Look-Up Table based Energy Efficient Processing in Cache Support for Neural Network Acceleration

Akshay Krishna Ramanathan[1*] Gurpreet S Kalsi[2] Srivatsa Srinivasa [1] Tarun Makesh Chandran[1]
Kamlesh R Pillai[2] Om J Omer[2] Vijaykrishnan Narayanan[1] Sreenivas Subramoney[2]
(1) The Pennsylvania State University (2)Processor Architecture Research Lab, Intel labs
{axr499, sxr5403, mzc88, vijaykrishnan.narayanan}@psu.edu
{gurpreet.s.kalsi, kamalesh.r.pillai, om.j.omer, sreenivas.subramoney}@intel.com

***Abstract*—This paper presents a Look-Up Table (LUT) based Processing-In-Memory (PIM) technique with the potential for running Neural Network inference tasks. We implement a bitline computing free technique to avoid frequent bitline accesses to the cache sub-arrays and thereby considerably reducing the memory access energy overhead. LUT in conjunction with the compute engines enables sub-array level parallelism while executing complex operations through data lookup which otherwise requires multiple cycles. Sub-array level parallelism and systolic input data flow ensure data movement to be confined to the SRAM slice.**

Our proposed LUT based PIM methodology exploits substantial parallelism using look-up tables, which does not alter the memory structure/organization, that is, preserving the bit-cell and peripherals of the existing SRAM monolithic arrays. Our solution achieves 1.72x higher performance and 3.14x lower energy as compared to a state-of-the-art processing-in-cache solution. Sub-array level design modifications to incorporate LUT along with the compute engines will increase the overall cache area by 5.6%. We achieve 3.97x speedup w.r.t neural network systolic accelerator with a similar area. The re-configurable nature of the compute engines enables various neural network operations and thereby supporting sequential networks (RNNs) and transformer models. Our quantitative analysis demonstrates 101x, 3x faster execution and 91x, 11x energy efficient than CPU and GPU respectively while running the transformer model, BERT-Base.

***Index Terms*—Processing-in-memory, SRAM, Look-up table, Neural networks**

I. Introduction

The rapid growth of application data volumes and the increasing gap between speed of logic and memory calls the conventional von-Neumann architecture based computing systems into questions regarding their compute efficiency. The excessive energy and latency costs associated with data movement have resurrected interest in processor-in-memory (PIM) architectures [1], [2]. These designs blur the gap between compute engines and storage to alleviate the data movement costs. Several emerging non-volatile memory technologies such as Resistive Random Access Memory (RRAM) [3], [4], [5], [6], [7] cross-point arrays augment the primary role of data storage with intrinsic computation support for the multiply-and-accumulate (MAC) operation facilitating various PIM design explorations. Further, the data-intensive nature of many emerging applications such as deep neural networks (DNN) benefit from in-place data manipulation abilities of PIM architectures.

SRAM memories predominantly have been on-chip caches in processor architectures constituting up-to 70% of the overall chip area [8]. The SRAM based PIM solutions take advantage of the same, and convert the huge memory area into compute units. They also leverage the potential for much higher internal data bandwidth within the cache than to external logic. Most of the existing state-of-the art SRAM based PIM designs [9], [10], [11] perform computation by asserting multiple rows of memory to establish data-dependent bitline discharge. This technique is used in conjunction with modified sense amplifiers or the augmentation of digital logic at the edge of the SRAM array to perform various logical functions within an array [11]. While such bitline computing offers potential for enormous parallelism in computing across all columns of a large cache, it imposes significant energy consumption involved with charging and discharging the bitlines. This overhead can become excessive for complex operations of DNN workloads that need to be broken down into a large sequence of simple bitline operations

Our work introduces a LUT-based bitline computing free PIM solution for in cache acceleration of various DNN workloads. It efficiently leverages the array-level parallelism for performance gains, while reducing the energy consumption of bitline computing approaches. The lookup table (LUT) based computational approaches have been widely used in the FPGA and custom-ASIC architectures to implement a wide variety of functions [12], [13], [14]. Specifically, Taylor's series expansion of functions (exponent, trigonometric functions) is leveraged for hardware implementation using a finite number of stored entries in a LUT along with minimal logic [15]. Further, there have been approaches that configure the on-chip SRAM in Field Programmable Gate arrays (FPGA) for supporting fixed-precision multiplications to augment the hardwired multiply-accumulate logic for DSP applications [16], [17]. Similarly, LUTNet [18] makes use of LUTs for realizing XNOR-gate to run Binarized Neural Networks. These existing LUT-based hardware implementations inspire us to design a LUT-based PIM architecture, as an alternative to

*This work was done as part of internship at Processor Architecture Research Lab, Intel Labs, Bangalore, KA, India.

This work was supported in part by Semiconductor Research Corporation (SRC) Center for Research in Intelligent Storage and Processing in Memory (CRISP).

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00020

bitline based architectures.

DNN workloads (CNNs, RNNs, Transformer models etc.) comprise various operations such as multiply-and-accumulate (MAC), normalization, element wise scalar arithmetic, and diverse range of non-linear functions like Sigmoid, Tanh, ReLu, and Softmax. In order to support end-to-end acceleration of these diverse set of workloads, accelerators need to incorporate the necessary compute logic for all the above mentioned operations, and thereby potentially incur area and static power consumption overheads. The reconfigurability provided by the LUT based compute engines aids in performing these diverse operations within the memory efficiently without sacrificing logic area overhead needed for these special functions. Most existing inference accelerators for DNN workloads incorporate integer-based (reduced precision) computations by leveraging quantization techniques for improved latency and energy consumption. In addition, recent works on quantization demonstrate that, it is beneficial to have various integer bit precision for different network layers for faster inference times, and energy efficiency without compromising on the accuracy. The need for varied bit precision, and the efficiency of DNNs with reduced compute precision further makes a better case for LUT-based approaches. LUTs can be reconfigured to support different precision and fewer LUT entries are required for reduced precision operations.

LUT based approaches can replace a sequence of operations in bitline based SRAM architectures with a single read operation from a memory subarray. The energy and latency benefits for such LUT based operations are especially significant for primitives that fundamentally take more cycles to compute such as multiplication or division. However, the integration of such LUT functionality for individual functions in custom designs is significantly different from our goal to support LUT based compute within SRAM caches. The LUT based compute should result in minimal perturbation in either latency or power consumption to the basic operations of read and write required from the caches.

Towards exploring the design space of LUT based compute in caches, this paper makes the following contributions.

- We present a bitline computation free PIM architecture (we name it BFree) capable of computing various complex neural network primitives at the subarray granularity. Alternate to bitline computing, BFree does not make use of multi row activation and repetitive bitline accesses while performing complex PIM operations like MAC, division, square root, exponent, sigmoid, tanh, softmax, etc.
- BFree transforms the subarray into a LUT-based compute engine and makes use of a tiny PIM controller named BFree compute engine (BCE). The BCE also aids in inter/intra subarray communication. We describe qualitatively and quantitatively the energy efficiency of this architecture as compared to state-of-the-art PIM solutions using bitline computing approach.
- We enable the systolic data movement strategies for the matrix multiply and MAC-based operations within the memory which leads to further improvement in performance by overlapping computation with input load time.
- We demonstrate the reconfigurability of the BFree architecture with different precision execution of the layers within the same network (switching between 4-bit, and 8-bit precision). We also show flexibility by executing different DNN workloads (CNN, RNN, Transformer models) within the same fabric.

Rest of the paper is organized as follows. Section II describes the background of neural network primitives, memory organization and motivates bitline compute free PIM for energy efficiency. Section III focuses on the BFree architecture. Section IV maps kernels in various neural networks on to the BFree architecture with PIM support. Section V describes in detail the analysis strategy and evaluation of our work. Section VI briefs about the relevant related works. Section VII summarizes and concludes.

II. Background & Motivation

In this section, we give an overview of cache memory organization in the existing general purpose architectures, and the basics, design considerations and various challenges in bitline computing-based PIM architectures.

