NNgine: Ultra-Efficient Nearest Neighbor Accelerator Based on In-Memory Computing

Mohsen Imani, Yeseong Kim, Tajana Rosing
Department of Computer Science and Engieering, UC San Diego, CA
{moimani, yek048, tajana}@ucsd.edu

***Abstract*—The nearest neighbor (NN) algorithm has been used in a broad range of applications including pattern recognition, classification, computer vision, databases, etc. The NN algorithm tests data points to find the nearest data to a query data point. With the *Internet of Things* the amount of data to search through grows exponentially, so we need to have more efficient NN design. Running NN on multicore processors or on general purpose GPUs has significant energy and performance overhead due to small available cache sizes resulting in moving a lot of data via limited bandwidth busses from memory. In this paper, we propose a nearest neighbor accelerator, called NN*gine*, consisting of ternary content addressable memory (TCAM) blocks which enable near-data computing. The proposed NN*gine* overcomes energy and performance bottleneck of traditional computing systems by utilizing multiple non-volatile TCAMs which search for nearest neighbor data in parallel. We evaluate the efficiency of our NN*gine* design by comparing to existing processor-based approaches. Our results show that NN*gine* can achieve 5590x higher energy efficiency and 510x speed up compared to the state-of-the-art techniques with a negligible accuracy loss of 0.5%.**

Keuwords*—Processing in-memory, Non-volatile memory, Content addressable memory, K-nearest neighbor search

I. INTRODUCTION

Internet of Things (IoT) has dramatically increased the amount of data generated and the size of the application datasets. In 2015, there were more than 25 billion smart devices around the world and this number is expected to double by 2020 [1]. Many IoT applications need to learn and classify the datasets to extract useful information. Massively parallel architectures are used to mask the computation burden of these operations [2], [3]. However, multicore CPUs and general purpose GPUs (GPGPUs) cannot process large datasets efficiently, resulting in high energy consumption and slow processing speed. This inefficiency is the consequence of the large amount of data movement between the memories and processing units due to small cache capacity and limited memory bandwidth [4], [5].

The Nearest Neighbor (NN) search problem arises in numerous fields of applications, including pattern recognition, statistical classification, biology, computer vision, databases, coding theory, computational geometry, etc [6]. The NN computation is highly parallelizable for small reference datasets (e.g., less than tens of megabytes). However, to run the algorithm on a big dataset, the existing general purpose processors suffer from high memory bandwidth and a large number of data movements across memory hierarchy. For example, to process 1 billion candidate points, one query needs 150 GFLOP computation and 500G of data communication [7].

This work was supported by NSF grants 1730158 and 1527034.

The k-Nearest Neighbor (kNN) algorithm is a method to solve the NN search problem [8]. For given ***N*** *reference* data points and ***M*** *queries*, the kNN algorithm finds ***k*** data points which are the nearest neighbors to each of queries. In order to understand how the state-of-the-art kNN algorithm performs for the large dataset on the general purpose processors, we considered a popular GPGPU-based implementation [9]. Figure 1 shows how the performance and cache hit ratio change as a function of different dataset sizes while running on GPGPU. As the dataset size increases, the performance and the memory cache hit ratio significantly decrease. This means that the GPU-based approach cannot deliver either reasonable performance or immediate responses. For example, to handle 128 queries for 1G bytes reference dataset, we observed that the execution time and the amount of energy increase about 5 and 6 times compared to the 512Mbyte case respectively. Figure 2 shows how the system power consumption changes when the algorithm processes a 512MB data set. This approach first computes all distances of each combination of data points and query points, and then selects ***k*** nearest neighbors for every query point using a (partial) sort. While the first step can be parallelized on the GPGPU, the sort procedure cannot leverage such parallelization because it requires lots of data movements and comparisons. Thus, high execution time and energy consumption are inevitable for this GPGPU-based solution.

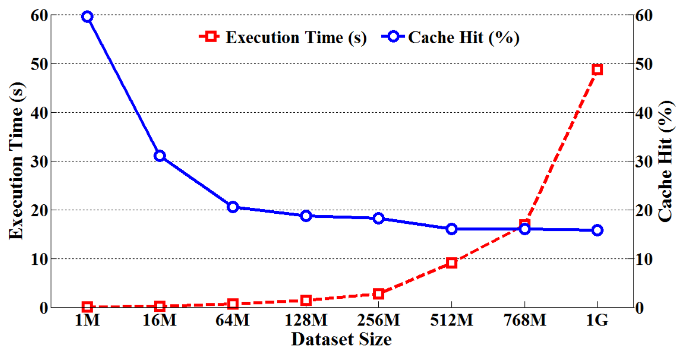


Figure 1. Performance and cache hit rates of GPU-based kNN for different dataset sizes (k=64 and M = 128)

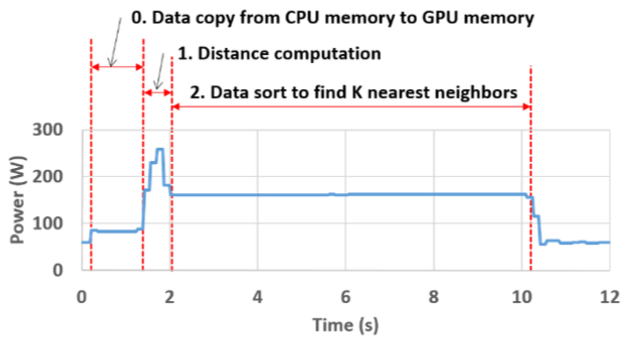


Figure 2. System power consumption of GPU-based kNN for k=64, data set size = 512MB and M = 128

Near-data computing and Processing In-Memory (PIM) are two promising solutions which would overcome this inefficiency [10], [11], [12]. Near data computing utilizes processing units located very close to the main memory, so computation is accelerated by avoiding the memory/cache bandwidth bottleneck [10]. While this idea may achieve higher performance, if a system design needs to add extra dedicated processing units, it may consume quite a bit of additional energy. In this situation, PIM can be a better solution, as it performs computations within memory instead of in processing units [13], [11], [12], [14]. The ternary content addressable memories (TCAM) in the form of lookup tables have been used for memory-based computing. Non-volatile memory (NVM)-based TCAMs allow for energy-efficient search of the entire TCAM block in a single cycle. Thus, in our approach, we use the NVM-based TCAM as a building block for finding the nearest neighbors.

