DAISM: Digital Approximate In-SRAM Multiplier-based Accelerator for DNN Training and Inference

Lorenzo Sonnino*, Shaswot Shresthamali*, Yuan He*, Masaaki Kondo*,†
{lsonnino, shaswot, isaacyhe, kondo}@acsl.ics.keio.ac.jp
*Keio University, Yokohama, Japan
†RIKEN Center for Computational Science, Kobe, Japan

***Abstract*—DNNs are widely used but face significant computational costs due to matrix multiplications, especially from data movement between the memory and processing units. One promising approach is therefore Processing-in-Memory as it greatly reduces this overhead. However, most PIM solutions rely either on novel memory technologies that have yet to mature or bit-serial computations that have significant performance overhead and scalability issues. Our work proposes an in-SRAM digital multiplier, that uses a conventional memory to perform bit-parallel computations, leveraging multiple wordlines activation. We then introduce DAISM, an architecture leveraging this multiplier, which achieves up to two orders of magnitude higher area efficiency compared to the SOTA counterparts, with competitive energy efficiency.**

***Index Terms*—approximate computing, processing in-memory, accelerator**

I. Introduction

Deep Learning (DL) has gained widespread popularity in recent years and become ubiquitous in many diverse applications, including daily tasks such as facial recognition, and applications that require extensive training like language models. As a result, many domain-specific accelerators have been proposed to streamline and optimize the computations for performance gains in terms of latency, energy, and chip area [1]. Typically, a large fraction of the computations for Deep Neural Networks (DNNs) are general matrix multiplications (GEMMs) and many accelerator designs have focused on accelerating them.

One method is to approximate the computations for performance gains by using approximate arithmetic [2] or reduced precision [3]. These methods leverage the inherent error resilience of Neural Networks (NNs) to small computational errors. It arises primarily due to parameter over-provisioning and the independent distributed computations within each layer of the NN.

Another direction for optimizing GEMMs is to use Processing-In-Memory (PIM). Since matrix multiplication is embarrassingly parallel, reading and transferring the data from memory to the processor consumes a lot of power and bottlenecks the entire computation pipeline [4], [5]. PIM solutions perform computation directly in/near memory and thus minimize this bottleneck [6], [7].

This work was supported, in part, by JST CREST from Japan with Grant JPMJCR18K1.

As attractive as PIM may be, current solutions have severe drawbacks that prevent their widespread adoption. For example, resistive memory-based designs are very sensitive to device-to-device variations and they also require conversion of data between analog and digital domains, which further drives up the energy cost and reduces throughput and accuracy. Furthermore, such analog computation-based technology requires significant changes in chip design and thereby incurs large design and manufacturing costs [8], [9].

Another alternative is in-memory bit-serial computation used, for example, by existing SRAM-based PIM technologies [10], [11]. As a consequence, this requires the data to be reorganized in a bit-serial manner and incurs significant overhead in both performance and complexity. While latency issues from bit-serial operations may be alleviated through pipelining, the area overhead and complexity cannot be overlooked. Furthermore, since most of the existing computations are optimized for bit-parallel operation, bit-serial hardware is bound to lag behind in terms of efficiency while combining these two types introduces additional complexity in designing hardware. Finally, fundamental device- and circuit-level limitations, such as the current carrying capacity of a metal wire, also prevent bit-serial solutions to scale [12].

In this work, we propose a novel in-SRAM approximate multiplier that brings the best of both worlds. Our multiplier performs matrix multiplications in memory thereby reducing the time and energy required for moving data. The multiplication is performed in a bit-parallel manner by using multiple wordlines activation that approximates multiplication with a simple bitwise OR, which is a perfect match to the tight and regular layout of conventional SRAM technology. The computational errors arising from this approximation are acceptable as DNNs are quite resilient due to over-provisioned parameters [13]. It is possible to implement our multiplier in conventional SRAMs with minimal design modification and thus making it easily adopted in existing systems. We also propose DAISM - a DNN accelerator architecture that leverages our novel in-SRAM approximate multiplier. Our evaluations show that energy and performance gains can be obtained compared to other existing baselines.

