FELIX: Fast and Energy-Efficient Logic in Memory

Saransh Gupta, Mohsen Imani, and Tajana Rosing
CSE Department, UC San Diego, La Jolla, CA 92093, USA
{sgupta, moimani, tajana}@ucsd.edu

ABSTRACT

The Internet of Things (IoT) has led to the emergence of big data. Processing this amount of data poses a challenge for current computing systems. PIM enables in-place computation which reduces data movement, a major latency bottleneck in conventional systems. In this paper, we propose an in-memory implementation of fast and energy-efficient logic (FELIX) which combines the functionality of PIM with memories. To the best of authors' knowledge, FELIX is the first PIM logic to enable the single cycle NOR, NOT, NAND, minority, and OR directly in crossbar memory. We exploit the voltage threshold-based memristors to enable single cycle operations. It is a purely in-memory execution which neither reads out data nor changes sense amplifiers, while preserving data in-memory. We extend these single cycle operations to implement more complex functions like XOR and addition in memory with 2× lower latency than the fastest published PIM technique. We also increase the amount of in-memory parallelism in our design by segmenting bitlines using switches. To evaluate the efficiency of our design at the system level, we design a FELIX-based HyperDimensional (HD) computing accelerator. Our evaluation shows that for all applications tested using HD, FELIX provides on average 128.8× speedup and 5,589.3× lower energy consumption as compared to AMD GPU. FELIX HD also achieves on average 2.21× higher energy efficiency, 1.86× speedup, and 1.68× less memory as compared to the fastest PIM technique.

CCS CONCEPTS

• **Hardware** → **Emerging technologies**; *Integrated circuits*; • **Computing methodologies** → *Machine learning*;

KEYWORDS

Processing in-Memory, Non-volatile memories, Memristors, Hyperdimensional computing, Machine learning, Energy efficiency

ACM Reference Format:
Saransh Gupta, Mohsen Imani, and Tajana Rosing. 2018. FELIX: Fast and Energy-Efficient Logic in Memory. In *IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD '18), November 5–8, 2018, San Diego, CA, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3240765.3240811

1 INTRODUCTION

The unprecedented increase in the number of interconnected devices and smart systems has resulted in the concept of big data. To process this data, it is either run on a multi-core system or sent to cloud and run on large servers [1, 2]. Both of these involve large energy and latency overheads. This is caused by the large amount of data movement between processing units and memory storing the data, owing to limited cache capacity and on-chip bandwidth [3, 4]. The processing in-memory (PIM) tries to address this issue by processing part of data in-place, eliminating the need to transfer all data to the processing unit.

PIM has recently become an active area of research. Most of it is driven by the emergence of new non-volatile memory technologies, like memristors. The logic state of memristors depends upon the resistance of the device, which is controlled by the charge through it. They have fast switching speed, low switching energy, and high scalability, making them suitable for dense and fast PIM solutions [5–8].

A number of recent publications have exploited memristors to enable PIM [9–15]. Some compute logic at the periphery of the memory by modifying the memory sense amplifiers [16, 17]. They read the stored data from the memory, use transistor based circuits to process data, and store the results back to the memory. In these designs, the amount of data that can be processed in parallel is limited by the amount of sense circuitry present at the periphery of the memory. Another work exploits the bipolar switching behaviour of memristors to implement logic in-memory [9–11, 18–20]. These designs depend on application of voltage at various memory cells with no change in the sense amplifiers. They are purely in-memory operations which do not need to read out data but are restricted by the limited functionality they can implement. For example, MAGIC [9] only supports NOR directly in crossbar memory. All other functions are implemented by repeated multiple NOR cycles.

In this paper, we propose a purely in-memory implementation of fast and energy efficient logic (FELIX). Our design extends the functionality of in-memory operations by implementing *single cycle* NOR, NOT, NAND, minority (Min), and OR directly in crossbar memory. We use these low latency functions to implement functions like XOR and addition 2× faster than MAGIC [9]. Our design further increases the amount of in-memory parallelism by using in-block switches, which segment the bitlines to make parallel operations independent of each other. We demonstrate the efficiency of our design by developing a state-of-the-art accelerator for HyperDimensional (HD) computing.

Our architecture fully processes HD encoding inside a crossbar memory, without using any computing units. Our evaluation shows that for all applications tested on HD, FELIX provides on an average 128.8× speedup and 5589.3× lower energy consumption as compared to GPU. FELIX HD also achieves on an average 2.21× higher energy efficiency, 1.86× speedup, and 1.68× lower memory requirement as compared to the fastest PIM technique.

2 RELATED WORK & BACKGROUND

Processing in-memory (PIM) accelerates computation by reducing the overhead of data movement and providing high parallelism in some cases [4, 21]. Early PIM designs use high performance CMOS logic alongside memory blocks. Some of the designs added simple processing units at the periphery of the main memory while others changed the sense amplifiers to support basic functions in

ICCAD '18, November 5–8, 2018, San Diego, CA, USA

ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240811

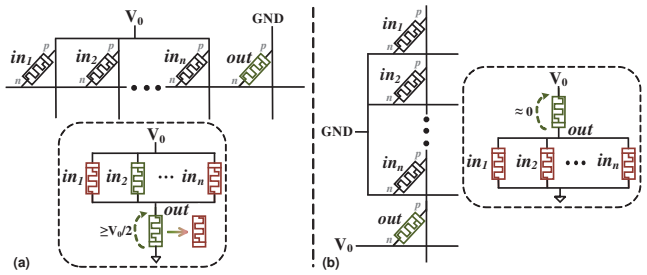

Figure 1: n-input NOR implementation in (a) a row and (b) a column.

memory. However, to implement more complex functions, dedicated CMOS-based processing cores were added, making the manufacturing process complicated and costly.