In this paper, we propose high performance energy efficient PIM-based query engine (NN*gine*) which computes nearest neighbor search operation. The proposed NN*gine* stores the input data and parallelizes the data search operation using multiple resistive TCAM blocks. Our design exploits dynamic voltage scaling and analog detector circuitry to find k-nearest data based on hamming distances. The proposed NN*gine* can accelerate many applications which need solutions of the nearest neighbor search problem. We compare the efficiency of our design to existing popular kNN implementations running on the recent general processors, Intel i7 6700K CPU and AMD Radeon R9 390 GPU. Our evaluation shows that the proposed NN*gine* performs with at least 5590**x** higher energy efficiency and 510**x** faster as compared to KNN running on CPU and GPU processors.

II. RELATED WORK

Several approaches have been proposed to improve performance and energy efficiency of the kNN algorithm. A well-known approach is to divide the input data into segments during a precomputation step, and then accelerate multiple search operations with the segments [30]. Other popular approaches exploit GPGPU [15], [16], [9]. They show that the performance speedup is possible by utilizing the parallelism that many small cores provide. However, these kNN computation approaches still incur large energy and performance overhead due to data movements between memory hierarchy and computing cores.

Ternary content addressable memories have been used in several application domains such as associative computing and networking. In CMOS technology, TCAMs are designed with SRAM cells to provide high performance at significant energy cost for each search operations [17]. The high search energy limits the number of available applications to only a few, such as networking and IP routing [18].

High density, CMOS compatibility and nearly-zero energy consumption make NVMs appropriate candidates for the TCAM design [19], [20]. Researchers have proposed diverse energy-efficient techniques that exploit NVM-based TCAMs. Previous work showed that NVM-based TCAMs next to parallel processors such as GPU can provide significant energy savings by reducing redundant computation [21], [22], [23], [24]. Work in [23] utilized the resistive associative memory by GPU floating point units to enable error free program execution. An approximate associative memory using TCAM with voltage overscaling was also proposed to relax FPU computation [21], [22]. This idea has been extended to a configurable approximate associative memory which tunes the level of approximation [23]. Work in [24],[25] used NVM-based TCAM to accelerate neural network on GPU architecture. Although these works showed high energy saving using associative memory, they still rely on the computing capability of the conventional general purpose processors. Thus, with the emerging need of computation for large-volume data, their designs cannot efficiently handle the data movements, which is the dominant component of energy consumption and the performance bottleneck for workload parallelization.

Near-data computation and processing in-memory can accelerate computation by reducing the number of data movements [5], [13], [11], [12]. Work in [5] designed a custom processing unit beside the main memory to accelerate search-based computations. This technique utilizes NVMs to achieve high energy efficiency for storing the data. However, their design does not support searching for nearest value and also suffers from high latency of bit-level serial searches on the additional processing units. The work presented in [12] proposed a novel NVM architecture which can accelerate addition and multiplication inside a memory. The work in [11] proposed a PIM architecture to accelerate bitwise and search operation inside a memory. Work in [13] designed application specific accelerator by enabling PIM functionality in-memory. Although these techniques decrease the amount of data movement, they cannot support algorithm with nearest search capability.

In contrast to previous work, we propose a resistive query accelerator which performs the search computation inside of content addressable memories without the need for extra processing units. The proposed NN*gine* consists of multiple resistive TCAMs which efficiently search for nearest neighbors in parallel. Since the computations run within memory, our design can completely avoid data movement.

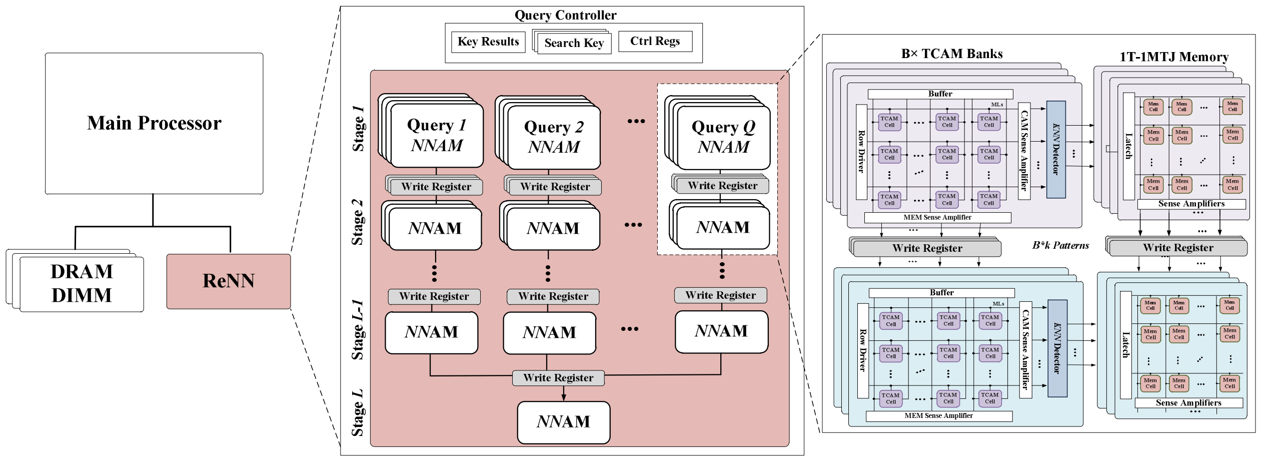


Figure 3. Architecture overview of the proposed NNgine

III. NN*GINE* ACCELERATOR DESIGN

A. NNgine Overview

Today's nearest neighbor algorithms run on multicore CPUs or general purpose GPUs. With large datasets, processors require a number of data movements across memories and computing units. This makes memory bandwidth a significant performance bottleneck, since the datasets often do not fit into cache.