A. Cache Organization

SRAM memory in a general purpose processor is typically organized as multiple cache hierarchies. Fig.1 shows the structure of the last level L3 cache hierarchy (similar to Intel E5 processor with 35MB L3 cache) for a specific processor with 14 cores. A typical L3 cache consists of multiple 'slices' and each of them will have direct access to its corresponding processor core. Fig.1(a) illustrates an L3 cache hierarchy made up of 14 slices. Slice0 can be accessed by its corresponding

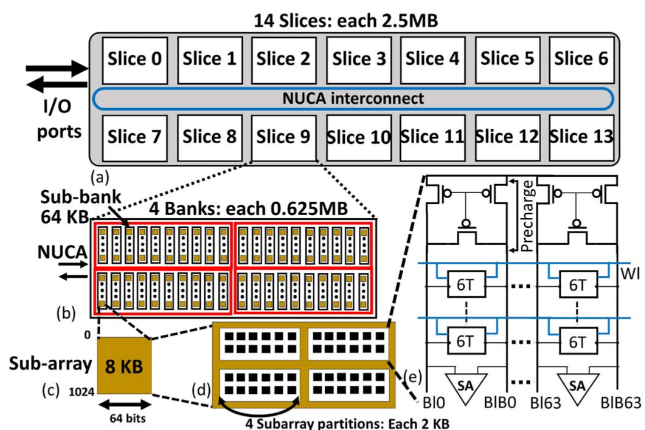


Fig. 1. Memory organization in a typical last level cache. (a) Slices and ring interconnect with NUCA support. (b) Slice partitioned into multiple banks, and each bank further into sub-banks. (c) 8KB subarray with 1024 rows and 64 cells in a row. (d) Subarray partitions and memory peripherals. (e) Subarray partition constituting of SRAM cells with precharge circuitry and sense amplifiers for data access.

processor core0 directly and access to other cores is established through a ring interconnect. Fig.1(b) shows a single slice with four banks, each bank consists of ten sub-banks and the sub-bank is further divided into 8KB subarrays. Every sub-array has four partitions (see Fig.1(c)) along with the necessary sub-array peripherals and timing circuitry (not shown in the figure). This kind of memory organization helps to retain the overall memory density by containing all the peripherals in the sub-arrays and also by avoiding splitting of address and data bus. Sub-array peripherals are responsible for address decoding (decoder), MUXing the entire row of data into the data bus (column multiplexer), precharge circuit (precharging the bitlines while reading), sense amplifier to sense the bitline discharge and write drivers to load the bitlines with the data to be written. Any PIM strategy which disturbs this tightly coupled sub-array organization will either negatively impact the data access (memory property) or increase the PIM execution latency. BFree architecture considers the same cache organisation and consciously incorporates the PIM compute circuitry at sub-array level with minimal perturbations to the peripheral circuitry and memory circuitry at bit-level or bank level. The resulting design has minimal impact on conventional memory performance and still achieves massive parallelism.

B. Bitline Computing and Design Considerations: Overview

One of the most common PIM techniques is to assert multiple wordlines where the input operands are located and establish a bitline discharge to obtain the computed output. Boolean operations take one compute cycle when both operands are stored in the same partition of the subarray. These computations are limited to bitwise operations. In order to obtain massive parallelism while performing multibit logic operations such as addition or MAC, input operands have to be stored in bit-serial fashion and the computation takes more than one cycle. However, certain constraints have to be met in designing the memory array with robust multiple row activation (MRA). Memory operations are very sensitive to noise especially under low voltage and scaled technology nodes. Activating multiple rows simultaneously for data read will cause a write bias condition and there are chances of data corruption. Zhang et al. [19], show that the MRA operation results in decrease of noise margin index. To address this, wordline voltage needs to be reduced to at least two-thirds of the supply voltage. Reduction in the supply voltage directly impacts the computation speed while needing to activate separate wordline under driving logic. Also, tweaking of wordline decoder is required and two sense amplifiers need to be designed (conventionally one) per column on the memory. In addition, additional computation logic (to execute addition and MAC) is required near the sense amplifier and column-wise data access extension need to be introduced. Even with all these PIM peripherals, operations take multiple compute cycles (approximately n cycles for add operation, and n^2 cycles for multiply operation). While the area overhead is still small compared to the overall chip area, the high number of activations on the bitline impede the energy efficiency.

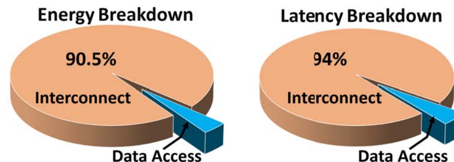


Fig. 2. Energy and Latency breakdown of a data access showing the impact of the interconnects

C. Bitline Computing: Pros and Cons

Fig.2 shows the breakdown of data access latency and energy in accessing an SRAM slice. Interconnect between the subarray and the slice port contributes to more than 90% of overall latency and energy. The sub-array access dominated by the bit line access contributes 6% and 9% to overall latency and energy. Hence it is desirable to avoid data movement through the highly parasitic interconnect and compute within a subarray. Further, if the bitline accesses are reduced the most significant part of subarray energy will also be mitigated. Further, massive parallelism can be enabled by concurrent processing at individual sub arrays. PIM performance can be determined by the number of operations per PIM cycle (PIM-OPC). PIM-OPC is an indicator of the degree of parallelism within the subarray. For example, considering the column muxing of 4:1, for the subarray in Fig.1(c), 8 Boolean operations (8-bit operands and bit-parallel computing) are possible in one PIM cycle. Hence PIM-OPC in this case is 8 considering all the 64 bitlines computing in parallel. Similarly, multiplication requires operands to be stored column-wise for bit-serial computation [9], [20], [21] through MRA. Even in this case, computation parallelism across all the bitlines is achieved. However, a 8-bit multiplication takes 102 PIM cycles [9]. Therefore PIM-OPC, in this case, is approximately 0.63 (the number of bitlines divided by the PIM cycle = 64/102) which is much less than 1. To achieve higher PIM-OPC, the number of columns within each subarray partition needs to be increased. An increase in the number of cells per row is a drastic change in the organization that will have serious implications on memory performance and is not recommended. Any technique with fewer PIM execution cycles eliminates the need for bitline granular computation while achieving the same performance through subarray level parallelism. Reading the operands from the subarray and designing a compute logic with fast execution (less than a PIM cycle) will alleviate the stress on bitline thereby achieving energy efficiency and performance improvement due to reduced PIM cycles.

D. Bitline Computing Alternatives

Various compute approaches can be adopted at the subarray level to perform bitline free computing. A straightforward approach is to integrate specialized hardware within each subarray. Neural networks perform multiple operations at every level. Consequently adding multiple specialized units will result in significant degradation of memory density and adversely impact normal memory operations. Instead, we introduce BFree, a LUT based compute that provides

configurable support for multiple operations. This approach minimally increases the memory area by introducing only hardware that assists in combining LUT entries to realize operations such as multiply, MAC, division, activation and pooling for varied input bit widths. The proposed design implements a LUT in the subarray that stores precomputed outputs and an associated small control logic to offer flexibility in terms of functionality, without the need to modify the subarray peripherals. BFree adopts the subarray level parallel computing approach, incorporating LUT at every partition within the subarray and BCE logic at the edge of the subarray, thus obtaining high energy efficiency for applications with numerous MAC and arithmetic operations.

In the subsequent sections, we will describe in detail qualitatively and quantitatively about LUT design strategy, BFree architecture, design space exploration for enabling seamless systolic dataflow within the cache.

III. BFree Architecture

In this section, we provide an overview of BFree architecture and its support for various kernels in diverse DNN workloads. BFree architecture incorporates compute capabilities at sub-array granularity in the conventional cache memory organization. Few rows in each partition of the sub-array are equipped with custom peripheral circuitry for the reduced cost access for LUT entries. In addition, BFree compute engine (BCE) comprises of additional control and compute logic, hardwired multiply-LUT to efficiently support the execution of various kernels. We discuss about the organization of BCE and its execution flow, LUT-based support for various kernels and the efficient systolic dataflow incorporated in the proposed BFree architecture.

A. BFree Sub-array and BCE Organization

Fig.3 shows the architectural overview of the sub-array with LUT and BCE. Conventionally, the sub-array is divided into 4 partitions, and all the partitions share the same timer and decoder logic (labelled as T&D). BFree sub-array design incorporates reduced access cost rows for LUT and CB storage. The configuration block (CB) stores the metadata such as bit-precision, operation, number of iterations, starting and ending address for the computation to be performed. BCE is responsible for orchestrating the PIM operations at the sub-array granularity, as well as aid in the systolic communication. It consists of fetch and decode logic for the PIM instructions, control logic to schedule LUT accesses and other arithmetic operations in sub-array and BCE. Since multiplication operation is the most widely used compute operation in the DNN workloads, a hardwired multiply-LUT (ROM in the Fig.3) is introduced in the BCE to reduce the number of accesses to sub-array partitions. BCE snoops on the sub-array data/address bus for scheduling the PIM operations and does not incur any additional interconnect overheads at sub-arrays. BCE incorporates a simple three stage in-order pipeline. In the first stage, it reads the metadata from CB and decodes the PIM instruction. Decoded instruction consists of information such as operation parameters, starting and ending address of the weights stored in the sub-array. In the second stage, it generates the necessary LUT addresses (in accordance with the sub-array address) based on the operands and the type of operation. In the third stage, the partial results from the LUT lookup are further accumulated/processed based on the operation and the final result is stored in the output registers.