High density, low-power consumption, fast switching, and CMOS-compatibility of emerging non-volatile memories (NVMs), make them promising candidates for low-power main memory. They have also been shown to have computing abilities [5–8]. Many logic families have been proposed for computation inside memristive crossbar. Some of them implement logic purely in memory such as stateful implication logic [18, 22], and Memristor Aided loGIC (MAGIC) [9]. Logic execution with MAGIC is fully compatible with the usual crossbar design, requires a lower number of voltages, and supports NOR which can be used to implement any Boolean logic. Also, it is non-destructive, unlike implication logic based designs like IMPLY [18] which destroy one of the inputs. These properties of MAGIC make it a preferred logic family for resistive crossbar memories. MAGIC uses voltage threshold based memristors which switch whenever the voltage difference between the two terminals, p and n shown in Figure 1, exceeds a threshold. However, they don't fully utilize the threshold based switching of memristors and only implement NOR in crossbar memory. All other functions are derived using NOR, which results in unnecessary latency overheads.

On the contrary, we propose FELIX in this work which enables single cycle operation of many Boolean functions in crossbar memory. This significantly reduces the latency of in-memory logic execution while providing higher memory utilization. The result is 2× faster execution of a whole machine learning application as compared to MAGIC [11]. We also improve parallelism in FELIX by using in-block switches to increase the performance of PIM.

3 BASIC OPERATIONS IN FELIX

FELIX is a purely in-memory implementation of Boolean logic. It executes NOR, NOT, Min, NAND, and OR in a single cycle in crossbar memory. FELIX uses a variable voltage based execution scheme, where the applied voltage defines the operation to be performed. In addition, instead of relying only on resetting behavior of memristors, we exploit it's two-way switching to extend the capabilities of PIM. When the voltage $V_{pn} > |v_{on}|$, a memristor switches from a high resistive state (R_{OFF}) to a low resistive state (R_{ON}). On the other hand, when $V_{np} > v_{off}$, it switches from R_{ON} to R_{OFF}. Here, $|v_{on}|$ and v_{off} are the device dependent voltage thresholds and V_{pn} is the voltage difference between terminals p and n.

Table 1 compares the execution of different boolean logic functions in FELIX with previously proposed PIM techniques. The latencies in the table and the discussion exclude the first initialization cycle which is common to both designs. The numbers in brackets represent the properties of the area conservative designs [11]. It shows that FELIX performs either the same as or significantly better

Table 1: Comparison of FELIX with state-of-the-art PIM technique designed for highest performance.

Property	Design	NOR3	NAND3	Min3	OR3	Maj3	AND3	XOR2	1-bit ADD
Latency (Cycles)	MAGIC	1	5 (6)	5 (6)	2	4	4	5 (7)	12 (14)
	FELIX	1	1	1	1	2	2	2	6
	Improv.	1×	5 (6)×	5 (6)×	2×	2×	2×	2.5 (3.5)×	2 (2.33)×
Memory (# of Cells)	MAGIC	1	5 (4)	5 (4)	2	4	4	5 (3)	12 (6)
	FELIX	1	1	1	1	2	2	1	4
	Improv.	1×	5 (4)×	5 (4)×	2×	2×	2×	5 (3)×	3 (1.5)×
Energy (fJ)	MAGIC	24.11	120.17	120.38	48.12	96.17	96.15	120.29	288.82
	FELIX	24.11	49.24	41.64	9.53	65.65	73.26	34.97	135.60
	Improv.	1×	2.44×	2.89×	5.05×	1.47×	1.31×	3.44×	2.13×

than the fastest state-of-the-art technique [9, 11]. For example, for addition, FELIX is 2× faster, has 2× better energy efficiency, and 3× lower memory size. In the following subsections, we outline the implementation of basic boolean operations in FELIX.

3.1 Single-Cycle Operations

NOR: Figure 1 shows how NOR is implemented in memristor-based crossbar array [9]. The output memristor is initialized to R_{ON} in the beginning. To execute NOR in a row, an execution voltage, V_0, is applied at the p terminals of the inputs and the p terminal of the output memristor is grounded, as shown in Figure 1a. When NOR is executed in a column, the n terminals of the inputs are grounded while V_0 is applied to the n terminal of the output, as shown in Figure 1b. The motive behind both the executions is to switch the output memristor from R_{ON} to R_{OFF} whenever the NOR output is '0'. FELIX is as fast as MAGIC [9] for NOR.

NAND and Min: FELIX does not directly depend upon the inputs but the voltage developed across the n and p terminals of the output device. This enables it to implement minority and NAND in memory. Consider the case of a 3-input NOR using a V_0 of 1V and v_{off} of 0.5V. The voltage developed across the output memristor is equal to 0V, 0.5V, 0.67V, and 0.75V when the inputs are '000,' '001,' '011,' and '111' respectively. Here, the output memristor switches in all cases except the first one. Now, if V_0 is changed to 0.75V, developed across the output memristor changes to 0V, 0.38V, 0.5V, and 0.56V when the inputs are '000,' '001,' '011,' and '111' respectively. In this case, the output switches to R_{OFF} only when there are at least two '1's in the input. The output is effectively the 3-bit minority function (Min3). As V_0 is further decreased to 0.67V, output changes only when the inputs are '111.' In other words, the output is '0' only when all the inputs are '1.' This is equivalent to a 3-bit NAND operation. The above logic can be extended to N-bit minority and NAND functions. The execution voltage, V_0, required to implement these functions is given by Equation 1, where N is the number of inputs.