TABLE 1. NN PROBLEM AND DESIGN PARAMETERS OF NNGINE

	Symbols	Description
NN problem-related parameters	$\boldsymbol{k}$	Number of nearest neighbors to search
	$\boldsymbol{N}$	Number of reference data points
	$\boldsymbol{M}$	Number of query data points
NN*gine* design parameters	$\boldsymbol{L}$	Number of subcores in a stage (except for the last stage)
	$\boldsymbol{B_i}$	Number of banks of each subcore in ***i***-th stage
	$\boldsymbol{Size_{NNAM}}$	Number of rows in a NNAM

In this paper, we propose a resistive nearest neighbor query accelerator, called NN*gine*, which can process nearest neighbor searches without the need for data movement. Figure 3 shows the architecture overview of the proposed NN*gine* design. The NN*gine* consists of multiple TCAM blocks which take on both roles of memory and processors, i.e., storing reference data points and finding nearest data points. In our architecture, the reference data point can be directly stored in the NN*gine*. For example, if the original data is stored on the hard disk, the main processor fetches the data to the NN*gine* instead of sending it to DRAM. Once a query operation is initiated by the main processor, the search operation in NN*gine* is processed through multiple stages where the reference data is located in the first stage. Table 1 shows the problem and design parameters of the NN*gine* architecture. Each stage ***i*** includes ***L*** subcores, where each subcore contains $\boldsymbol{B_i}$ banks. Each bank is a TCAM-based associative memory, called NNAM. The NNAM has multiple rows capable of finding ***k*** nearest data (***k***=1, 2, 3, …). It is divided into two parts: TCAM for storing data rows to be searched and a resistive memory for the corresponding ID of each data row of the 1T-1MTJ (1 transistor and 1 magnetic tunneling junction) memory. Once a NNAM finds matched rows in the TCAM, the corresponding ID rows of the resistive memory are activated to return the nearest matched IDs.

As Figure 3 shows, for each query, the NN*gine* searches through subcores of the first stage in parallel. Then, each NNAM corresponding to each bank returns ***k*** rows with the nearest hamming distances for the input query. Since each subcore of the first stage has $\boldsymbol{B_1}$ banks, it yields $\boldsymbol{B_1{*}k}$ outputs which include data and the IDs. These $\boldsymbol{B_1{*}k}$ outputs are written on the NNAM blocks of the second stage using *Write Registers (*Shown in Figure 3*)*. The write registers take the $\boldsymbol{B_1{*}k}$ output rows, and put them into the next $\boldsymbol{B_2}$ banks. This is processed in parallel using each of $\boldsymbol{B_2}$ write registers. This procedure is repeated until the single bank of the last stage is filled. That is, the subcore of the ***i***-th stage finds $\boldsymbol{B_i{*}k}$ outputs, and delivers the found data rows to the next subcore of the ***(i+1)***-th stage using $\boldsymbol{B_{i+1}}$ writer registers. As the final step, The NNAM of the last stage computes ***k*** nearest neighbors among ***L*k*** data points.

The number of NN*gine* banks and processing cores is determined based on the energy and performance requirements. For example, to configure the NN*gine* so that it can handle ***N*** data points and **k** nearest neighbors, we first determine ***L*** and $\boldsymbol{B_1}$ as follows:

$$N = L * B_1 * Size_{NNAM}$$

A large number of NNAM blocks, i.e. ***L*** and $\boldsymbol{B_1}$, improves parallelism at the cost of higher energy consumption. The number of NNAM blocks is also inversely proportional to the NNAM size, $Size_{NNAM}$. The search operation on a large TCAM slows down due to the large buffer size of the TCAM and the long interconnection delay. A block with many rows (>4K rows) requires a large input buffer to distribute the input signal among all rows, making the NNAM inefficient because the buffer energy is a dominant factor in the total energy consumption of the TCAM. We discuss the possible configuration of NN*gine* in Section 4.2. Once ***L,*** $\boldsymbol{B_1}$ ***and***

$\boldsymbol{Size_{NNAM}}$ are chosen, we compute the number of banks of the next stages as follows:

$$B_{i+1} = \lceil k * B_i / Size_{NNAM} \rceil$$

If a number of banks of a stage, $\underline{\boldsymbol{B_j}}$, is 1, then we add the last stage which has a single NNAM, giving us $\boldsymbol{j}$**+1** stages in total.

Figure 4 shows an example of searching for a six bit “000000” query key on NN*gine* with four banks of a subcore. The subcore should find the two nearest data points during a search operation (k=*2*). The search processing is performed in parallel for all NNAM blocks. Each TCAM finds two nearest data points (shown as orange color) which have the minimum hamming distance to the input query. Then, the selected data are written on the second TCAM stage serially using write buffer, and the NNAM of the stage also identifies the two nearest neighbors. Intuitively, the selected two data points in the second stage represent the actual two nearest neighbors of all the original data points of the first stage.