B. Look Up Table (LUT) Implementation

The number of entries in the LUT plays a significant role in determining the PIM performance. For example, to compute the output of 8-bit multiplication in one cycle, LUT should store 65,536 ($2^8 \times 2^8$ input combinations) pre-computed entries (each of two bytes) which makes the design very impractical. Hence, the design should consider the tradeoff between number of LUT read cycles and the size. Each sub-array in the BFree design stores the LUT entries and configuration block, which stores the metadata (operands, type of operation etc.) corresponding to the PIM instructions. We explored three distinct design strategies for carefully optimizing LUT latency and area. In first approach, we design a standalone LUT with

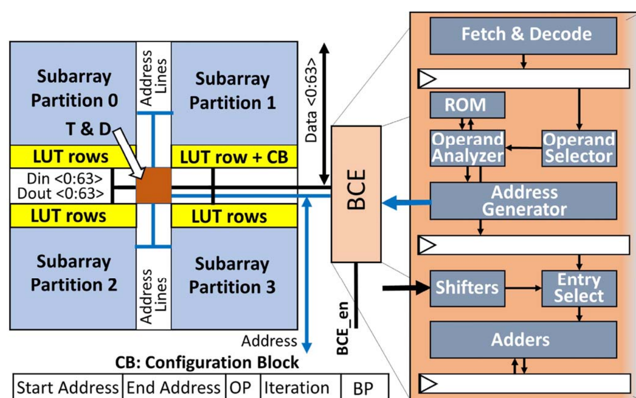
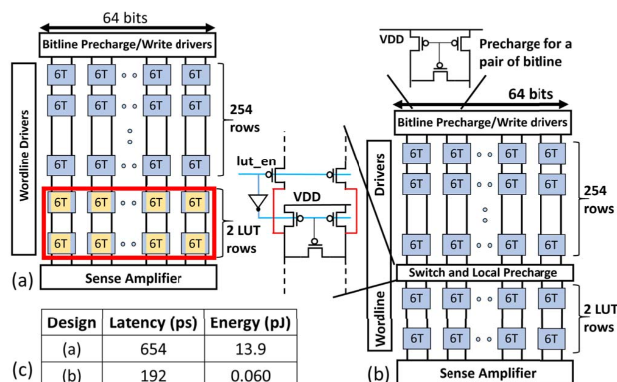


Fig. 3. BFree architecture at the sub-array level showing LUT, pipeline stages of the BCE and CB address field



Design	Latency (ps)	Energy (pJ)
(a)	654	13.9
(b)	192	0.060

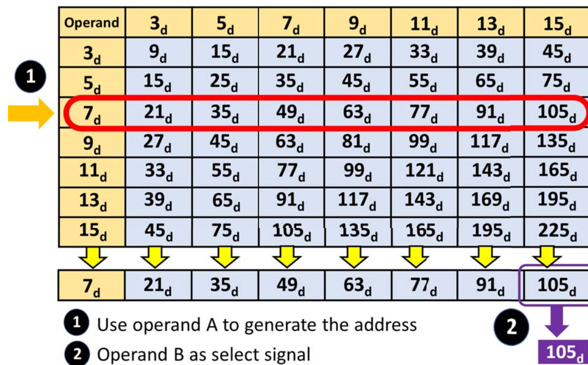
Fig. 4. (a) Dedicated low cost access LUT rows in one of the four partitions of a sub-array. (b) Local precharge and segregated bitline for fast LUT access. (c) Latency and energy comparison

separate LUT peripherals. A LUT for 4-bit multiply operands requires two 256 8-bit entries with a total memory size of 256 bytes. This approach significantly impacts the sub-array area and performance. In the second approach, we allocate a certain portion of the existing memory sub-array for storing the LUT entries. Fig.4(a) shows one of the four partitions of a sub-array in which two rows are dedicated for LUT entries. Hence, a sub-array with four partitions will have in total eight dedicated LUT rows (64 entries). LUT access latency and energy will be the same as accessing any row of a partition since the LUT shares the parasitic bitline of that partition (see Fig.4(a)). While memory density is unaffected, high access latency and energy will make the PIM operations inefficient. In the third approach, we designed decoupled bitlines for the LUT rows alone for reduced access cost. In cache mode (lut_en = 0), a single bitline runs across the entire column cells as shown in Fig.4(b). In PIM mode (lut_en = 1), we activate a local precharge circuit connecting only to the LUT region of the sub-array partition. This reduces additional load on the bitline and thus making data lookup 3x faster and 231x energy efficient, while utilizing the already available array peripherals. High energy gains are attributed to the higher Vth transistors used for the 2 LUT rows, and the precharge circuitry, as well as, the minimum sized precharge drivers. Introduction of the additional precharge circuitry increases the sub-array area by a meager 0.5%.

C. LUT: Supported Operations

This section describes the various optimizations and mapping strategies adopted to support diverse operations in the DNN workloads to LUT PIM operations. Naive mapping of compute operations to LUTs require huge storage space, which cannot be accommodated at cache sub-array granularity. We adopt several optimization strategies to reduce the LUT storage space.

1) Multiplication: For reducing the number of LUT entries, we only store the entries for 4-bit operands in the sub-arrays. For supporting multiplication operation for higher precision operands (8-bit, 16-bit etc.), BCE decomposes the operands into smaller operands with 4-bit precision and accumulate



Operand	3_d	5_d	7_d	9_d	11_d	13_d	15_d
3_d	9_d	15_d	21_d	27_d	33_d	39_d	45_d
5_d	15_d	25_d	35_d	45_d	55_d	65_d	75_d
7_d	21_d	35_d	49_d	63_d	77_d	91_d	105_d
9_d	27_d	45_d	63_d	81_d	99_d	117_d	135_d
11_d	33_d	55_d	77_d	99_d	121_d	143_d	165_d
13_d	39_d	65_d	91_d	117_d	143_d	169_d	195_d
15_d	45_d	75_d	105_d	135_d	165_d	195_d	225_d

Fig. 5. Execution steps of BCE when both operands are odd with reduced LUT entries for multiply operation

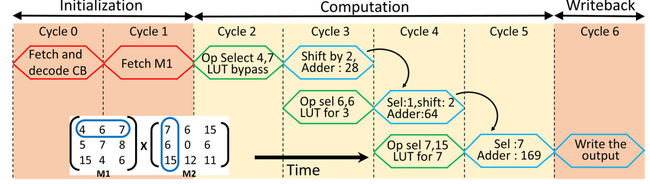


Fig. 6. BCE execution pipeline showing initialization, computation and writeback

the partial products accordingly. For a LUT supporting 4-bit operands, the maximum number of entries required for multiplication operation is 256. However, we reduce the number of LUT entries to just 49 by utilizing fundamental multiplication properties illustrated in [17]. Preloaded LUT entries are shown in Fig.5. We store the products in LUT only if both the operands are odd numbers. If either of the operands are powers of two, then BCE shifts the other operand before adding it to the partial result. If the operands are both odd numbers, then BCE directly fetches its product from the LUT. If both the numbers are even but non-powers of two, then BCE decomposes it into multiples of odd number and powers of two, and appropriately shift the partial product based on the odd part fetched from the LUT before adding it to the partial result. These decisions are made by the operand analyzer within BCE logic (described in Section 3.1). Hence, decomposing the operation between the LUT and the operand analyzer will ensure fewer LUT entries. LUT entries can be further reduced by half, by storing only the upper or lower triangle entries but this will lead to reduced PIM parallelism.

Fig.6 illustrates the detailed execution steps for an example matrix multiplication operation with 4-bit operands. In cycle 0, BCE reads the contents of CB and decodes the address of first row of M1 along with the matrix dimensions. In cycle 1, first column of M2 is streamed in (input streaming are explained detailed in Section IV) from the external bus and first row of M1 is read from sub-array using the generated addresses onto the BCE registers. In the next three cycles, BCE performs three multiplication and two addition operations to generate the first element of the output matrix. Since M1 data (“4” in this case) is in powers of 2, we do not access the LUT in cycle 3 but perform left shifting for multiplication. In cycle 4, two left shift operations are performed since the input even number is split into two powers-of-two numbers. LUT is accessed only in cycle 5 since both inputs are odd numbers. ‘Output will be written back in cycle 6. This pipeline continues until the end of complete matrix multiplication. Since initialization is performed only once during the beginning, computation cycles are proportional to number of multiplications with a small overhead of reading the operands. Even for higher bit-width operands (8-bit, 16-bit), the BCE decomposes the operands into 4-bit operands for the multiplication operation and accumulates the partials in a similar pipelined manner. Further, we show an optimized matrix multiply execution which increases the number of MAC (8-bit operands) operations per cycle from 0.5 to 4.