$$\frac{v_{off}}{R_{ON}} \cdot \{R_{ON} + \left(\frac{R_{OFF}}{N-n+1}\right) || \left(\frac{R_{ON}}{n+1}\right)\} < V_0 < \frac{v_{off}}{R_{ON}} \cdot \{R_{ON} + \left(\frac{R_{OFF}}{N-n}\right) || \left(\frac{R_{ON}}{n}\right)\}, \quad (1a)$$

$$\left(\frac{n+2}{n+1}\right) \cdot v_{off} < V_0 < \left(\frac{n+1}{n}\right) \cdot v_{off}, \quad (1b)$$

The value of n is defined by the operation to be executed. For Min, $n = \lceil N2 \rceil$ and for NAND, $n = N$. Equation 1b is an approximation of Equation 1a under the assumption that $R_{OFF} >> R_{ON}$.

Hence, in addition to NOR and NOT, FELIX supports a single cycle MinN and NAND. In theory this technique can be extended for any n. However, the non-availability of different voltage levels challenges its practical feasibility for large values of n. For example, FELIX requires a V_0 of 0.58V to implement a 6-bit NAND. It changes to 0.57V and 0.56V in case of a 7-bit and 8-bit NAND respectively. It is difficult to reliably generate these different and

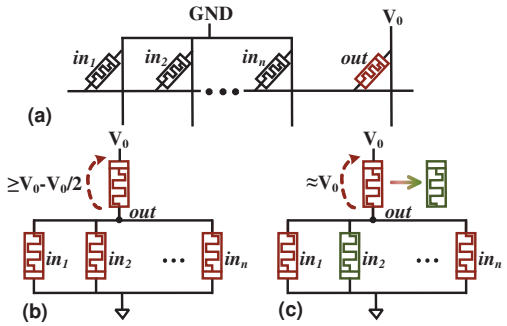

Figure 2: Voltage division for FELIX OR. (a) Application of voltage for OR, (b) output memristor remains R_{OFF} when all the inputs are R_{OFF} (red), and (c) output memristor switches to R_{ON} when one or more inputs are R_{ON} (green).

closely valued levels of voltages. Hence, to keep the implementations practical, we restrict FELIX to 2-bit and 3-bit NAND and Min.

OR: FELIX reduces the latency of OR operation to one cycle by exploiting the setting behavior of the memristor device. As discussed in Section 2, a device can be switched from R_{OFF} to R_{ON} by applying a voltage greater than the threshold, $|v_{on}|$. On the other hand, since MAGIC relies just on the resetting behavior of memristors, implementing OR in crossbar memory using MAGIC involves two NOR cycles. Figure 2 shows the voltages division for different possible inputs. Ground and V_0 terminals are opposite of those used for MinN. When all the input and output memristors are R_{OFF}, the voltage across the output memristor is much less than V_0. On the other hand, if one or more inputs are R_{ON}, the voltage across the output is approximately V_0. If V_0 is greater than v_{on}, then the output memristor switches to R_{ON}.

The above behavior is exploited to implement OR in memristive memory. The output memristor is first initialized to R_{OFF}. To execute OR in a row, the p terminals of the input memristors are grounded while V_0 is applied at the p terminal of the output. In case of OR in a column, V_0 is applied at the n terminals of the inputs the n terminal of the output is grounded (show p and n terminals in a figure). If the logical 'high' and 'low' states are represented by R_{ON} and R_{OFF} states of memristor, the result of OR operation corresponds to R_{OFF} when all the input bits are low and R_{ON} otherwise. The execution voltage, V_0, required to implement OR is given by,

$$\frac{|v_{on}|}{R_{OFF}} \cdot \{R_{OFF} + (R_{ON}) \,||\, \left(\frac{R_{OFF}}{N-1}\right)\} < V_0 < \frac{|v_{on}|}{R_{OFF}} \cdot \{R_{OFF} + \left(\frac{R_{OFF}}{N}\right)\}, \quad (2a)$$

$$\left(1 + \frac{R_{ON}}{R_{OFF}}\right) \cdot |v_{on}| < V_0 < \left(\frac{N+1}{N}\right) \cdot |v_{on}|, \quad (2b)$$

where N is the number of inputs. Equation 2b is an approximation of Equation 2a under the assumption that $R_{OFF} >> R_{ON}$.

3.2 Multi-Cycle Operations

The in-memory operations proposed in the above section can be combined to extend the functionality of the memory.

Maj and AND: Majority (MajN) and AND can be implemented by inverting MinN and NAND respectively. This results in 2-cycle MajN and AND in FELIX in contrast to four cycles in MAGIC.

XOR: XOR ($\oplus$) can be expressed in terms of OR (+), AND (.), and NAND ($.'$) as follows:

$$A \oplus B = A + B.A.B' \quad (3)$$

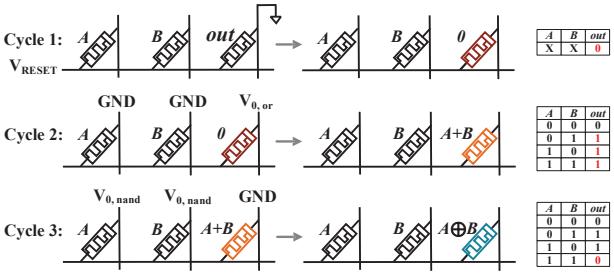

Figure 3: Stages in implementing 2-bit XOR using FELIX.