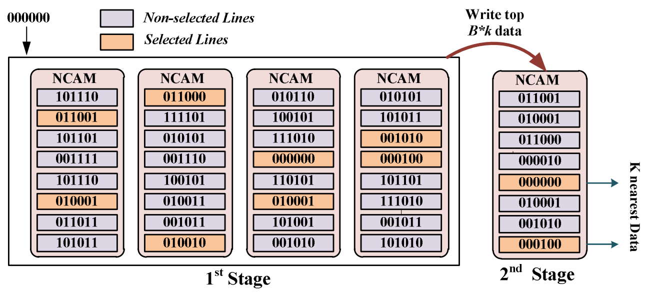


Figure 4. An example of 2-nearest neighbor search on a single subcore of proposed NNgine containing 4 banks

B. NNAM Architecture

The NNAM architecture is designed using a NVM-based TCAM and a resistive memory as illustrated in Section 3.1. Figure 5 shows the architectural overview of the TCAM side of NNAM in detail. Each match line (ML) of the TCAM is connected to multiple cells. Before each search operation, the row driver precharges all MLs to V_{dd}. During the search, if the input query data of the TCAM buffer is matched with the stored data of the TCAM line, the ML is not discharged and the CAM sense amplifier always yields V_{dd}. Otherwise, any mismatched cell discharges its ML, and after a while the sense amplifier yields 0. The number of mismatched cells determine the speed of discharging. Figure 6 shows how the output voltage of a sense amplifier changes over time for different numbers of mismatches. The graph shows the worst case, when we consider 10% process variations on the transistor sizes, and threshold voltages [26]. For example, if we have one mismatch (1-HD) in the TCAM line, the discharging speed of the ML is slower than with two mismatches (2-HD). We can use this timing characteristic to identify nearest hamming distance data, i.e., smallest number of mismatches in a TCAM.

However, there are two technical challenges to directly use these characteristics in the nearest data identification. First, it is hard to dynamically change the clock interval for the sense amplifier to detect the voltage drop timing for each hamming distance. We exploit the fact that applying voltage overscaling (VOS) changes the discharging time for a given hamming distance difference [23]. Figure 6 shows the output voltage using different supply voltages for different numbers of mismatches. As shown in the figure, decreasing the supply voltage changes the discharging time for different hamming distances. Thus, by changing the level of the supply voltage, we can examine the hamming distance. For example, reducing TCAM voltage to 800mV, 780mV and 710mV increases the mismatches to 1, 2 and 3 bits hamming distances so that all of them are discharged at the same time, e.g., 9 ns for 32-bit TCAM. In our case, the error in terms of the hamming distance can be translated to data similarity. In addition, we can leverage the key benefit of VOS, that is energy saving of TCAM search operations.

The second challenge is the nonlinear relationship between ML discharging speed and the number of hamming distances. For example, in a 32-bit TCAM, five and six bits hamming distance have similar ML discharging times. It is because the discharging current saturates for a large number of misses. Thus, to identify hamming distances on a large word size block, we use a TCAM cell whose discharging rate is low due to its large ON resistance. This design keeps the ML voltages stable during search operations, and creates distinct time differences over different hamming distances at the cost of minimal search speed degradation.

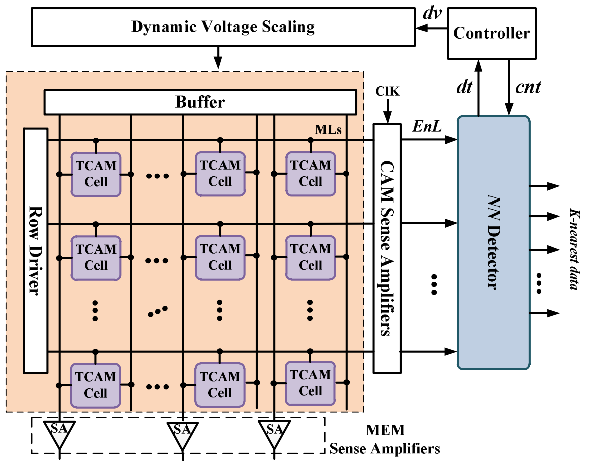


Figure 5. The overview of proposed NNAM architecture

Based on these discharging characteristics of NNAM, in an actual search operation, we first precharge MLs and search for exactly matched rows with the input query data at a nominal supply voltage (1*V*). Then, using dynamic voltage scaling (DVS), we start reducing the voltage of the TCAM to find the $\boldsymbol{k}$ nearest data. At the same time, the detector circuit checks the number of matches on the TCAM output. In case of having $\boldsymbol{k}$ active rows, the detector circuit sends an acknowledgement signal to the controller to stop the DVS, and the controller puts the k activated data to the writer registers. This enables an efficient way of finding k-nearest data without calculating the exact distances and comparisons.

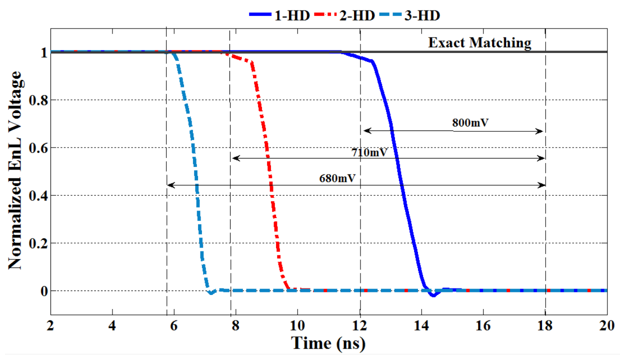
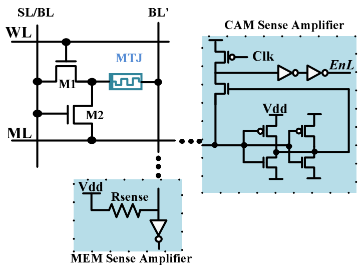
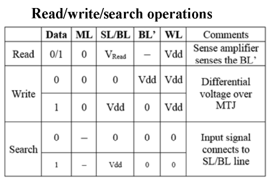


Figure 6. Normalized match-line voltage in different supply voltages

Note that the NN*gine* has been designed to work for general *NN* applications with different number of data points and varying sizes of each data point as long as the size fits into the maximum capacity limit of NN*gine* which is determined at the design time. To run the applications which handle smaller data sizes on the NN*gine*, the controller can gate on the rest of the unused TCAM bits. The other advantage of the proposed CAM is that it can work as a normal memory to load and save the data. While memory does not use the accelerator functionality, it can store some parts of data of DRAM. To support the normal memory operation, as shown in Figure 5, the NNAM also has another sense amplifier on each of vertical lines, thus allowing it to read data on each memory line.