Optimizing BCE for matrix multiplications: The neural

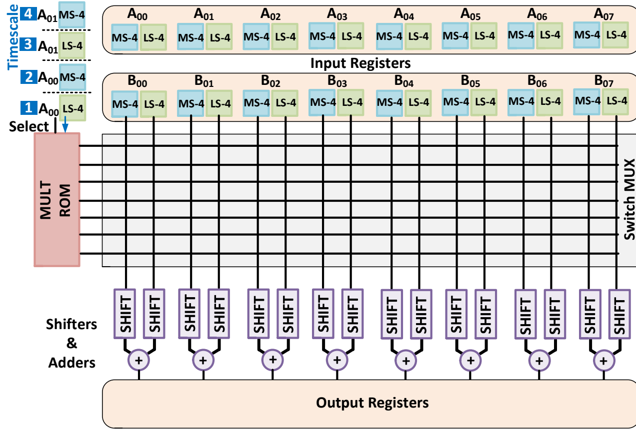


Fig. 7. BCE optimized for matrix multiply, increasing number of multiply operations by 8 times

network workloads are dominated by the convolution or matrix multiply operations. Therefore, increasing the number of multiplications per cycle leads to improved performance. Every access of the LUT for the selected operand utilizes only a byte of data, thereby under-utilizing the LUT read operation. In the case of matrix multiply, we can take advantage of the LUT read by scheduling the inputs appropriately. Fig.7 shows the optimized BCE unit with efficient input scheduling. The input registers hold the first rows of matrices A and B. At timescale 1, the lower nibble (LS-4) of A_{00} selects the appropriate LUT from the hardwired multiply-LUT (MULT ROM in Fig.7) in the BCE. This enables all the outputs with respect to the B matrix (by use of the switch MUX comprising of 16x{8:1 MUX} and I/O is of a byte length) stored in the input register and accumulates the values into the output registers. In the next timescale, the upper nibble (MS-4) of A_{00} performs the same sequence of operations, thereby achieving 8 multiplications in 2 cycles. The subsequent row of matrix B is loaded into the input register. LS-4 and MS-4 of A_{01} are fed at timescales 3 and 4 respectively. The hardwired multiply-LUT in the BCE reduces the bus traffic and achieves seamless data movement within the memory. In addition, the intermediate values generated are stored in the reduced access cost rows of sub-arrays to further minimize the access time and energy.

2) Division: Pooling operations in a neural network are generally used for down-sampling. Max/Min pooling are easily supported with the help of adder block in the BCE. The average pooling involves calculating the average across the patches of input map based on the filters. This operation requires accumulating all the entries of input matrix and dividing it by the total number of entries. For the division operation, we adopt the LUT-based approach proposed in the [22]. It uses Taylor's series expansion of the operands for faster division operation, reduced LUT entries and performs the division operation using the Equation 1. X and Y are the division operands represented with 2m bits. Y_h and Y_l are the upper and lower m-bits. The input operand values are mapped to $[1,2)$ using the shift operations (the shift counter value is stored to re-map the final result). Next, $X(Y_h - Y_l)$ and $1/Y_h^2$

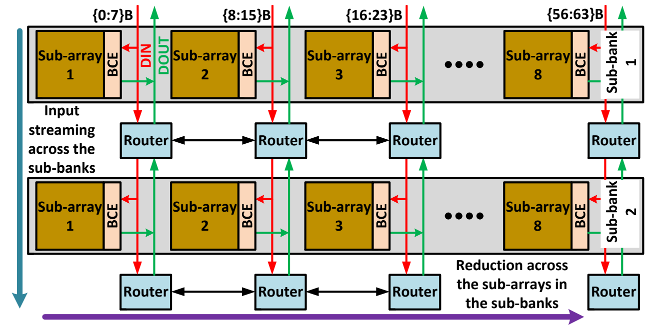


Fig. 8. Routers introduced into slice for systolic flow between sub-arrays

values are calculated concurrently using the LUTs and adder logic in BCE. Finally, the computed result is re-mapped to the original data range using the shift operations.

$$\frac{X}{Y} \approx \frac{X(Y_h - Y_l)}{Y_h^2}; \quad X, Y \in [1,2) \tag{1}$$

3) Activation Functions: The exponent, sigmoid and tanh operations are implemented using LUT based on the piecewise linear approximation, proposed in [23], [24], [25], [26], [27]. The piecewise linear approximation for the exponent operation is shown in the Equation. 2, where S is the number of piecewise segments (for sigmoid and tanh, replace y_l^s with appropriate function).

$$\begin{aligned} f^s(x) = \alpha^s * (x - x_l^s) + y_l^s = \alpha^s * x + (y_l^s - \alpha^s * x_l^s) \\ x \in [x_l^s, x_r^s], y_l^s = e^{x_l^s}, s \in [1,S] \end{aligned} \tag{2}$$

LUT stores the values of α^s and $(y_l^s - \alpha^s * x_l^s)$ corresponding to x_l^s. Similarly, exponential operation in softmax is also computed using the LUT approach and the results are accumulated for further normalization operation. The normalized operation (involves division) is performed using the above illustrated division operation.

D. BFree Systolic Dataflow Within the Slice

For incorporating the systolic dataflow within the slice, we augment the conventional cache sub-array level interconnect with simple routers. Conventional interconnect consists of data-in, data-out, and address bus from the port to every sub-array. For any memory access, all the sub-arrays in a particular sub-bank (determined by the address) will be activated since the data is stripped across the sub-bank (data bus (DIN/DOUT) in Fig.8). Conventional interconnect in conjunction with BCE supports data connectivity between the sub-arrays in the same column, for example, sub-array 1 of sub-bank 1 shares connectivity to sub-array 1 of sub-bank 2. Whereas, the routers are used to provide connectivity between sub-arrays in the same sub-bank as shown in Fig.8. The interconnects are unidirectional; hence router connects data-in of a sub-array to data-out of the neighbouring sub-array. The data during the systolic operations are stored within the registers in the BCE to allow seamless dataflow. The reduction of the partial products are across the sub-arrays local to the sub-bank, whereas the

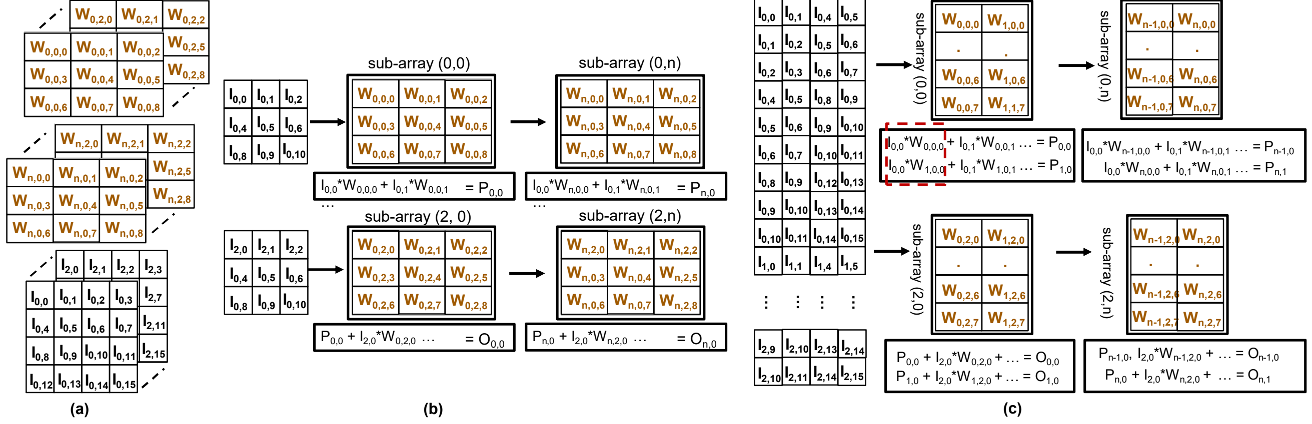


Fig. 9. (a) Weight filters and input activations (b) Systolic mapping of naive convolution operation (c) Systolic mapping of convolution operation transformed into matrix multiplication. Input matrix is in its transposed form. Note that the dimensions of the activations and filters, and the number of sub-arrays are for illustrative purposes only.

input streaming are across the sub-banks as shown in Fig.8. The systolic data movement within the slice is completely uniform at any given time, thus easily managed by the BCEs in conjunction with controllers (discussed in section 4.3).