Figure 3 shows the in-memory implementation of Equation 3. Instead of calculating OR and NAND separately and then ANDing them, we first calculate OR and then use its output cell to implement NAND. In this way, we eliminate separate execution of AND operation. This logic just requires 2 FELIX cycles and one additional memristor device, which also acts as the output cell. In contrast, the state-of-the-art PIM technique proposed in [11] uses 5 cycles and 5 memristors for the fastest XOR implementation, while the most area conservative approach takes 7 cycles and 3 memristors. Hence, the proposed XOR implementation is both faster and smaller.

Addition: FELIX implements addition by combining XOR and MajN operations. A 1-bit adder can be represented by,

$$S = A \oplus B \oplus C_{in}, \quad (4a)$$

$$C_{out} = A.B + B.C + C.A = MajNA, B, C_{in}, \quad (4b)$$

where A, B, and C_{in} are 1-bit inputs while S and C_{out} are the generated sum and carry bits respectively. Here, S is implemented as two serial in-memory XOR operations. C_{out}, on the other hand, can be executed by inverting the output of MinN. Hence, S takes a total of 4 cycles and 2 additional memristors, while C_{out} needs 2 cycles and 2 additional memristors.

The previously proposed state-of-the-art processing in-memory techniques also support addition within the crossbar memory [11, 19]. These approaches break down an operation into a series of NOR operations. A typical addition implementation requires 12 NOR operations, resulting in 12 MAGIC NOR cycles [11] as compared to 6 in FELIX and 12 additional memristors as compared to 4 in FELIX.

4 PARALLELISM IN FELIX

In order to increase in-memory parallelism, we split the array into smaller partitions such that the achievable parallelism directly depends upon the number of partitions of the memory block. These partitions are created by the transistor switches which divide the bitlines into smaller segments. This enables FELIX to independently implement multiple operations simultaneously. Consider a memory array with a 64-bit wordline and capacity of 1024 words. Now, a bitwise OR operation needs to be carried out between 10 pairs of words and outputs be stored in the memory. All these steps are independent of each other and can happen in parallel if the memory supports it. FELIX can execute the same in a single cycle if the memory has 10 or more partitions. In contrast, the conventional design would execute it in 10 steps with each step implementing 64 parallel single cycle OR operations between a pair of words [11].

Figure 4 shows how FELIX behaves in a memory with no partition when operations are parallelized across rows and columns simultaneously. The currents from different operations interfere with each other, as shown by {Op1 Op3} and {Op2 Op4} in Figure 4. It

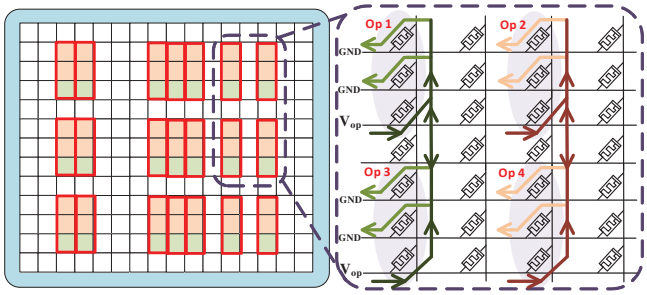

Figure 4: Limitation of the memory in implementing operations in a row and column simultaneously. The crossbar structure does not distinguish the currents from different operations.

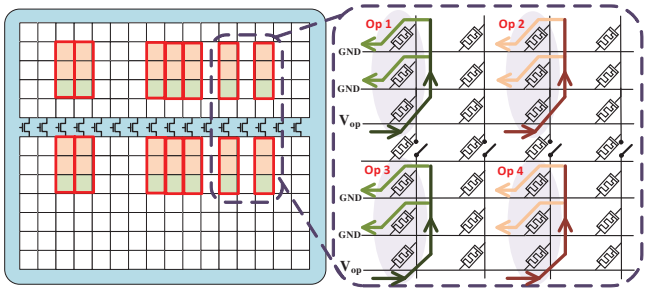

Figure 5: FELIX with transistors and the resultant memory crossbar. The currents for the four operations do not interfere with each other.

effectively results in a single operation with more inputs and multiple copies of output. Transistors physically split the bitlines while keeping them logically the same. Figure 5 shows how FELIX behaves when a memory is divided into two partitions using transistors. The transistors, when switched off, prevent the currents belonging to different operations from merging. This enables FELIX to parallelize operations in rows and columns simultaneously.

Ideally, we would like to have as many partitions as possible. However, increasing the number of partitions comes with additional overhead. First, more partitions lead to reduced memory utilization. FELIX requires some additional devices or processing elements for executing logic. These elements store the intermediate states involved in achieving the final output. Since each partition needs its own processing elements, increasing the number of partitions linearly increases the devices required. These processing elements cannot be used for storing logic because they are used by FELIX to implement every operation. In case of a fixed memory size, increasing the number of processing elements directly reduces the amount of memory usable for storage. Second, more partitions require more number of transistors to segment the bitlines, leading an increased area overhead. Figure 6 shows the change in memory utilization and area overhead for a 1024×64 memory block as the number of partitions increases.

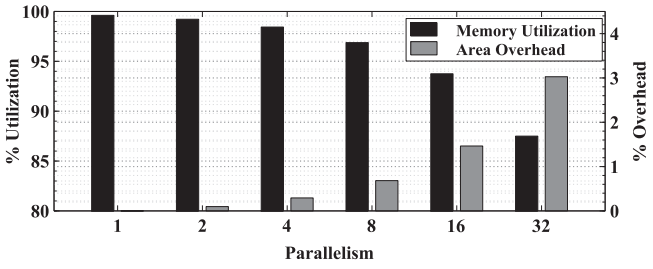

Figure 6: Change in memory utilization and area overhead due to transistors with increase in parallelism in FELIX.