The main contributions of the paper are:

- We propose a novel in-SRAM approximate multiplier

979-8-3503-4859-0/DATE24/© 2024 EDAA

that approximates matrix multiplication with bitwise OR operation.
- We propose the DAISM architecture that leverages the in-SRAM approximate multiplier to realize performance gains and trade-offs between latency, area, and energy efficiency.
- We perform extensive evaluations on our proposed DAISM architecture and compare it with current SOTA baselines. Our results show that it is energy efficient, and requires fewer clock cycles with minimal to no degradation in model accuracy.
- We discuss and analyze the different trade-offs possible with our architecture.

This article is structured as follows. Section II recap the basics of binary multiplication, then discusses some related work and how this work differs from previously published papers. Section III presents the basic concept behind the proposed approximate multiplier as well as some ways of improving its performances. Section IV then introduces the DAISM architecture. Section V explains the evaluation methodology and discusses the multiplier and accelerator's performances. Finally, Section VI concludes this work.

II. Background and related works

A. Binary multiplication

To multiply two operands (the multiplicand and the multiplier), partial products (PP) first need to be generated. Each PP equals either the shifted multiplicand or 0 if the corresponding bit from the multiplier is 0. Those partial products then need to be added, which incurs significant overhead due to carry propagation. Meanwhile, floating point (FP) numbers consist of 3 segments: sign, exponent, and mantissa, varying in size by data type. Multiplying them involves multiplying mantissas as unsigned integers and adding exponents. The mantissa is then normalized and the exponent is realigned. Finally, the output's sign bit is an XOR of the operands' sign bit.

B. Related works

To enhance DL architecture performance, some new multipliers employ approximations [2], [3] as DNNs have a large error resilience. For instance, [2] decreases PPs by performing bitwise OR operations among them. However, they still demand adder trees. [3] instead approximates the lower part of the result via PP bitwise OR, and the upper part using approximate Full-Adder logic. Still, none of these multipliers can operate in memory. Our approximate multiplier can, which solves the data movement problem.

Other works such as [6], [7] use Processing-in-Memory (PIM), in which novel memory technologies are used for direct in-memory computations, bypassing processing units. Examples include [6] which employs RaceTrack memory for in-memory integer multiplication, and [7] which leverages ReRAM memory for MAC operations. These technologies showcase minimal energy use, no data movement, and better performances by fully capitalizing on DNNs data parallelism. Nonetheless, these memory technologies face challenges to their novelty. RaceTrack and ReRAM, being yet-to-mature, lack the research and optimization of traditional SRAM. Analog in nature, they need digital-analog-digital conversions, impairing signal, increasing power use, and limiting throughput [8], [9]. Our approach is based on conventional digital SRAM instead.

Finally, SRAM-based PIM technologies such as [10], [11] all perform bit-serial computations. They hence suffer from data reformulation and lack the support of SOTA hardware, as these are often bit-parallel. Bit-parallel multipliers instead benefit from the latest breakthrough and can be easily integrated into existing systems. Finally, in-memory bit-serial multiplication often has a very large complexity [14]. While this can be solved through pipelining, it comes with complexity and area overheads. Our architecture only uses bit-parallel hardware and can easily be implemented in existing technologies.

III. Proposed multipliers

A. Core concept

Carry propagation during partial summing decreases throughput and increases energy consumption. The proposed multiplier therefore avoids this by approximating this sum by a bitwise OR, requiring no adder tree, sacrificing some computational accuracy instead. Previous work proposed this as well, but only on the lower part of the result with no accuracy recovery mechanism [3]. Furthermore, by using a slightly modified SRAM memory thoroughly described by [15], this step can be performed in memory. Indeed, by reading multiple wordlines at the same time, a bitwise OR between them is read instead. [15] proved such technology to be viable and at a negligible cost as it only required some extra sense amplifiers. This SRAM can also function as a traditional memory but to allow multiple wordlines activation, a special address decoder must be designed though this will be proven to be negligible later on. Multiple wordlines activation has also been shown to pose no major problems in terms of signal-to-noise ratio or throughput [15]. Finally, the cost of the extra sense amplifier can be avoided by re-wiring the existing one in traditional SRAM.