Read/write/search operations

	Data	ML	SL/BL	BL'	WL	Comments
Read	0/1	0	V_{Read}	–	Vdd	Sense amplifier senses the BL'
Write	0	0	0	Vdd	Vdd	Differential voltage over MTJ
	1	0	Vdd	0	Vdd	
Search	0	–	0	0	0	Input signal connects to SL/BL line
	1	–	Vdd	0	0	

Figure 7. 2T-1R cell structure in memory and CAM functionalities

B.1. TCAM Cell

The NNAM includes multiple TCAM cells. In a NVM-based TCAM, values are stored based on the NVM resistance states (Low or High). In this work, we use a 2T-1MTJ CAM cell structure which can be used as both CAM and memory cells [5] thanks to its high density and power efficiency. Figure 7 shows the TCAM cell and its control signals used in the CAM and memory functionalities. The device can be switched between ON and OFF by applying either negative or positive voltage across the device. During the read, BL' precharges V_{dd}, and then the sense amplifier reads the state of the MTJ. The search operation can be performed by applying a small voltage across the BL and SL and sensing the ML voltage using the CAM sense amplifier.

B.2. NN Detector

The NN detector checks if the number of matched TCAM rows is equal to ***k*** while the supply voltage changes, and lets the controller stop the DVS. The main design goal of the circuit is that it should be fast enough to work at the speed of DVS (see Figure 5). For example, implementing "1s" counter circuitry in CMOS incurs large energy and latency overhead. Thus, we designed an *analog* detector circuit which keeps track of the output signal of the sense amplifier and informs the controller when ***k*** rows are matched. Figure 8 illustrates the structure of the KNN detector circuit. Each active row in NNAM adds an extra current to the BL line. A large number of active *EnL* signals increases the BL current (I_{BL}). This current is sensed using an analog-to-digital convertor (*ADC*). The ADC compares the voltage of V_K with the threshold voltage (V_{th}) in order to identify the number of ***k*** active rows. The threshold voltage is set based on the ***k*** value where the larger ***k*** requires higher V_{th} voltage. At the very beginning of the search operation, the *dt* digital signal is a large number since the difference between the voltage V_k and V_{th} is high since the number of matched lines is small. In contrast, decreasing the level of voltage by the DVS, the number of matched rows starts increasing and results in increase of I_{BL} and V_k. Thus, it lowers the *dt* signal. This signal sends to the controller described in Section 3.2.3. The accuracy of ADC is based on the number of NNAM rows. In our case, we use 13-bit ADC to detect all possible hits on the CAM rows of 8K=(2^{13}) [27].

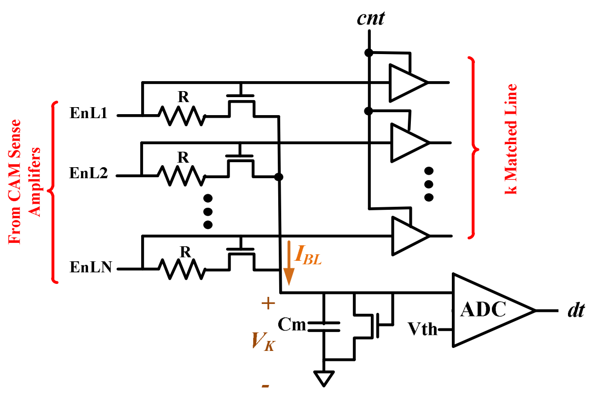


Figure 8. Detector circuit to find the k nearest neighbors of TCAM block with N rows

B.3. DVS block and Controller

The controller dynamically changes the supply voltage every cycle using the DVS block. We used the voltage scaling strategy similar to [28]. Since decreasing the supply voltage at a small step incurs large latency to find the *k* nearest data, in order to accelerate the search operation, the controller uses two different levels of the voltage steps, in our case 50mV and 3mV. First, it decreases the voltage with a larger step, i.e., 50mV. Once the detector circuit gives a negative *dt* signal at *i*-th iteration, it decreases the voltage with a small voltage step from the voltage used at the previous (*i-1)*-th iteration. Once the *dt* digital signal becomes 0, the controller stops the DVS and activates the *cnt* signal to the NN detector, so delivering the found data to the next stage.

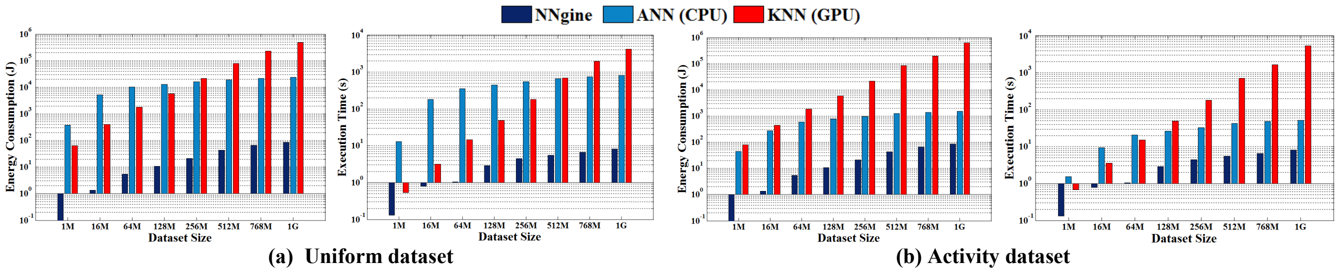


Figure 9. Energy and performance comparison of KNN, ANN and NNgine for uniform (a) and activity dataset (b) over different data sizes