5 HYPERDIMENSIONAL COMPUTING

5.1 HD Classification Overview

Brain-inspired HyperDimensional (HD) computing is a computing paradigm which works based on understanding the fact that brains compute with *patterns of neural activity* [23], where such neural activity patterns can only be modeled with points of high-dimensional space (e.g., D=10,000). Classification is one of the most important supervised learning algorithms. Figure 7a shows the overview of HD computing architecture for a classification problem consisting of an encoder module and an associative memory. The goal of the encoder is to map an input data to a single hypervector with D dimensions and then combine these hypervectors for all of the inputs in a class to generate a unique hypervector representing each class [24–26]. Each class hypervector is a long vector with D dimension, where each dimension can have binary (0, 1) elements. Associative memory stores the trained hypervectors for all classes. In test mode, HD classifies an unknown input by encoding the input image to a hypervector using the same encoder used for training. The query hypervector has binary elements and the same D dimension as the class hypervectors. Next, associative memory checks the similarity of the query hypervector to all classes and classifies it to a class which has the closest similarity.

5.2 Encoding Module

Figure 7b shows the encoding module in HD computing. Let's assume each data point in original space can be represented using a features vector $\{v_1, \ldots, v_n$. The goal of encoding module is to map this feature vector to high-dimensional space while keeping all its information in a high-dimensional vector. Each feature vector stores two types of information: the value of signal and the index of each feature.

Feature values: In order to consider the impact of feature values, our design first identifies the minimum and maximum value that signal can take in all dimensions $\{v_{min}\&v_{max}\}$. Then, it quantizes the feature values into Q levels were v_{min} and v_{max} are the first and last levels respectively. HD assigns a single hypervector with D dimension to each of the quantized levels $\mathbb{L} = \{L_1, L_2, \ldots, L_Q\}$ where $L_i \in \{0,1\}^D$ and L_1 and $L-Q$ correspond to the v_{min} and v_{max} respectively. The generation of the level hypervector is similar to work [24], such that the level hypervectors have similar values if the corresponding original data are closer, while L_1 and L_Q will be nearly orthogonal.

Feature index: To specify the impact of each feature index on encoded hypervector, HD generates a set of random identification hypervector $\{ID_1, \ldots, ID_n\}$, where $ID_i \in \{0,1\}^D$ represents a hypervector corresponding to i^{th} feature index. Due to random generation, the ID hypervectors are semi-orthogonal, meaning that:

$$\delta ID_i,\ ID_j \simeq D2 \quad \text{for } i \neq j$$

Depending on feature values, each feature maps to one of the Q generated hypervectors. Hypervectors are combined together using element-wise XOR of the level and ID hypervector, and then

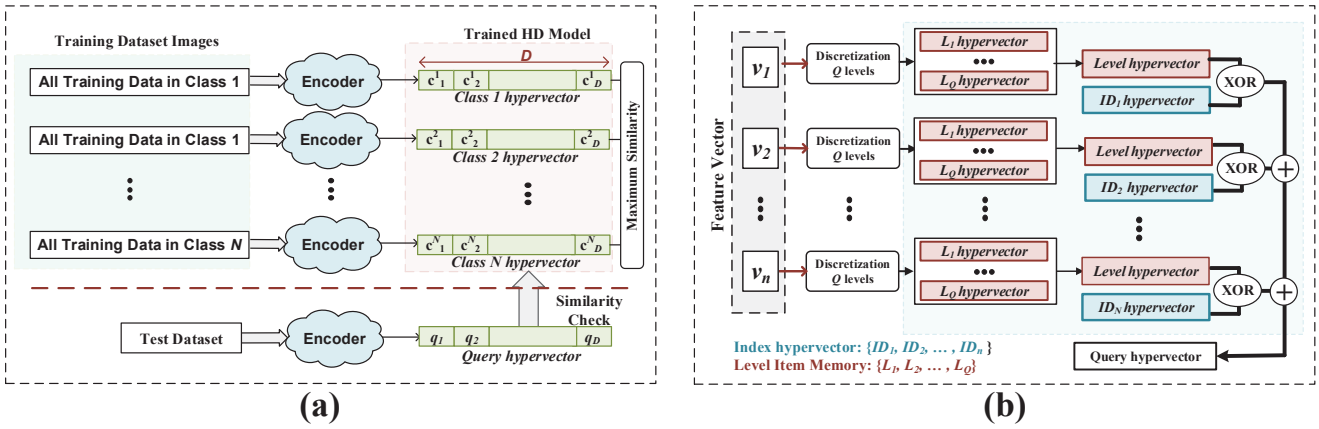

Figure 7: (a) The overview of HD computing architecture for classification task. (b) The encoding module of HD computing mapping a feature vector with n elements to high dimensional space using pre-generated identity and level hypervectors.

summing the resulting hypervectors over all features:

$$I = \overline{L}_1 \oplus ID_1 + \overline{L}_2 \oplus ID_2 + \cdots + \overline{L}_n \oplus ID_n \; f \in 1, m$$

where $\overline{L}_i$ is the (binary) hypervector corresponding to the i-th feature of vector v.

5.3 FELIX-Based HD Acceleration

In this section, we discuss how HD is mapped to memory and its acceleration using FELIX. The HD efficiency depends on the amount of parallelism which we can apply to encoding module. The most area efficient method is to store all ID and level hypervectors in a single memory partition and perform XOR operation between each ID and corresponding level in series. This method serially processes the features and its performance is directly related to the number of features. Our design parallelizes the encoding module by partitioning the memory block.