Fig. 1 describes the proposed multiplier. First, the multiplicand is stored. The multiplier is then used to activate multiple wordlines, generating PPs. By doing so, a wired-OR between PP is read, which approximates the result. This multiplier will be referenced as `FLA`, standing for *Full Lines Activation*. For DNNs, the multiplicand is a kernel element, and the multiplier is an input element. The small size of most kernels makes a moderately-sized memory enough to store them, as will be discussed in Section V. Finally, this approach makes handling data represented in two's complement difficult. This work however focuses on FP mantissa multiplication, which only uses unsigned integers.

B. Storing pre-computed values

As previously stated, accuracy drops most when two successive lines must be activated. Let us assign capital letters to each PP. For instance, in an 8-bit setup, A refers to the multiplicand shifted 7 times, and H represents the unshifted

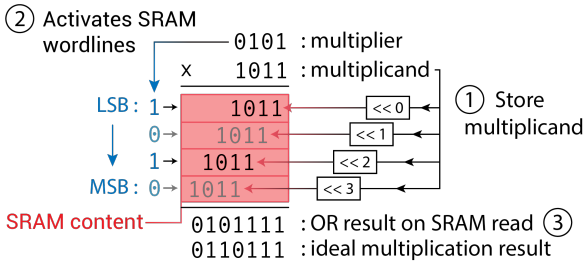

Fig. 1: Example of the proposed multiplier's concept for $a = 1011$ and $b = 0101$. The SRAM line is read if the corresponding bit from the multiplier is 1

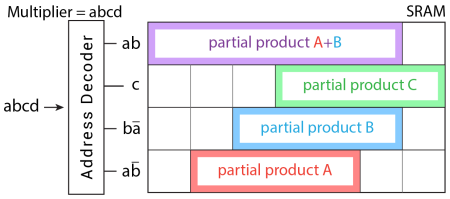

Fig. 2: In `PC2`, the pre-computed sum between the two largest PP is stored

multiplicand. Most of the accuracy loss happens when A and B's wordlines are active at the same time. Indeed, neighboring PPs have a high chance of collisions and those two directly affect the MSBs of the output. If instead of storing the LSB's PP (H for 8-bit values) the exact result of $A + B$ is stored as shown in Fig. 2, accuracy can be recovered with only a slightly more complex address decoder at no additional memory cost. This new PP would be selected whenever A and B should both be active. This is referenced as `PC2` standing for *Pre-Computed* sums between 2 partial products. All other PPs are handled as in `FLA`. This article will explore `PC3` as well, in which the pre-computed sum of all possible combinations of the A, B, and C lines are stored.

C. Floating point generalization

For FP arithmetic, the proposed multipliers are only capable of handling mantissa multiplication. The exponent and sign bits are handled separately, and multiplications by zero are bypassed.

Furthermore, the IEEE FP standard requires an extra "1" to be added at the MSB of the mantissa. This "1" is implicit in the binary representation. A 23-bit mantissa hence becomes a 24-bit unsigned integer whose MSB is 1. The PP A is hence active for all operands and, if PP B must be activated as well, the AB line (storing the pre-computed sum of these PPs) in `PC2` will be activated instead. The line for PP B will hence never be active and can be left out, reducing memory consumption. `PC3` also greatly benefits from this, as many combinations between the A, B, and C lines are no longer possible.

Moreover, because the proposed multiplier does not use any carry, the computation can be truncated arbitrarily, greatly improving performances at the cost of accuracy. This article will hence explore `PC2_tr` and `PC3_tr` in which the result is truncated to only compute the n MSB. The value of n is the mantissa width of the data type, including the leading "1".