IV. Neural Network Mapping and Execution

In this section, we discuss various mapping and execution strategies to support the execution of diverse DNN workloads on the BFree architecture. The network parameters such as weight filter size, stride, and input size have significant impact on the overall data movement and available parallelism in execution. For accelerating the convolution operation, it can be formulated as matrix-matrix multiplication to leverage the operand re-use capabilities. However,in the matrix-matrix multiplication formulation, the input vectors to each layer need to be unrolled, resulting in large storage requirements. If there is enough space to store all the unrolled intermediate features (which will be fed as inputs to the next layer), it is beneficial to adopt matrix formulation, otherwise, we perform conventional convolution operation. BFree incorporates efficient support for both conventional convolution and matrix-matrix multiplication operations. In addition, BFree also supports various non linear functions such as Sigmoid, Tanh, and Softmax to support other (other than CNN) widely used networks such as recurrent neural networks and transformer models.

A. Convolution Mapping

Consider the filter and input activations shown in Fig.9(a). An element in weight tensor $W_{n,i,j}$ corresponds to j^{th} element in the ith input channel of nth filter. $I_{i,j}$ is the jth element of ith channel of input activation. A convolution operation is essentially scanning the filter across input activations and performing dot product between the corresponding elements at each position. Fig.9(b) shows the systolic dataflow and mapping of weight filters for naive convolution operation. To setup the systolic dataflow for convolution operation, we distribute the weight tensors as follows: different filters are distributed across the columns of sub-arrays and the input channels of a filter are distributed across the rows of sub-arrays within a column. For instance, in Fig.9(a) there are n different filters with three input channels each. Input channel-1 of filter-n is loaded onto the sub-array at 1st row of nth column. Now at every step, one set of input activations from channel-i are supplied to all the sub-arrays in row-i. The BCE performs dot product between these two vector of elements and forwards the partial product to the adjacent sub-array in vertical direction. The partial products get accumulated at every sub-array within a column to form one element in the output feature map. Essentially, each column produces one element of output feature map at every step.

B. Matrix Multiply Mapping

Convolution operations can be converted into a 2D matrix-matrix multiplication as discussed in [28]. This is accomplished by transforming the filters with dimensions (n,i,j) into a matrix of dimensions $(n*i,j)$. Since the weight matrix is read only during the inference phase, it is statically unrolled into the matrix format. Similarly the input vector has to be transformed into a 2D matrix to perform matrix-matrix multiplication. However, transforming the input feature map is not as trivial as transforming the weight matrix. Every row of the transformed matrix contains the elements that get multiplied with a filter in one convolution step. This corresponds to a matrix with number of rows equal to the number of convolution steps per filter, which is dependent of the stride and size of the filter. Consequently, there could be redundant copies of elements based on the stride between two convolution steps, leading to wasted memory space as shown in Fig.9(c). Note that the input activation matrix shown in the figure is in its transposed form. Since the input activation for intermediate layers are dynamically generated, the matrix formation has to be dynamic as well. This is handled by the slice controller which writes the results in appropriate format in the storage.

The transformed weight matrix is partitioned into sub-matrices and mapped onto the sub-arrays as show in Fig.9(c). Every element (i,j) in input activation matrix gets multiplied

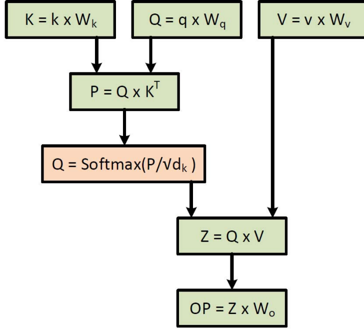


Fig. 10. Self-attention layer in BERT

with all the elements of a row i in the weight matrix. For example, the input $I_{0,0}$ gets multiplied with $W_{0,0,0}, W_{1,0,0}, ...W_{n,0,0}$ to generate the partial products of elements $O_{0,0}, O_{1,0}, ...O_{n,0}$ respectively. Since we perform matrix multiplication in this format, the full compute capability of the BCE unit, as discussed in Section III-C1, is exploited.

1) Mapping Recurrent Neural Networks: Recurrent neural networks process a sequence of inputs using the internal state through a series of gating mechanisms and activation functions. Unlike CNNs, RNNs process the inputs sequentially, meaning to process the input at current step, output of previous step is required. LSTMs and GRUs are the widely used RNN variants which differ in the gating mechanism used. Typically LSTM employ more number of gates than GRUs which makes it computationally intensive. Therefore we use the LSTM [29] network to evaluate BFree architecture. The operations on LSTM network comprises of matrix multiplication, tanh, sigmoid, and softmax operations. The BFree architecture performs all the operations sequentially within the cache, crediting to the re-configurable LUT based operations.

2) Mapping Transformer Models: The primary operation in transformer models is matrix-matrix multiplication which is highly optimized on CPUs and GPUs. Unlike LSTMs, transformer model has abundant parallelism as all the inputs of a sequence can be processed in parallel. Bidirectional encoder representation from transformers (BERT) [30] is an instance of the transformer model which is used in the evaluation of BFree architecture. Fig.10 shows the dataflow of the self-attention layer, which is the basic building block of the BERT model. BFree executes softmax operation similar to matrix multiply using efficient systolic dataflow. Each sub-array processes unique sets of elements in the vector, and accumulates across the sub-array to get denominator of the softmax ($\sum e^x$) operation in the last sub-array. This denominator is redistributed to all the sub-arrays (increased parallelism) for computing the final output. Matrices K, Q, and V can be processed in parallel. However, matrices K and Q are required for further computation of P and P' matrices whereas V is not required until P' is computed. So, we overlap the computation of V with the computation of P' which only involves scalar and softmax units. This scheduling improves the utilization of the compute resources in the system.

C. BFree Execution Flow

BFree incorporates hierarchical control mechanism across cache, slice, bank and sub-array granularity for the execution of DNN workloads. BFree requires new instructions for supporting these in-memory operations (convolution, matrix multiplication, pooling, and activation functions), similar to the state-of-the-art processing in cache works [9]. These in-memory instructions are directed to the cache controller, and it executes the kernel. Each instruction executes a kernel, thus performing layer by layer execution of the NN workloads. Overall BFree execution flow is illustrated in Fig.11. In the configuration phase, depending on the kernel, BFree cache controller loads the LUT rows at sub-arrays with appropriate entries and also programs the slice controller with the necessary control data for the kernel. Next, it loads the weight parameters of the network into all the slices in a broadcast fashion. Depending on the kernel parameters (dimensions of filter, number of channels etc.), it distributes the weights across and within each slice for efficient execution. It employs weight duplication, and efficient partition across sub-arrays to increase the parallelism. Slice controller, loads the configuration blocks (CB) of each BCE with the appropriate metadata depending on the kernel operation.

In the computation phase, cache controller loads the input features/operands onto the input registers of the BCEs of the first sub-bank using slice controller (these inputs will fed to the adjacent sub-banks in a systolic manner in the subsequent cycles). Next, BCE performs the corresponding operation (multiplication, convolution) using the LUT entries in sub-array or its hardwired multiply-ROM. It also accumulates the partial products from the adjacent sub-arrays in a systolic fashion. The final product will be accumulated in the last sub-array in each sub-bank. Depending on the operation, the accumulated products will be further distributed across different sub-arrays in the same sub-bank depending on the output channel. The final results will be either stored in the sub-arrays for processing the next layer features or stored back in the next level storage (such as DRAM) depending on the storage requirements (for example, when we do batch inference, the output features are stored back in the DRAM due to cache storage space constraints).

V. Analysis and Results

This section describes in detail, the design framework, circuit and layout design feasibility to memory system level quantitative evaluations. We also describe strategies and optimizations used in running various types of neural networks, and compare our results with relevant bitline computing techniques. We perform comprehensive evaluation of the SRAM circuit design, layout design and area overhead, BCE and router design.

A. Design and Simulation Environment Setup

As part of the BFree work, we have designed the SRAM bitcells and the subarray partition using circuit design with

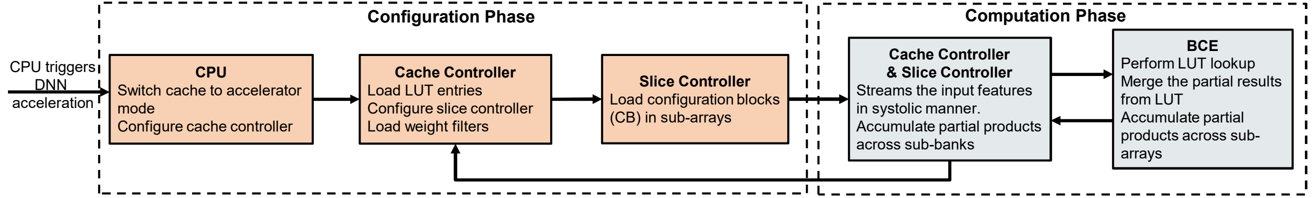


Fig. 11. Overview of BFree execution flow

standard SRAM design techniques. A comprehensive design and analysis requires system level evaluations with various toolsets. Table-1 shows the circuit simulation parameters and tools used for design and evaluation. SRAM circuit, LUT integration and layout design are implemented using TSMC 16nm circuit simulation and layout design rules. We have synthesized and performed auto place and route for a slice with BCEs using Synposys DC and ICC compiler respectively. Next, we evaluated power of the BCE for different operations using Synopsys Primetime-PX. We have incorporated these performance and power metrics into BFree cycle accurate simulator for end-to-end evaluation of various NN workloads (steps shown in Section 3 & 4).