Let us assume that a feature vector has n elements and Q corresponding Ls (Levels). In HD, this results in n IDs. In total we have $n + Q$ vectors with D=10,000 dimensions. All these vectors are stored in a memory block. Each vector is mapped to a row of the memory, where each row has a capacity of 10,000 bits as shown in Figure 8. In HD encoding, as discussed in the previous section, first each ID is XORed with one of the Ls depending upon the values of the feature. The results of the XOR operations are then added together dimensionwise. We map all these operations in FELIX-enabled memory.

We perform n FELIX XORs (one for each ID) and generate n outputs. For the first n iterations, we select one ID and XOR it with one of the Ls in every iteration. For a pair of ID and L, FELIX XOR can be computed in parallel for all dimensions since all dimensions of a vector are stored in the same row. Each XOR operation requires one additional memory cell to store the output of the result. This requires 10,000 additional cells, equivalent to a row of the memory, to store the output. We then count the number of 1s in all XOR results for each dimension. We execute this by FELIX addition. In order to perform addition for all the dimensions in parallel, we store the output of addition vertically in a column, instead of a row, as shown in Figure 8. We add n elements serially, three bits at a time. If $X_1, X_2, \ldots X_n$ are the vectors to be added together, we first add X_1, X_2, and X_3 to generate S_1 and C_1. We then add X_4, X_5, and $\{C_1, S_1\}$ to generate S_2 and C_2, and so on till we have added all XOR results. The addition of n 1-bit elements results in an output

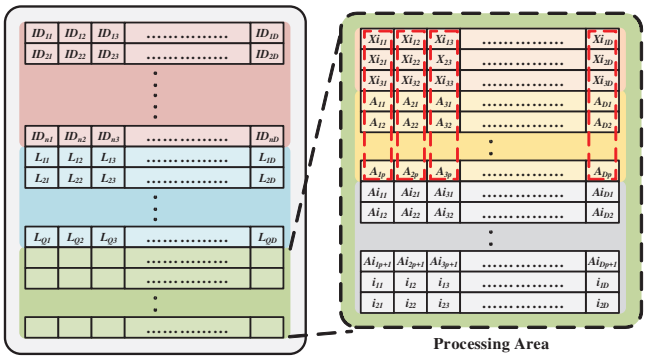

Figure 8: Organization of data in FELIX-enabled HD accelerator.

with $p = \lceil log_2 n \rceil$ bits, requiring p rows to store the output of addition. In addition, FELIX also requires $p+1$ rows to store the intermediate results of addition like C_1, S_1, S_2, C_2, etc.

As described above, we add a maximum of three XOR results at a time. Hence, instead of calculating all XOR results first and then adding them, we calculate them two at a time, except the first step when we calculate three, and add them. This reduces the memory requirement for XOR results from n rows to just 3. Apart from the 3 rows for XOR results and p rows required for the addition results, we require $p+1$ rows to store the intermediate addition results and 2 rows for the intermediate FELIX stages, as shown in Figure 8.

6 RESULTS

6.1 Experimental Setup

We compare FELIX performance and energy efficiency with AMD Radeon R9 390 GPU with 8GB memory and Intel i7 7600 CPU with 16GB memory. For the measurement of system and processor power, we used Hioki 3334 power meter and AMD CodeXL [27]. All software support for application level evaluation including training and testing of HD model have been performed in CPU using C++ implementation. For hardware level evaluation we have designed a cycle accurate simulator which emulates the HD computing functionality at inference. Our simulator pre-store the randomly genrated

Table 2: The energy efficiency, speedup and memory efficiency of FELIX as compared to MAGIC running different applications.

	ISOLET	FACE	UCIHAR	PAMPA
Energy Improv.	2.20×	2.20×	2.21×	2.26×
Speedup	1.86×	1.86×	1.88×	1.87×
Memory Efficiency	1.61×	1.61×	1.61×	1.82×

level and index hypervectors in memory and performs the encoding operations in-memory using the controller signals.

We extract the circuit level characteristic of FELIX performing basic operations and give them as input to simulator. Performance and energy consumption are obtained from circuit level simulations for a 45nm CMOS process technology using Cadence Virtuoso. We use VTEAM memristor model [28] for our memory design simulation with R_{ON} and R_{OFF} of $10k\Omega$ and $10M\Omega$ respectively.

6.2 Workloads

We evaluate the efficiency of the proposed FELIX on four popular classification applications, as listed below:

Speech Recognition (ISOLET): The goal is to recognize voice audio of the 26 letters of the English alphabet. The training and testing datasets are taken from Isolet dataset [29]. This dataset consists of 150 subjects speaking each letter of the alphabet twice. The speakers are grouped into sets of 30 speakers. The training of hypervectors is performed on *ISOLET* 1, 2, 3, 4, and tested on *ISOLET* 5.

Face Recognition (FACE): We exploit Caltech dataset of 10,000 web faces [30]. Negative training images, i.e., non-face images, are selected from CIFAR-100 and Pascal VOS 2012 datasets [31]. We select 10% of images for the testing dataset which are completely separated from the training dataset. For the Histogram of Oriented Gradients (HOG) feature extraction, we divide a 32x32 image to (i) 2x2 regions for three color channels and (ii) 8x8 regions for gray-scale.

Activity Recognition (UCIHAR) [32]: The dataset includes signals collected from motion sensors for 8 subjects performing 19 different activities. The objective is to recognize the class of human activities.

Physical Activity Monitoring (PAMPA) [33]: This dataset includes logs of 8 users and three 3D accelerometers positioned on arm, chest and ankle. They were collected over different human activities such as lying, walking and, ascending stairs, and each of them was corresponded to an activity ID. The goal is to recognize 12 different activities.