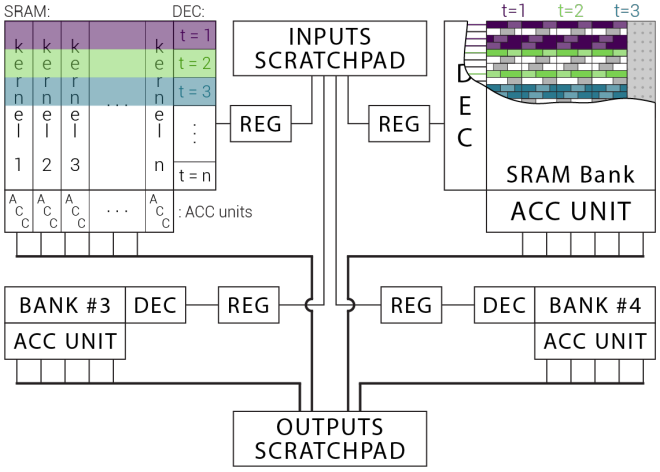

Fig. 3: 4 banks DAISM architecture. Inputs are fed one at a time to the SRAM from a register file through the address decoder. The dotted area represents unused SRAM space (not to scale)

Finally, many accelerators do not use standard FP but rather variations such as BFP. Because this multiplier handles arbitrary-size integer mantissa, any other FP representation can make use of this multiplier as long as it requires integer multiplications. This article explores the `float32` format as well as `bfloat16` [16]. The latter is similar to `float32` but uses a 7-bit mantissa instead of the standard 23. This number format is most notably used in Google's TPU.

IV. Accelerator Architecture

A. Core architecture

The proposed architecture (Fig. 3) replaces the systolic array with a large SRAM memory, modified as proposed by [15] to support a wired-OR operation through multiple wordlines activation. Each kernel would be flattened and stored as shown in Fig. 3. The inputs are taken from the top scratchpad and stored in a register file. They are then read one at a time and used to activate SRAM wordlines through an address decoder. Each input is hence multiplied by all the kernel elements on the same row at the same time. The results from these products are then fed to an accumulator at the bottom, accumulating the results of the multiplications. The final results are finally stored in another scratchpad memory.

This architecture can be applied to any variation of the proposed multiplier and any SRAM size.

B. Architecture variations

A variation of this architecture involves dividing the large square SRAM memory into smaller square banks. This eases SRAM manufacturing and allows for different inputs to be fed to different banks simultaneously, as shown for the four banks in Fig. 3.

The architecture also prefetches inputs from the scratchpad into an intermediary register file, like [1] does, except it only has one per bank. This reduces the frequency of expensive scratchpad reads, favoring the smaller register file.

This architecture is hence compatible and comparable to most systolic array architectures, only changing the way operands are multiplied and the way kernels are encoded.

TABLE I: Summary of the proposed multipliers

Config.	Precomputed wordlines	Truncation
FLA	No	No
PC2	Between 2 PP	No
PC3	Between 3 PP	No
PC2_tr	Between 2 PP	Yes
PC3_tr	Between 3 PP	Yes

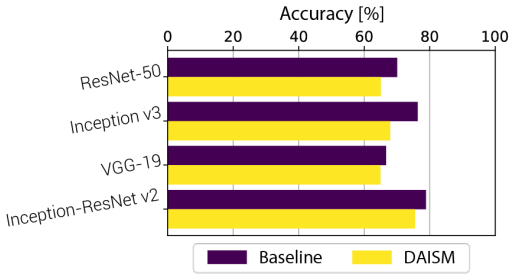

Fig. 4: Accuracy evaluation for larger CNN using bfloat16 truncated PC3 compared to an exact float32 baseline

For floating point numbers, this pipeline can only be used to multiply mantissa's as unsigned integers. The exponents must be handled separately, similar to how a block floating point architecture would work. This data type only has one exponent per matrix, reducing data size and improving performance.

V. Evaluation

This section evaluates the multipliers shown in Table I in terms of energy consumption per computation, and accuracy loss.

As a baseline multiplier for the energy consumption, the 32-bit floating point multiplier from [17] is assumed to be used in an architecture similar to Eyeriss [1] to take into account operands read. [17] is chosen as a baseline multiplier as it provides energy consumption and area for different levels of truncation. The proposed multipliers are evaluated for both float32 and bfloat16 operands. The multiplier from [17] must hence be adapted to approximate the energy consumption of a bfloat16 multiplier, as will be explained in Section V-B1.