B. Design Analysis

BFree incorporates minimal changes to conventional L3 cache to support PIM LUT based compute, this subsection analyzes the optimized modes, power, performance and area overheads of the additional logics:

Reduced access cost for LUT: The reduced cost access rows for the LUT look-up are used for all operations except multiply. During the convolution or matrix multiply operation, these reduced cost access rows are used for the intermediate partial products. Sub-array requires two additional components for enabling LUT feature into the subarray partition. These are LUT precharge and LUT enable circuitry which will enable the Bitline decoupling to the rows of subarray partition. Bitline connectivity to only few rows and reuse of the already existing subarray peripherals will result in faster data lookup. This additional circuit constitutes to 0.5% area compared to the sub-array.

Controller: The controller at various granularity (cache: 0.8mW, slice: 1.4mW) orchestrating the dataflow constitutes to 0.1% area compared to the whole L3 cache.

BCE: BCE in convolution mode (conv_mode) consumes 0.4mW, which utilizes 1x{8:1 MUX}, 1xAdder, 2xShifters. Whereas in matrix multiplication mode (matmul_mode) utilizes the switch MUX (8x{8:1 MUX}), all the Adders and Shifters, thereby consuming 1.3mW. For all other operations, BCE consumes 0.4mW. By integrating 16nm technology parameters into CACTI [31], we calculate area overhead for larger cache slices. The BCE area overhead is 6% for a cache slice of 2.5MB. Increased access latency due to BCE area overhead is countered by repeated up sizing which aids in balancing the delay overheads.

Comparison with Specialized MAC unit: Having a specialized MAC at the subarray level might be an alternative for bitline computing. While this design is straightforward, BCE when compared to specialized MAC unit with equivalent configurable features, occupies 3% lesser area and offers 48% more energy efficiency. The BCE design eliminates costly multipliers, that accounts for the energy savings compared to the MAC unit. Also, simple 8-bit MAC unit does not support special operations like sigmoid, tanh, softmax, etc., whereas BCE and LUT together supports them.

C. Experimental Setup

In the subsequent section, we first compare our BFree PIM approach with a recent processing in cache technique - Neural Cache [9]. For this purpose, we run inception-v3 [32]. Even though we give emphasis towards PIM techniques for caches, we also compare BFree with NN accelerator like Eyeriss [33] using VGG-16 [34]. We further point out that DRAM bandwidth is the performance bottleneck of BFree and present the performance scaling for increasing DRAM bandwidth for one of the networks. We also discuss the mapping of LSTM and BERT, to demonstrate the re-configurability of the BFree architecture.

We use Intel Xeon CPU E5-2697 running at 2.6 GHz for CPU baseline performance. For GPU performance, we use NVIDIA Titan-V GPU having 5120 cores, 8.5MB shared cache and 12 GB HBM2. We evaluate BFree with 35MB L3 cache over different networks with varied parameters, shown in Table-II. We profiled CPU and GPU using Pytorch and Tensorflow profiler. To measure the power, we use Intel Rapl [35] Nvidia-smi [36] tools for CPU and GPU respectively. The maximum frequency for BFree is same as the subarray access latency (1.5GHz).

TABLE I
DESIGN TOOLS ANALYSIS SETUP FRAMEWORKS

SRAM, LUT partition, and BCE design	16nm SPICE simulation model and standard cell libraries
Cache level evaluation	CACTI [31]
Synthesis	Synopsys Design Compiler
Power and Timing analysis	Synopsys Primetime-PX

TABLE II
SUMMARY OF NEURAL NETWORK WORKLOADS

Network	Layers	Params	Mults	Dataset
Inception-v3	48	24M	4.7G	ImageNet [37]
VGG-16	16	138M	15.5G	
LSTM	1	4.3M	4.35M	TIMIT [38]
BERT-base	12	87M	11.1G	MRPC [39]
BERT-large	24	324M	39.5G	

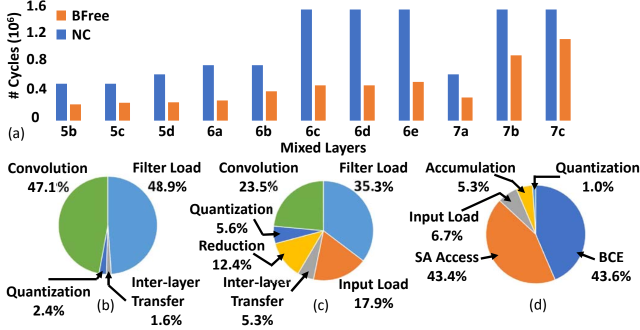


Fig. 12. (a)Layer-wise run-time comparison of mixed layers in Inception-v3. Latency breakdown of (b)BFree, and (c)Neural Cache. (d)Energy breakdown of different operations in the Cache for BFree excluding DRAM energy.

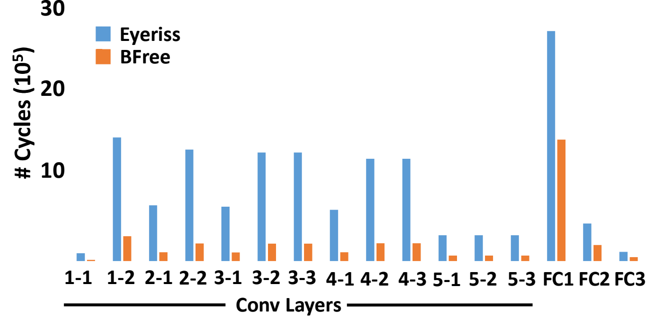


Fig. 13. Layerwise latency comparison against Eyeriss

D. Evaluation

Comparison with Neural Cache: Compared to Neural Cache [9], BFree on conv_mode (0.5 MAC/cycle per sub-array) shows a 1.72x overall speedup and 3.14x energy savings for the same L3 cache size of 35MB. BFree requires minimal perturbation to the sub-array, hence running at sub-array's frequency. Whereas, Neural Cache adds additional peripheral logics in the sub-array to perform computations, thereby decreasing the sub-array's frequency. The speedup advantage is also due to the systolic dataflow processing incorporated into our design. On the other hand, Neural Cache loads all inputs into the appropriate subarrays before the processing can begin. Furthermore, the outputs available on different bitlines have to be read out and written back multiple times for accumulation. These overheads are hidden in BFree as the inputs and partial sum flow across the systolic array which is pipelined.

Fig.12(a) shows the performance of BFree and Neural Cache when running mixed layers of Inception-V3. Since input loading is not a contributing factor to the runtime of BFree, it performs significantly better for layers with larger input sizes.

Fig.12(b)&(c) shows the distribution of runtime for Inception-V3 on BFree and Neural Cache respectively. Majority of the runtime for both the architectures is spent in loading filter elements from DRAM. The next major component is the convolution phase which primarily comprises of MAC operations. During the convolution phase, both BFree and Neural Cache have a similar performance. However, Neural Cache spends almost 30% of the execution time in loading inputs and reducing the partial products obtained during convolution phase. But, BFree does not have these overheads due to systolic data processing. For quantization, we use gemmlowp [40] technique which requires multiplying a scaling factor and adding a bias to the feature before shifting it to obtain the final value. This is performed by all the subarrays hosting the data, eliminating the round trip to the processor. Hence, we see a 1.72x speedup compared to Neural Cache.

The energy savings can be attributed to the reduced number of subarray reads and writes, and low power design of BCE units. Neural Cache has to activate all the bitlines every cycle either for read, write or MAC operations. The energy cost for the read/write access, and compute operation is 8.6pJ and 15.4pJ respectively. In contrast, BCE accesses the subarray only for reading and writing data, whereas the MAC operations are performed using the BCE hardwired-LUT(consumes about 0.5pJ) resulting the significant energy gains over Neural Cache.