6.3 HD Results

Here we compare the efficiency of HD computing with D=10,000 for three different platforms: CPU, GPU, proposed in-memory FELIX architecture and MAGIC. Table 2 compares the energy efficiency, speedup and memory efficiency of FELIX with MAGIC [11] while running different HD classification applications. The memory efficiency is defined as the number of processing cells required to execute in-memory operations. For a fair comparison, we use the proposed architecture to evaluate both FELIX and MAGIC. The results show that FELIX HD can achieve on an average 2.21× higher energy efficiency, 1.86× speedup, and 1.68× lower memory requirement as compared to MAGIC.

Figure 9 shows the energy consumption and execution time of HD for different application on all platforms. Our evaluation shows that for all tested applications, FELIX can provide on average 978.7× and 128.8× speedup and 14,960.3× and 5,589.3× lower energy

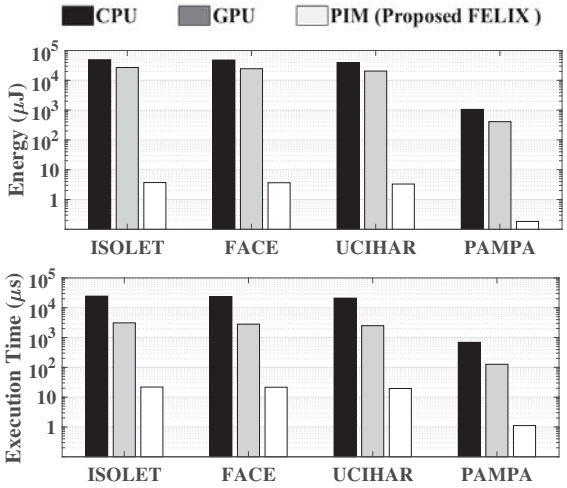

Figure 9: Energy consumption and execution time of the baseline HD on CPU, GPU, and the proposed FELIX-based architecture.

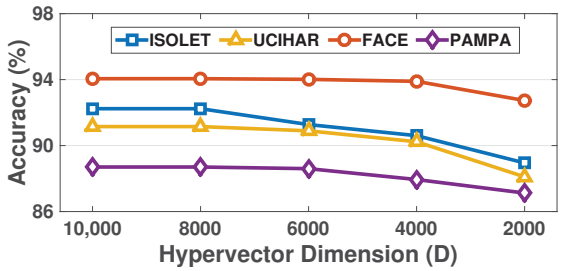

Figure 10: Impact of HD dimension reduction on the applications classification accuracy.

consumption as compared to the CPU and GPU respectively. Here, we reported the result for FELIX with single memory partition. The higher efficiency of the FELIX comes from (i) memory compatible operations of HD which enables FELIX to parallelize the operations in different dimensions (ii) lower data movement and higher locality of the data in-memory for FELIX computation. As we can see in results, the energy and execution time of each application depends on the number of features. Applications with large number of features require more resources for computation.

6.4 FELIX Efficiency-Accuracy Trade-off

HD computing works based on the pattern of neural activity which are in high-dimensional space. In theory, the dimensional of the hypervector should be large enough (e.g., $D = 10,000$) to ensure the randomly generated base hypervectors are nearly orthogonal. However, HD computing shows robustness to scaling the hypervector dimensions. Figure 10 shows the HD accuracy when the hypervector dimension scales from 2000 to 10,000. The result shows that for all applications the HD can provide the same accuracy as 10,000 when the hypervector dimension scales to 8000. In addition, reducing the hypervector dimension to 2000, HD can provide on average 2.3% lower accuracy as compared to baseline HD with full 10,000 dimensions.

Our design exploit the robustness of HD to dimensionality in order to reduce the computation cost. Figure 11 shows the energy

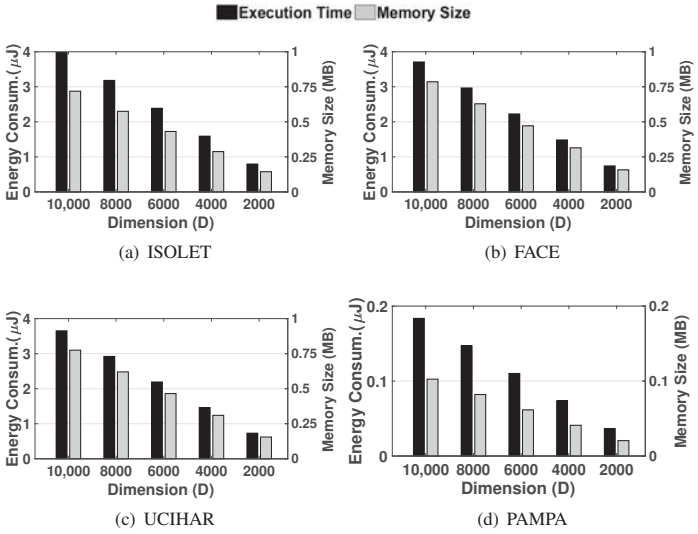

Figure 11: Energy consumption and memory requirement of FELIX running HD in different dimensions.

consumption and memory requirement of HD computing running in-memory architecture. The memory requirement includes the memory requirement to store the base hypervectors and intermediate results during the computation. Our evaluation shows that reducing the hypervector dimension reduces FELIX energy consumption and memory size. This efficiency comes from the less number of class elements and operations that HD needs to store and process in lower dimension. Our result shows that FELIX memory requirement decreases linearly with the hypervector dimensions. For example, HD in D=2000 dimensions consume 76% lower energy consumption, and required 80% smaller memory size as compared to baseline HD with full dimensions. Note that the execution time of FELIX does not change with the hypervector dimensions. In fact, FELIX is designed to perform bit parallel operations where all computations can be parallelized through different dimensions.