The proposed architecture is evaluated in terms of on-chip area and performance compared to Eyeriss, using Accelergy, as will be explained in Section V-A1.

A. Accuracy

1) Methodology: The accuracy drop is evaluated on large models trained on ImageNet, such as ResNet-50 [18], [19]. The baseline uses float32 data while the proposed architectures use bfloat16.

2) Results: Fig. 4 shows the accuracy for various large CNNs when executed on PC3 compared to the FP32 baseline. Despite some accuracy drop being felt compared to the float32 baseline, DAISM is still able to achieve high accuracy on larger models while bringing energy and performance benefits as will be discussed in the following sections.

Finally, DNN inference with approximate computing is especially targeted toward edge devices that rarely employ deeper neural networks. The choice of accelerating floating point mantissa arithmetic also limits error magnitude (as opposed to integer arithmetic or exponent handling) while still providing great benefits as will be shown in Sections V-B and V-C.

B. Energy consumption

1) Methodology: The energy consumption of the proposed multipliers has been evaluated with CACTI [20], [21] and Synopsys's Design Compiler using NANGATE 45nm technology. Our multipliers are compared to the truncated float32 multiplier from [17]. For both the proposed and the baseline multipliers, operands read has been considered.

Finally, a baseline bfloat16 multiplier can be deduced by scaling [17] as in (1) in which the energy consumptions $E_{\text{sim},16}$ and $E_{\text{sim},32}$ have been simulated using NANGATE 45nm (bfloat16 and float32 respectively) and E_{16}, E_{32} are the energy consumptions of the baseline multipliers (bfloat16 and float32 respectively).

$$E_{16} = E_{32} \cdot \frac{E_{\text{sim},16}}{E_{\text{sim},32}} \cdot T \tag{1}$$

2) Results: Fig. 5 shows the energy consumption for all the proposed multipliers compared to the baseline. This includes the energy consumption required for additional components, such as the address decoder. The figure compares the energy consumption per computation across proposed multipliers, datatype, and the size of the bank performing in-memory computations. We can see the following points from Fig. 5:

1) The cost of the address decoder is negligible. It represents less than 0.5% of the energy consumption in all cases.
2) Memory read plays an important role in energy consumption. For in-memory computations, this cost is reduced by the many reuses and by reading one operand from a register file.
3) While using a smaller memory decreases energy consumption per read, the decrease in the number of computations per memory read cancels out the benefits. There is hence no major difference in terms of energy consumption per computation.
4) Truncation allows for drastically improved performances as it nearly doubles the number of computations per memory read. Truncation therefore nearly halves memory read energy consumption, greatly reducing overall energy consumption.

The In-Memory multipliers require fewer data movements to perform a multiplication operation. Indeed, one operand can be left in place, unlike a traditional multiplier. While this has been shown to have a significant impact on energy consumption [4], [5], it has not been taken into account as when integrated into a DL accelerator, data movement is still required between the SRAM and an accumulator.

Exponent adding and realignment are common costs for both the baseline and the proposed multipliers. Adding this common cost reduces the benefits realized by using the proposed multipliers. Fig. 6 shows the improvement in energy consumption when using PC3_tr compared to the baseline.

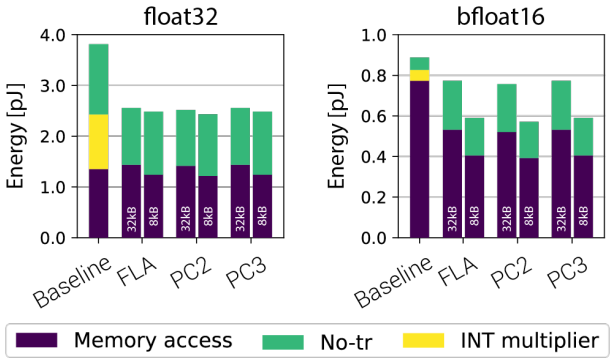

Fig. 5: Energy break-down for all the proposed mantissa multipliers compared to a common baseline for either a 32kB or an 8kB SRAM. *No-tr* represent the spared energy consumption by truncation

