013.
[21] A, Shafiee, *et al.* ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 2016.
[22] H, Sharma, *et al.* Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. *ISCA*, 2018.
[23] A, Yasoubi, *et al.* Power-efficient accelerator design for neural networks using computation reuse. *CAL*, 2017.
[24] A, Zhou, *et al.* Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv*, 2017.
[25] MICRON DRAM, *et al.* https://www.micron.com/products/dram. , 2018.