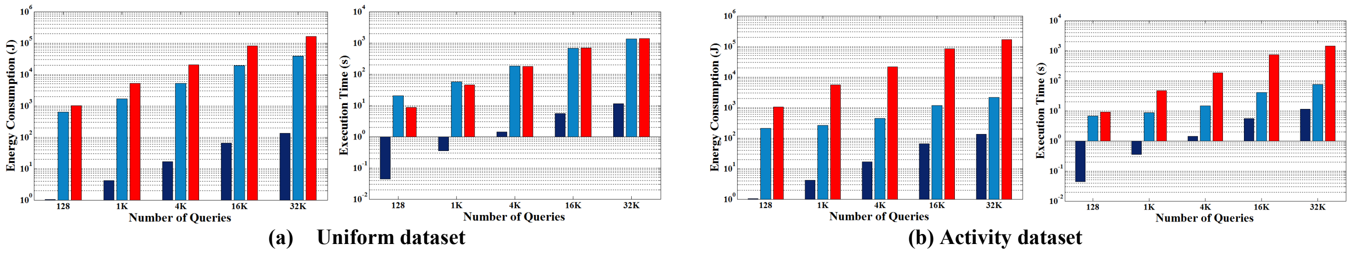


Figure 10. Energy and performance comparison of KNN, ANN and NNgine for uniform (a) and activity dataset (b) over different query numbers

IV. EXPERIMENTAL RESULTS

A. *Experimental Setup*

We experimented the three kNN approaches for the two different data sets called *activity* and *uniform*. The first dataset, *activity*, was from the physical activity monitoring data set PAMAP2 [29]. This data set 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 second dataset, *uniform*, is uniform-randomly drawn to represent the large data variance. We note that the uniform dataset used is consistent with previous work [30], [9] and IoT data is also expected to have such large variation [2]. To convert important features into a floating-point vector (i.e., a sequence of bits stored in TCAM) of each data point from the raw data, we extracted four features; mean, standard deviation, energy, and correlation for each of the three axes of an accelerometer in the way used in [31]. In total, we extracted 36 features for the three accelerometers, and then further applied the principal component analysis (PCA) to select most significant features. Finally, a data point is represented as a 16-element vector which includes most significant features by PCA dimension reduction with 0.1% of variance.

We compare NN*gine* performance and energy efficiency to two state-of-the-art kNN approaches, GPU-based kNN [9] and CPU-based ANN (Approximated NN) [30] in the same technology nodes. The GPU-based kNN parallelizes the distance computations and *k* data points selections for multiple queries using OpenCL. The ANN approach builds binary trees for segmented data subsets, which include reference data points closely located, and selects k neighbors for queries in an exact mode. We have evaluated the two popular approaches on AMD Radeon R9 390 GPU with 8GB memory and Intel i7 7600 CPU with 16GB memory. For the measurement of the system and processor power, we used Hioki 3334 power meter and AMD CodeXL [32]. The CodeXL tool has been also used to extract GPU performance counters such as cache hits. The efficiency of the proposed NN*gine* accelerator is evaluated using the HSPICE simulator. We used the TCAM cell with a large ON resistance to distinct the output voltages of different hamming distances. The TCAM design, NN detector and the DVS block [33] have been designed in circuit level using 45-nm TSMC technology. All circuit level simulations performed considering 10% (3σ) process variations on the transistor sizes and threshold voltage using 5000 Monte Carlo simulations.

B. *NNgine Configuration*

As discussed in Section 3.1, there are three main factors that affect the overhead in terms of energy and performance: the number of subcores, ***L***, the number of banks in a subcore of the first stage, ***B***$_1$, and the NNAM size, ***Size***$_{NNAM}$. As the number of subcores increases, parallelism can be better utilized, but also more overhead may be incurred. Thus, we need to balance the number banks and the NNAM size. The second factor is the number of banks since a large ***B***$_1$ increases the number of write operations between the NNAM stages resulting in both energy and latency degradation. In contrast, the small number of banks requires a larger NNAM block to provide enough capacity. Lastly, a large-size NNAM consumes a lot of energy during search with long latency. These overheads come from the large buffer size that a NNAM requires to distribute all data between CAM rows at the same time.

Table 2 shows the energy consumption and performance over different NN*gine* configurations to handle 1Gbyte reference data in the accelerator. Our results show that the case of ***B***$_1$=512, ***SIZE***$_{NNAM}$=4K, and **L**=512 can provide the lowest energy-delay product (EDP) compared to all other configurations. We observed that this case minimizes the energy consumption of NN*gine* by balancing the energy consumption of each NNAM and the number of writes between the stages.

TABLE 2. ENERGY CONSUMPTION OF PROPOSED NNGINE ON DIFFERENT CONFIGURATIONS

	L **(512 subcores)**			***L*** **(1024 subcores)**		
B_I (# of Bank)	*256*	*512*	*1024*	*256*	*512*	*1024*
$Size_{NNAM}$ (# of rows)	*8K*	*4K*	*2K*	*4K*	*2K*	*1K*
Energy (mJ)	6.72	5.37	9.92	5.82	6.31	8.43
Execution Time (ms)	0.45	0. 34	0.31	0.34	0.35	0.39
EDP (uJ.s)	3.02	1.82	3.07	1.97	2.20	3.27

C. Accuracy

Many applications using kNN algorithms exploit Euclidean distance metric to calculate the distance of the query data with the reference dataset. However, the NN*gine* decides the distances between two data points using hamming distance metric. In order to verify if our technique can support such applications as well with minimal accuracy loss, we compare our technique to the kNN using Euclidean distance metric. We here use the 8 users' log of the activity dataset since the dataset includes the ground truth, i.e., the activity IDs of each data point. Table 1 shows the accuracy of NN*gine*. The result shows that the proposed NN*gine* using hamming distance strategy can achieve to 99.48% accuracy in average. This results shows that NN*gine* can provide enough accuracy to run more general kNN applications.