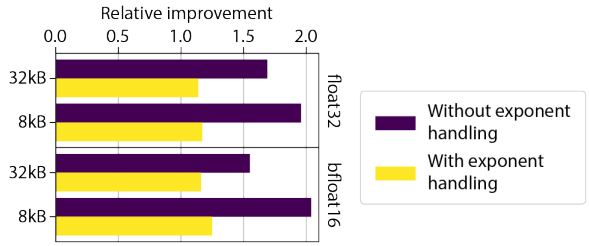

Fig. 6: Relative improvement in energy consumption when taking into account exponent handling for different SRAM bank sizes and data types

Finally, the cost of pre-loading data is made negligible by the large operands reuse. For instance, The first layer of VGG-8 has 150,528 inputs for 1728 kernel elements. This means that each input is reused for a very large number of kernel elements and each kernel element is reused for thousands of inputs, making the cost of any pre-loading negligible.

C. Architecture evaluation

1) Methodology: The proposed architecture and its variations (with a varying number of banks and memory size) are compared to the Eyeriss architecture [1] using Accelergy and Timeloop [22]. VGG-8 will be used to evaluate the architecture as it is widely used and allows us to better highlight the key differences between the architectures as it uses a larger number of processing elements and require a larger amount of memory.

All architectures use the `bfloat16` datatype. The area of a truncated `bfloat16` has been computed the same way the energy consumption has been computed in V-B1.

Finally, in-memory technologies have not been evaluated as they often perform full integer MAC operations and are therefore unable to perform floating point operations.

2) Results: Fig. 7 shows the trade-off between the number of cycles and the on-chip area to execute the first layer of VGG-8 on different architectures. The proposed architecture has been evaluated by using one single 512kB or 8kB SRAM memory, then by splitting it into smaller square banks.

Single bank architectures suffer from the lack of inputs that can be fed at each cycle. Indeed, some input elements must not be multiplied by all kernel elements, which decreases

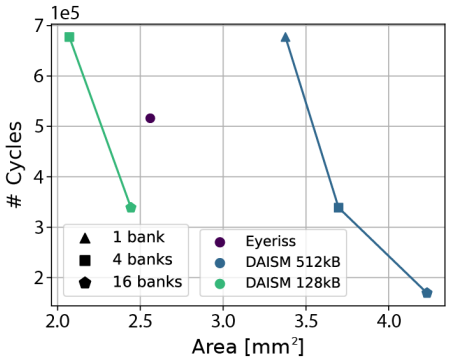

Fig. 7: Architectures performances comparison when executing the first layer of VGG-8 in `bfloat16` representation between the proposed `PC3_tr`-based architecture with different 45nm variations

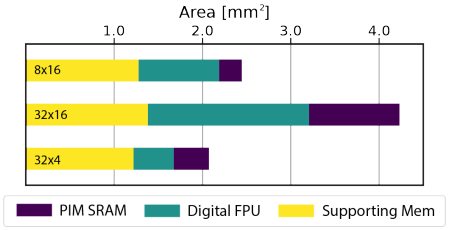

Fig. 8: Detailed area breakdown of the DAISM architecture

utilization. Moreover, while DAISM suits any memory shape, a standard squared memory is assumed. While such a 512kB bank can store up to 128x256 kernel elements, the considered layer only has 1728, leaving most of the memory unused.

Furthermore, the 1x512kB architecture can only use 128 kernel elements at a time. Hence, dividing the SRAM memory into smaller square banks, each taking distinct inputs at each cycle, decreases the number of cycles at the expense of some on-chip area and a larger data bus connecting the scratchpad to the SRAM banks increasing costs. As a consequence, the 16-bank design has 512 processing elements which are about 3x those of Eyeriss. This however requires more hardware for exponent handling and accumulation.

Decreasing the total on-chip memory allows an increase in the number of banks while maintaining a small on-chip area. This makes the 16 banks of 8kB variation the smallest architecture while maintaining the same performance as the 128kB bank one.