For BFree, almost 80% of the energy is attributed to the weight loading phase from DRAM. Fig.12(d) shows the energy distribution after excluding the DRAM energy. From this figure, we can observe that the sub-array access energy (SA access) and BCE contribute to 85% of the energy. SA access is incurred while reading and writing weight elements and partial products. BCE energy is the total energy consumed by BCE during the execution which includes the accumulation operation across sub-arrays.

Comparison with Eyeriss: VGG16 network comprises of huge weight filter sizes, thereby enabling matrix multiplication based dataflow (discussed in section 4.2). Hence, BCE in matmul_mode (4 MACs/cycle per sub-array) can utilize all the 16 adders & shifters when performing matrix multiplication. The filter matrix can be loaded onto the SRAM subarray in the appropriate format. The input activation matrix is dynamically generated by issuing multiple reads to the DRAM buffers.

We compare the performance of running VGG-16 against Eyeriss [33] with iso-compute area and iso-frequency for a slice (SRAM memory with 2.5MB). We get an area overhead of 6% for the proposed configuration of BCE units in a 2.5MB slice. So, we configured Eyeriss to the same area as the additional custom logic needed for BFree. The equivalent configuration of Eyeriss accelerator consists of 12x12 array with 8-bit MAC units (area of the PE of Eyeriss is scaled to 16nm technology). Fig.13 shows the layer wise computation cycle breakdown and BFree is 3.97x faster than Eyeriss. BFree requires reduced number of computation cycles compared to Eyeriss. Execution of layers in BFree is dominated by weight and input loading time rather than the execution ($\sim 10\%$). The overhead due to weight loading can be amortized by performing batch processing. However, the input load overhead cannot be hidden by batch processing which becomes the bottleneck of the system. It is due to the limited main-memory bandwidth available in the system.

Increasing the main memory bandwidth: In Fig.14, we show the trend in execution time when we increase the avail-

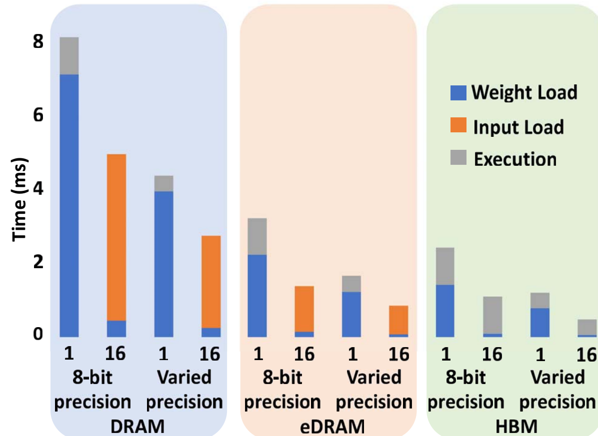


Fig. 14. Latency breakdown for VGG16 network with varied main memory bandwidth (DRAM: 20GBps, eDRAM: 64GBps, and HBM: 100GBps).

able bandwidth using eDRAMs [41] and HBM [42] memories for batch sizes 1 and 16. For a single image processing (batch is 1), we store the intermediates within the SRAM, but in the case of batch processing, BFree uses the next level memory to hold the intermediates, thereby suffering from input load time. We could observe that the eDRAM still suffers from the feature loading time, whereas with HBM the BFree is highly efficient without much loading overheads. Also, the Fig.14 shows the performance benefits due to varying bit precisions among the layers of VGG-16. BFree seamlessly supports operations on different bitwidths, enabling it to exploit the layerwise varied precision trained using [43]. Varied bit-precision (accuracy loss of 1%) reduces the 50% of execution time compared to the 8-bit precision, since most of the layers are executed using 4-bit precision. While running a single batch, inputs are loaded from the SRAM but input load time increases with higher number batches, since it is stored in next level memory. This trend is valid with varied inputs and weight bit precision.

Comparison with CPU and GPU: BFree achieves 259x, 5.5x speedup and 307x, 11.8x energy savings w.r.t CPU and GPU for a batch size of 16 when running Inception-V3. For VGG-16, we get a speedup of 193x, 3x, and, energy savings of 253x and 7x w.r.t CPU and GPU for a batch size of 16.

LSTM and BERT: The core computation in LSTM and BERT are matrix-vector and matrix-matrix multiplications, so BFree is operated in matmul_mode Also, these networks make use of special functions like tanh, sigmoid, softmax, etc, which can be computed with the help of LUTs.

The whole LSTM model fits within the SRAM cache. Therefore, the weight load overhead is amortized over the sequence of inputs. BFree executes LSTM-1024 network using all the slices.

Table-III shows the runtime of LSTM networks of baseline and BFree architecture for a sequence length of 300. The runtime presented for baseline architectures is only for the computation involved in a timestep during steady state of

TABLE III
RUNTIME & ENERGY COMPARISON OF BFREE W.R.T CPU & GPU FOR LSTM, BERT-BASE AND BERT-LARGE.

Net-work	Ba-tch	Execution Time in ms			Energy in J		
		CPU	GPU	BFree	CPU	GPU	BFree
LSTM		888.3	96.2	0.43	31.09	4.33	0.01
BERT -base	1	1160.0	47.3	5.3	34.80	1.67	0.12
	16	121.3	3.8	1.2	3.64	0.45	0.04
BERT -large	1	2910.0	89.7	35.6	87.3	4.5	0.39
	16	453.1	11.1	6.7	13.6	1.7	0.12

execution. The runtime excludes the data transfer time from main memory as weights will be available in cache. Because LSTM is a sequential model, the data movement overheads in general purpose processors cannot be hidden as much from the runtime.

Table-III also contains the performance of baselines and BFree for BERT-base and BERT-large models. We observe relatively lesser speed-ups for BFree compared to the baseline models, since CPUs and GPUs use highly optimized BLAS routines [44], [45] for matrix multiplication which achieves good resource utilization. The speedup w.r.t GPU (5120 cores) can be attributed to large number of compute units in BFree (4 MACs/subarray, and a total of 4480 sub-arrays) and energy savings are due to reduced data movement overheads. CPU, GPU and BFree all benefit from batching of inputs by amortizing the data movement overheads. BERT-base has smaller layers compared to BERT-large and therefore has more replicas of the layer. In such scenarios, different inputs have to be supplied to the replicas to perform useful work, which is again limited by the available main memory bandwidth. Hence the impact of main memory bandwidth on runtime is significant for BERT-base than for BERT-large.

VI. RELATED WORKS

In this section, we discuss existing custom digital accelerator and PIM-based accelerator architectures for DNN acceleration. Several neural network accelerators have been proposed to leverage the computational and energy efficiency offered by dedicated hardware. Eyeriss [33] optimizes data movements by maximizing input data reuse and minimizing partial sum accumulation cost. DaDianNao [46] proposes a compact neural network supercomputer. They map specialized logic of the DNNs to multiple chips/nodes which are tightly interconnected to optimize the data transfer. Simba [47] maps the DNN inference operation onto multiple smaller chiplets in a distributed fashion. WAX [48] incorporates the compute units adjacent to each SRAM scratch pad sub-arrays to minimize the wire lengths between compute units and storage. SCNN [49] and SparTen [50] propose sparse neural network accelerators. They customize the dataflow to leverage the weight and activation sparsity in the neural networks for faster inference and improved energy efficiency.

PIM-based accelerators can be broadly categorized into two categories depending on the underlying memory technology. First category of PIM accelerators [3], [5], [51], [52], [53]

leverage the in-situ analog and digital MAC capabilities of emerging cross-point memory architectures to accelerate the DNN workloads. These architectures augment the cross-point memory arrays with necessary peripheral circuitry such as analog-to-digital and digital-to-analog converters, shift & add units. Second category of PIM accelerators [9], [54], [55], [56] leverage the SRAM and DRAM based bitline computing techniques. Aga et al. [57] proposed compute caches which repurpose the cache for computational purposes. Boolean operations are performed using multi row activation technique. Slower compute time as compared to the data access time as a measure to retain the memory robustness. This Bitline computing technique accelerates applications having bulk Boolean operations. Works from Zhang et al. [19], [58] have proposed multi row activation in SRAM for in memory linear classification by performing multiplications using bitline current summation technique.

Neural Cache [9] and Duality cache [54] run neural networks on repurposed caches of the processor using multi row activation and bit serial computing. Since caches occupy majority of the processor real estate and bitline level computing can transform the memory into thousands of PIM accelerators, both Neural Cache and Duality Cache achieve performance much higher than CPU and GPU running the same application. However fundamental operations like MAC require multiple access to the bitlines. All of the above works either design a novel SRAM cell, use new technology or modify the memory peripherals to achieve PIM at the bitline level. PIM solutions are not just limited to SRAMs but works from V. Seshadri et al., [55], [56] perform similar multi row activations in DRAM. However, our work relies on LUTs within the subarray hierarchy of the memory and compute neural network primitives in lesser cycles. We show higher energy efficiency than the related works which reply on the bitline discharge for computation. To the best of our knowledge ours is the first work to propose PIM using light weight LUTs within the SRAM subarray and reduce the role of bitlines in the computation. We show performance boost and higher energy efficiency by computing neural network primitives in lesser number of cycles.