TABLE 3. ACCURACY OF HAMMING DISTANCE-BASED NNGINE OVER DIFFERENT USERS OF THE ACTIVITY DATA SET

Dataset	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Avg.**
Accuracy (%)	99.55	99.48	99.60	99.46	99.40	99.42	99.47	**99.48**

D. Energy and Performance Scalability

The efficiency of the three different approaches, NN*gine*, GPU-based kNN and CPU-based ANN, mainly depends on the size of datasets, denoted ***S*** (*=N*64* since sixteen 4-bytes features are used), and the number of queries, denoted ***M***.

Figure 9 shows the performance and energy comparison of the approaches in different dataset sizes in logarithmic scale. We used two datasets: uniform and activity, when ***k***=64 and ***M***=16K, varying the dataset size ***S*** from 1M bytes to 1G bytes. As shown in the figure, unlike our proposed NN*gine*, the energy and performance to run two other existing implementations are significantly impacted by the dataset size. Even though the GPU-based kNN outperforms the CPU-based ANN for the small dataset (e.g., 1MB), the energy and performance efficiency degrade due to the limited memory capacity. Interestingly, for large size data, ANN outperforms GPU-based kNN by using the preprocessed tree structure which is used for handling queries. However, the efficiency is still poor since it takes a long time to build the segmented tree structure.

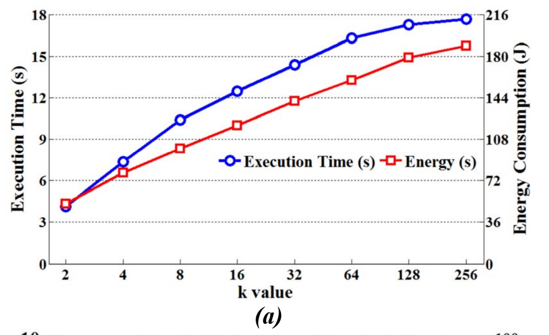
In contrast to these approaches, the proposed NN*gine* does not significantly degrade the energy and performance over the input dataset increase. Although there is small impact on the efficiency of NN*gine* as the dataset size increases due to the large number of write operations, our NN*gine* achieves great energy saving and performance improvement. For example, the evaluation shows that, even in the worst case scenario, we can achieve 5590**x** and 17**x** energy efficiency improvement as compared to the GPU-based kNN and ANN approaches respectively. In addition, as technology is moving toward big data which ponds to have large diversity, kNN algorithms should effectively handle such data set close to the uniform distribution. However, as shown in the results, ANN would not handle such dataset since the number of the segmented tree structures is likely to increase as data diversity grows. For example, in the performance comparison for the largest 1GB uniform dataset, we can achieve 510**x** and 100**x** speedup compared to the ANN and GPGPU approaches respectively because the NN*gine* can search the nearest neighbors without any dependency to data types.

We next evaluate how the three technique scale with the number of queries, ***M***. Figure 10 shows the energy and performance comparison of the three approaches in logarithmic scale while varying ***M*** from 128 to 32K for the 512MB dataset. Even though the GPU-based computation can handle multiple queries at a time, the number of parallelized queries is limited because of the memory capacity. CPU-based ANN were not parallelized. As a result, the performance and energy efficiency of the two techniques significantly decrease. In contrast, since our design can find the nearest neighbors for a single query very quickly (e.g., 0.5ms when ***S*** = 1GB and ***k*** = 64), this compensates for the serial searches for multiple queries. As a result, the performance increases linearly to the number of queries. Thus, even for searches of 32K queries in the uniform dataset, the NN*gine* has energy saving 1206x and 287x higher than GPU-based kNN and ANN respectively. In terms of performance, we outperform the two techniques by up to 121**x** and 117**x**.

E. NNgine Efficiency for k and Feature Sizes

In a general cases, the number *k* is relatively small, e.g., under 128, to capture representative close data points [9]. As discussed 3.2.2, the proposed design performs DVS on the CAM blocks until the number of activated rows matches the desired *k*. Thus, using a larger *k* increases the waiting time and the amount of written data through stages, which may result in a negative impact on the performance and energy. We compare the energy consumption and performance of NN*gine* for different *k* values. We did not include the CPU and GPU approaches since the parameter *k* has a very small impact on their efficiency. As shown in Figure 11*a*, the required energy and performance of NN*gine* is only slightly degraded as the *k* value increases. However, the performance and energy differences are minimal and still much more efficient than the two other approaches. For example, the NN*gine* can achieve 100x performance speedup compared to ANN when using k=256, M=32K and S=1GB.

The feature size, i.e., the vector dimension of each data point, is another parameter given by applications. Some applications can decrease the feature size without impact on their target accuracy through statistical analysis such as PCA. Our design can work more efficiently with the smaller feature size because of the decreased size of CAM bit lines. Figure 11*b* shows the energy and performance of the NN*gine* running at different feature sizes. For example, if the applications can decrease the dimension from 16 to 8, the NN*gine* can improve the performance and energy consumption by 1.2**x** and 1.5**x** respectively.



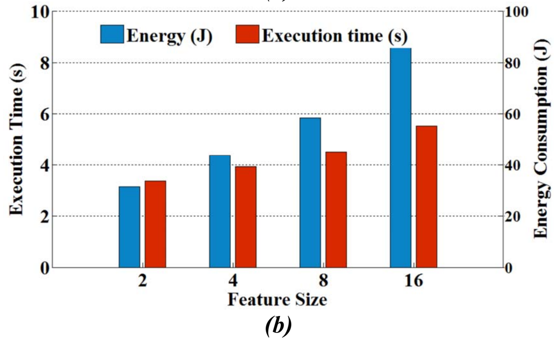


Figure 11. Energy consumption and performance of NNgine (a) over different k values (S=1GB, M=32K) (b) for different dataset feature sizes (S=1G, M=16K)

Comparing the NN*gine* with the state-of-the art KNN accelerator [32], shows that NN*gine* can provide at least an order of magnitude higher performance and energy improvement. Instead of previous work which exactly calculate the distance of input query with dataset values, our NN*gine* efficiently detects the k nearest rows just by analogy sensing the nearest rows in CAM structure.