In Fig. 8, the main SRAM's area relative to other required digital circuits (exponent handling, accumulators) for each processing element is shown. When the SRAM's width is increased, its area is squares quadratically while the number of PE increases linearly. The opposite is true when the number of banks increases instead. Therefore, as memory banks get larger, the area becomes dominated by the SRAM memory with little performance benefits. However as the number of banks increases, the area becomes dominated by other digital circuits, and a larger cost is associated with each bank.

Table II compares DAISM to Z-PIM [10] and T-PIM [11] which both use digital in-SRAM computation logics. Despite using 45nm technology, DAISM achieves comparable energy

TABLE II: Performances comparison between different PIM architectures

Architecture	DAISM		Z-PIM [10]	T-PIM [11]
Config	16x8kB	16x32kB	—	—
Computations	bit-parallel		bit-serial	bit-serial
Node [nm]	45		65	28
Area [mm^2]	2.44	4.23	7.57	5.04
GE Area[§] [mm^2]	3.81	6.61	5.91	15.51∼24.83
Clock [MHz]	1000		200	50∼280
Supply [V]	1.0		1.0	0.75∼1.05
GOPS	502.52	1005.04	1.52∼16.0*	5.56[†]
GOPS/mW	0.23		0.31∼3.07*	0.13∼1.26[‡]
GOPS/mm^2	205.68	237.55	0.53∼5.31*	1.1[†]

§ Gate Equivalent area computed using nodes from [23]
* Varies according to the weight sparsity (0.1∼0.9).
† Measured with input sparsity of 0.9 and weight sparsity of 0.5.
‡ Varies with the input sparsity (0.1∼0.9); weight sparsity is set to 0.5.

efficiency but up to two orders of magnitude higher overall performance and area efficiency with 16-bit inputs and weights. On the other hand, this advantage in computation density over Z-PIM and T-PIM remains an order of magnitude higher even if the operating frequency of DAISM is scaled down to 200MHz.

D. Final analysis

Prior sections outlined trade-offs in the proposed multipliers and their impact.

First, between the evaluated multipliers, `PC3` is the best choice for three reasons:

1) `PC3` has better accuracy;
2) `PC3` requires fewer simultaneously active wordlines;
3) The cost in terms of energy consumption per computation is similar.

Truncation minimally affects accuracy, but significantly enhances energy efficiency per computation due to increased computations per memory read. Additionally, our bit-parallel approximate multiplier employs multiple wordlines for storing partial products, temporarily using more SRAM. This doesn't pose a major problem as SRAM is abundant and can accommodate many kernels at runtime (each kernel may take around a few tens of bytes). Moreover, when batch size is large during inference, it amortizes the cost of populating SRAM with the shifted bit patterns.

Finally, The proposed architecture improves performance through more processing elements and reduces energy consumption compared to Eyeriss due to lower per-computation energy. A trade-off exists between performance and on-chip area, which can be fine-tuned by selecting an appropriate number of banks and memory size. Table III summarises the key benefits of the proposed multiplier and accelerator compared to other technologies.

VI. Conclusion

In this article, we propose multiple variations of an approximate digital in-SRAM multiplier for multiple data types. Most notably, the `PC3_tr` variation stores pre-computed values, then uses an in-memory wired-OR to combine them, approximating the result. This allows for a decrease in energy consumption compared to a traditional multiplier while avoiding large accuracy drops. Finally, an accelerator architecture has also been introduced, capitalizing on this multiplier. Through our comprehensive evaluations, this accelerator has been shown to outperform Eyeriss, a cutting-edge accelerator, for a comparable chip area, and it is more area efficient than two SOTA SRAM-based PIM counterparts.