VII. Conclusion

Technology downscaling challenges have negatively impacted the memory design. Over the years, design strategy for memories have been capacity centric and two thirds of modern-day processor area are occupied by on chip caches. Data movement in and out of the caches have become the major bottleneck in executing neural network tasks. PIM solutions have been able to alleviate this bottleneck for present day applications. PIM using bitline discharge for computations have been well researched and demonstrate performance and energy efficiency. However accessing the bitlines repeatedly along with modifying the SRAM cell and peripherals have limited the maximum achievable gains. In this regard we propose a bitline computing free PIM solution (BFree) by configuring a small portion of the sub-array as LUTs. Segregation of the bitlines and the presence of small compute engine (BCE) configures the sub-array to perform neural network computation. In this work we have shown that BFree supports various kinds of neural networks. This PIM solution achieves 1.72x better performance while being 3.14x energy efficient compared to the state-of-the-art DNN in-memory accelerators when running inception v3. Our analysis show 101x, 3x speed up and 91x, 11x energy efficient than CPU and GPU respectively for a transformer model, BERT-Base. BFree has the potential to further unlock more efficient PIM capabilities with better mapping techniques and task sharing in a tightly coupled compute-memory system.

Acknowledgements

We thank the anonymous reviewers of ISCA 2020 and MICRO 2020 for their constructive and insightful comments.

References

[1] Jay B Brockman, Peter M Kogge, Thomas L Sterling, Vincent W Freeh, and Shannon K Kuntz. Microservers: A new memory semantics for massively parallel computing. In *Proceedings of the 13th international conference on Supercomputing*, pages 454–463, 1999.

[2] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, Jaewook Shin, and Joonseok Park. Mapping irregular applications to diva, a pim-based data-intensive architecture. In *SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, pages 57–57, Nov 1999.

[3] 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 *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, June 2016.

[4] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 236–249, New York, NY, USA, 2019. ACM.

[5] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 14–26, Piscataway, NJ, USA, 2016. IEEE Press.

[6] A. Biswas and A. P. Chandrakasan. Conv-ram: An energy-efficient sram with embedded convolution computation for low-power cnn-based machine learning applications. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 488–490, Feb 2018.

[7] W. Khwa, J. Chen, J. Li, X. Si, E. Yang, X. Sun, R. Liu, P. Chen, Q. Li, S. Yu, and M. Chang. A 65nm 4kb algorithm-dependent computing-in-memory sram unit-macro with 2.3ns and 55.8tops/w fully parallel product-sum operation for binary dnn edge processors. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 496–498, Feb 2018.

[8] S. Kumar, V. A. Tikkiwal, and H. Gupta. Read snm free sram cell design in deep submicron technology. In *2013 INTERNATIONAL CONFERENCE ON SIGNAL PROCESSING AND COMMUNICATION (ICSC)*, pages 375–380, Dec 2013.

[9] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 383–396, Piscataway, NJ, USA, 2018. IEEE Press.

[10] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester. A 28-nm compute sram with bit-serial logic/arithmetic operations for programmable in-memory vector computing. *IEEE Journal of Solid-State Circuits*, 55(1):76–86, Jan 2020.

[11] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits*, 51(4):1009–1021, 2016.

[12] A. S. Noetzel. An interpolating memory unit for function evaluation: analysis and design. *IEEE Transactions on Computers*, 38(3):377–384, 1989.

[13] H. Ling. An approach to implementing multiplication with small tables. *IEEE Transactions on Computers*, 39(5):717–718, 1990.

[14] P. T. P. Tang. Table-lookup algorithms for elementary functions and their error analysis. Argonne National Lab. Tech. Report MCS-P194-1190, 1991.

[15] S. S. Prasad and S. K. Sanyal. Design of arbitrary waveform generator based on direct digital synthesis technique using code composer studio platform. In *2007 International Symposium on Signals, Circuits and Systems*, volume 1, pages 1–4, 2007.

[16] P. K. Meher. New approach to look-up-table design and memory-based realization of fir digital filter. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(3):592–603, 2010.

[17] P. K. Meher. Lut optimization for memory-based computation. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 57(4):285–289, April 2010.

[18] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. LUTNet: Rethinking inference in FPGA soft logic. In *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2019.

[19] J. Zhang, Z. Wang, and N. Verma. In-memory computation of a machine-learning classifier in a standard 6t sram array. *IEEE Journal of Solid-State Circuits*, 52(4):915–924, April 2017.

[20] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492, Feb 2017.

[21] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[22] P. Hung, H. Fahmy, O. Mencer, and M. J. Flynn. Fast division algorithm with a small lookup table. In *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers (Cat. No.CH37020)*, volume 2, pages 1465–1468 vol.2, 1999.

[23] H. Amin, K. M. Curtis, and B. R. Hayes-Gill. Piecewise linear approximation applied to nonlinear function of a neural network. *IEE Proceedings - Circuits, Devices and Systems*, 144(6):313–317, 1997.

[24] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 243–254. IEEE Press, 2016.

[25] Y. Bengio and J. Senecal. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, 19(4):713–722, 2008.

[26] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

[27] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$1mb model size. *CoRR*, abs/1602.07360, 2016.

[28] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. Circnn: Accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 395–408, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH*, pages 338–342, 2014.

[30] Kenton Lee Kristina Toutanova Jacob Devlin, Ming-Wei Chang. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[31] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 3–14, Dec 2007.

[32] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, June 2016.

[33] Y. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, June 2016.

[34] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[35] Intel rapl power meter. https://01.org/rapl-power-meter.

[36] Nvidia-smi. https://developer.nvidia.com/nvidia-system-management-interface.

[37] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[38] Garofolo, john s., et al. timit acoustic-phonetic continuous speech corpus ldc93s1. web download. philadelphia: Linguistic data consortium, 1993.

[39] Microsoft research paraphrase corpus. https://www.microsoft.com/en-us/download/details.aspx?id=52398.

[40] Gemmlowp: a small self-contained low-precision gemm library. https://github.com/google/gemmlowp.

[41] F. Hamzaoglu, U. Arslan, N. Bisnik, S. Ghosh, M. B. Lal, N. Lindert, M. Meterelliyoz, R. B. Osborne, J. Park, S. Tomishima, Y. Wang, and K. Zhang. A 1 gb 2 ghz 128 gb/s bandwidth embedded dram in 22 nm tri-gate cmos technology. *IEEE Journal of Solid-State Circuits*, 50(1):150–157, 2015.

[42] D. U. Lee, K. W. Kim, K. W. Kim, K. S. Lee, S. J. Byeon, J. H. Kim, J. H. Cho, J. Lee, and J. H. Chun. A 1.2 v 8 gb 8-channel 128 gb/s high-bandwidth memory (hbm) stacked dram with effective i/o test circuits. *IEEE Journal of Solid-State Circuits*, 50(1):191–203, 2015.

[43] Md Fahim Faysal Khan, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Vijaykrishnan Narayanan. Learning to quantize deep neural networks: A competitive-collaborative approach. In *Proceedings of the 57th Annual Design Automation Conference*, 2020.

[44] mklblas. https://software.intel.com/en-us/mkl.

[45] https://docs.nvidia.com/cuda/cublas/index.

[46] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, Dec 2014.

[47] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Sumanth Gudaparthi, Surya Narayanan, Rajeev Balasubramonian, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. Wire-aware architecture and dataflow for cnn accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1–13, New York, NY, USA, 2019. Association for Computing Machinery.

[49] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40, 2017.

[50] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. Sparten: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 151–165, New York, NY, USA, 2019. Association for Computing Machinery.

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

[52] T. Yang, H. Cheng, C. Yang, I. Tseng, H. Hu, H. Chang, and H. Li. Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks. In *2019 ACM/IEEE 46th Annual*

International Symposium on Computer Architecture (ISCA), pages 236–249, 2019.

[53] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 802–815, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. Duality cache for data parallel acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 397–410, New York, NY, USA, 2019. ACM.

[55] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 273–287, New York, NY, USA, 2017. ACM.

[56] In-dram bulk bitwise execution engine, arxiv:1905.09822.

[57] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492, Feb 2017.

[58] Jintao Zhang, Zhuo Wang, and N. Verma. A machine-learning classifier implemented in a standard 6t sram array. In *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages 1–2, June 2016.