V. CONCLUSION

In this paper, we propose a resistive in-memory accelerator for the nearest neighbor search, called NN*gine*, based on TCAM blocks. The proposed technique has the capability of storing large data and finding the nearest neighbors without CPU or GPU computation. Based on the specialized associative memory which utilizes dynamic voltage scaling and analog detector circuitry, the proposed NN*gine* significantly improves energy and performance efficiency in k-nearest neighbor search. Our evaluation shows that NN*gine* can improve KNN energy efficiency for a large dataset of 1Gbyte by 5590**x** while accelerating the computation performance by 510**x,** over the existing approaches. This approach provides at least an order of magnitude higher performance and energy saving compared to previous KNN accelerators.

REFERENCES

[1] J. Gantz, et al., "Extracting value from chaos," IDC iview, vol. 1142, pp. 1-12, 2011.

[2] J. Gubbi, et al., "Internet of Things (IoT): A vision, architectural elements, and future directions," Future Generation Computer Systems, vol. 29, pp. 1645-1660, 2013.

[3] S. Ghoreishi, et al., "Adaptive Uncertainty Propagation for Coupled Multidisciplinary Systems," AIAA Journal, 2017. [4] A. M. Aly, et al., "M3: Stream processing on main-memory mapreduce," in Data Engineering (ICDE), 2012 IEEE 28th International Conference on, 2012, pp. 1253-1256.

[5] Q. Guo, et al., "AC-DIMM: associative computing with STT-MRAM," in ACM SIGARCH Computer Architecture News, 2013, pp. 189-200.

[6] N. Bhatia, "Survey of nearest neighbor techniques," arXiv preprint arXiv:1007.0085, 2010.

[7] Y. Deng, et al., "Content-based search of video using color, texture, and motion," in Image Processing, 1997. Proceedings., International Conference on, 1997, pp. 534-537.

[8] Z. Song, et al., "K-nearest neighbor search for moving query point," in Advances in Spatial and Temporal Databases, ed: Springer, 2001, pp. 79-96.

[9] V. Garcia, et al., "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching," in Image Processing (ICIP), 2010 17th IEEE International Conference on, 2010, pp. 3757-3760.

[10] Q. Guo, et al., "A resistive TCAM accelerator for data-intensive computing," in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011, pp. 339-350.

[11] M. Imani, et al., "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific, 2017, pp. 757-763.

[12] M. Imani, et al., "Ultra-Efficient Processing In-Memory for Data Intensive Applications," in Proceedings of the 54th Annual Design Automation Conference 2017, 2017, p. 6.

[13] M. Imani, et al., "Exploring hyperdimensional associative memory," in High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on, 2017, pp. 445-456.

[14] Y. Kim, et al., "ORCHARD: Visual Object Recognition Accelerator Based on Approximate In-Memory Processing," IEEE International Conference On Computer Aided Design (ICCAD), 2017 2017.

[15] H. Jégou, et al., "Searching with quantization: approximate nearest neighbor search using short codes and distance estimators," 2009.

[16] D. Qiu, et al., "GPU-accelerated nearest neighbor search for 3D registration," in Computer Vision Systems, ed: Springer, 2009, pp. 194-203.

[17] A. Goel, et al., "Small subset queries and bloom filters using ternary associative memories, with applications," in ACM SIGMETRICS Performance Evaluation Review, 2010, pp. 143-154.

[18] R. Cohen, et al., "Simple efficient TCAM based range classification," in INFOCOM, 2010 Proceedings IEEE, 2010, pp. 1-5.

[19] M. Hoseinzadeh, et al., "Reducing access latency of MLC PCMs through line striping," ACM SIGARCH Computer Architecture News, vol. 42, pp. 277-288, 2014.

[20] Y. Kim, et al., "CAUSE: critical application usage-aware memory system using non-volatile memory for mobile devices," in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 2015, pp. 690-696.

[21] M. Imani, et al., "ACAM: Approximate Computing Based on Adaptive Associative Memory with Online Learning," in ISLPED, 2016, pp. 162-167.

[22] M. Imani, et al., "MASC: Ultra-low energy multiple-access single-charge TCAM for approximate computing," in 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016, pp. 373-378.

[23] M. Imani, et al., "Resistive Configurable Associative Memory for Approximate Computing."

[24] M. S. Razlighi, et al., "LookNN: Neural network with no multiplication," in 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 1775-1780.

[25] M. Imani, et al., "Efficient neural network acceleration on gpgpu using content addressable memory," in 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 1026-1031.

[26] M. Imani, et al., "A Low Power And Reliable 12T SRAM Cell Considering Process Variation In 16nm CMOS," INTERNATIONAL JOURNAL OF TECHNOLOGY ENHANCEMENTS AND EMERGING ENGINEERING RESEARCH, vol. 2, p. 76, 2014.

[27] Y. Zhu, et al., "A 10-bit 100-MS/s reference-free SAR ADC in 90 nm CMOS," Solid-State Circuits, IEEE Journal of, vol. 45, pp. 1111-1121, 2010.

[28] P. K. Krause, et al., "Adaptive voltage over-scaling for resilient applications," in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, 2011, pp. 1-6.

[29] A. Reiss, et al., "Creating and benchmarking a new dataset for physical activity monitoring," in Proceedings of the 5th International Conference on PErvasive Technologies Related to Assistive Environments, 2012, p. 40.

[30] S. Arya, et al., "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," Journal of the ACM (JACM), vol. 45, pp. 891-923, 1998.

[31] N. Ravi, et al., "Activity recognition from accelerometer data," in AAAI, 2005, pp. 1541-1546.

[32] "AMD CodeXL, http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/."

[33] Y. Wang, et al., "Energy Efficient RRAM Spiking Neural Network for Real Time Classification," in Proceedings of the 25th edition on Great Lakes Symposium on VLSI, 2015, pp. 189-194.