TABLE III: Summary of the key differences between the DAISM accelerator and related work

	Data Movement	**Type of Computation**	**Memory Technology**	**Memory Reads**
DAISM	**None**	**Digital**	**Legacy**	**Single**
Digital Multipliers	Required	Digital	Legacy	Single
Analog PIM	None	Analog	Novel	Single
SRAM Digital PIM	None	Digital	Legacy	Multiple

References

[1] Y.-H. Chen *et al.*, "Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE JSSC*, vol. 52, no. 1, pp. 127–138, Jan 2017.
[2] I. Qiqieh *et al.*, "Energy-efficient approximate multiplier design using bit significance-driven logic compression," in *DATE'17*, Mar 2017, pp. 7–12.
[3] Y. Guo *et al.*, "Design of power and area efficient lower-part-OR approximate multiplier," in *TENCON'18*, Oct 2018, pp. 2110–2115.
[4] G. Singh *et al.*, "Near-memory computing: past, present, and future," *MICPRO*, vol. 71, no. C, Nov 2019.
[5] T.-J. Yang *et al.*, "A method to estimate the energy consumption of deep neural networks," in *ACSSC'17*, 2017, pp. 1916–1920.
[6] T. Luo *et al.*, "Energy efficient in-memory integer multiplication based on Racetrack memory," in *ICDCS'20*, Nov 2020, pp. 1409–1414.
[7] H. Jin *et al.*, "ReHy: a ReRAM-based digital/analog hybrid PIM architecture for accelerating CNN training," *IEEE TPDS*, vol. 33, no. 11, pp. 2872–2884, Dec 2022.
[8] M. Hassanpour *et al.*, "A survey of near-data processing architectures for neural networks," *MAKE*, vol. 4, no. 1, pp. 66–102, Jan 2022.
[9] S. Mittal, "A survey of ReRAM-based architectures for processing-in-memory and neural networks," *MAKE*, vol. 1, no. 1, pp. 75–114, Apr 2018.
[10] J.-H. Kim *et al.*, "Z-PIM: a sparsity-aware processing-in-memory architecture with fully variable weight bit-precision for energy-efficient deep neural networks," *IEEE JSSC*, vol. 56, no. 4, pp. 1093–1104, Jan 2021.
[11] J. Heo *et al.*, "T-PIM: an energy-efficient processing-in-memory accelerator for end-to-end on-device training," *IEEE JSSC*, vol. 58, no. 3, pp. 600–613, Nov 2023.
[12] S. Hamdioui *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *DATE'15*, Mar 2015, pp. 1718–1725.
[13] S. Shresthamali *et al.*, "FAWS: fault-aware weight scheduler for DNN computations in heterogeneous and faulty hardware," in *ISPA'22*, Dec 2022, pp. 204–212.
[14] J. Wang *et al.*, "A 28-nm compute SRAM with bit-serial logic/arithmetic operations for programmable in-memory vector computing," *IEEE JSSC*, vol. 55, no. 1, pp. 76–86, Sep 2020.
[15] Q. Dong *et al.*, "A 0.3v VDDmin 4+2t SRAM for searching and in-memory computing using 55nm DDC technology," in *VLSIC'17*, Jun 2017, pp. C160–C161.
[16] N. Burgess *et al.*, "Bfloat16 processing for neural networks," in *ARITH'19*, Jun 2019, pp. 88–91.
[17] P. Yin *et al.*, "Design and performance evaluation of approximate floating-point multipliers," in *ISVLSI'16*, Sep 2016, pp. 296–301.
[18] K. He *et al.*, "Deep residual learning for image recognition," Dec 2015.
[19] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, Dec 2015.
[20] R. Balasubramonian *et al.*, "CACTI 7: new tools for interconnect exploration in innovative off-chip memories," *ACM TACO*, vol. 14, no. 2, Jun 2017.
[21] N. P. Jouppi *et al.*, "CACTI-IO: CACTI with off-chip power-area-timing models," *IEEE TVLSI*, vol. 23, no. 7, pp. 1254–1267, Aug 2015.
[22] Y. N. Wu *et al.*, "Accelergy: an architecture-level energy estimation methodology for accelerator designs," in *ICCAD'19*, Nov 2019, pp. 1–8.
[23] Semiconductor Industry Association, "Overall roadmap technology characteristics," accessed: 2024-01-06. [Online]. Available: https://www.semiconductors.org/wp-content/uploads/2018/08/2003Overall-Roadmap-Technology-Characteristics.pdf