There is a trade-off between the accuracy and efficiency when the hypervector dimension reduces. The results are relative to FELIX architecture running the baseline HD with $D = 10,000$ dimensions. The quality loss, ΔE, is defined as the difference between the HD classification accuracy in low dimension and baseline HD. When our design ensures 0.5% quality loss ($\Delta E = 0.5\%$), the FELIX can provide 25% energy efficiency and memory improvement as compared to the baseline HD model. Similarly, ensuring quality loss of less than 1% (2%), FELIX energy and memory efficiency further improve by 65% and 70% receptively.

7 CONCLUSION

In this paper, we proposed an in-memory implementation of fast and energy efficient logic. Our design extends the functionality of in-memory operations by implementing many single cycle operations directly in crossbar memory. We use these low latency functions to implement more complex functions efficiently. Our design further increases the amount of in-memory parallelism by using in-block switches, which segment the bitlines to make parallel operations independent of each other. We demonstrate the efficiency of our design by developing a state-of-the-art accelerator for HD computing. Our evaluation shows that for all tested applications, FELIX can provide on average 128.8× speedup and 5,589.3× lower energy consumption as compared to GPU.

ACKNOWLEDGEMENTS

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034.

REFERENCES

[1] C. Perera *et al.*, "Context aware computing for the internet of things: A survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 414–454, 2014.
[2] J. Gubbi *et al.*, "Internet of things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.
[3] R. Balasubramonian *et al.*, "Near-data processing: Insights from a micro-46 workshop," *Microarchitecture*, vol. 34, no. 4, pp. 36–42, 2014.
[4] G. Loh *et al.*, "A processing-in-memory taxonomy and a case for studying fixed-function pim," in *WoNDP*, 2013.
[5] Q. Guo *et al.*, "Ac-dimm: associative computing with stt-mram," in *ISCA*, vol. 41, pp. 189–200, ACM, 2013.
[6] Q. Guo *et al.*, "A resistive tcam accelerator for data-intensive computing," in *Microarchitecture*, pp. 339–350, ACM, 2011.
[7] M. Imani *et al.*, "Acam: Approximate computing based on adaptive associative memory with online learning," in *IEEE/ACM ISLPED*, pp. 162–167, 2016.
[8] L. Yavits *et al.*, "Resistive associative processor," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 148–151, 2015.
[9] S. Kvatinsky *et al.*, "MAGIC – memristor-aided logic," *TCAS II*, vol. 61, no. 11, pp. 895–899, 2014.
[10] A. Siemon *et al.*, "A complementary resistive switch-based crossbar array adder," *JETCAS*, vol. 5, no. 1, pp. 64–74, 2015.
[11] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.
[12] M. Imani *et al.*, "Genpim: Generalized processing in-memory to accelerate data intensive applications," in *DATE*, IEEE, 2018.
[13] M. Imani *et al.*, "Efficient query processing in crossbar memory," in *ISLPED*, pp. 1–6, IEEE, 2017.
[14] M. Imani *et al.*, "Nvquery: Efficient query processing in non-volatile memory," *IEEE TCAD*, 2018.
[15] M. Imani *et al.*, "Rapidnn: In-memory deep neural network acceleration framework," *arXiv preprint arXiv:1806.05794*, 2018.
[16] S. Li *et al.*, "Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*, p. 173, ACM, 2016.
[17] M. Imani *et al.*, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *IEEE ASP-DAC*, pp. 757–763, IEEE, 2017.
[18] S. Kvatinsky *et al.*, "Memristor-based material implication (IMPLY) logic: design principles and methodologies," *TVLSI*, vol. 22, no. 10, pp. 2054–2066, 2014.
[19] M. Imani *et al.*, "Ultra-efficient processing in-memory for data intensive applications," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 6, ACM, 2017.
[20] A. Haj-Ali *et al.*, "Efficient algorithms for in-memory fixed point multiplication using magic," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, pp. 1–5, IEEE, 2018.
[21] A. M. Aly *et al.*, "M3: Stream processing on main-memory mapreduce," in *ICDE*, pp. 1253–1256, IEEE, 2012.
[22] J. Borghetti *et al.*, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.
[23] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009.
[24] M. Imani *et al.*, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *ICRC*, pp. 1–6, IEEE, 2017.
[25] M. Imani *et al.*, "Low-power sparse hyperdimensional encoder for language recognition," *IEEE Design & Test*, vol. 34, no. 6, pp. 94–101, 2017.
[26] M. Imani *et al.*, "Hierarchical hyperdimensional computing for energy efficient classification," in *DAC*, p. 108, ACM, 2018.
[27] "Amd." http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/.
[28] S. Kvatinsky *et al.*, "Vteam: a general model for voltage-controlled memristors," *TCAS II*, vol. 62, no. 8, pp. 786–790, 2015.
[29] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.
[30] G. Griffin *et al.*, "Caltech-256 object category dataset," 2007.
[31] M. Everingham *et al.*, "The pascal visual object classes challenge: A retrospective," *IJCV*, vol. 111, no. 1, pp. 98–136, 2015.
[32] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities.
[33] A. Reiss and D. Stricker, "Creating and benchmarking a new dataset for physical activity monitoring," in *Proceedings of the 5th International Conference on PErvasive Technologies Related to Assistive Environments*, p. 40, ACM, 2012.