Efficient Query Processing in Crossbar Memory

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

***Abstract*—Today's computing systems use huge amount of energy and time to process basic queries in database. A large part of it is spent in data movement between the memory and processing cores, owing to the limited cache capacity and memory bandwidth of traditional computers. In this paper, we propose a non-volatile memory-based query accelerator, called NVQuery, which performs several basic query functions in memory including aggregation, prediction, bit-wise operations, as well as exact and nearest distance search queries. NVQuery is implemented on a content addressable memory (CAM) and exploits the analog characteristic of non-volatile memory in order to enable in-memory processing. To implement nearest distance search in memory, we introduce a novel bitline driving scheme to give weights to the indices of the bits during the search operation. Our experimental evaluation shows that, NVQuery can provide 49.3× performance speedup and 32.9× energy savings as compared to running the same query on traditional processor. In addition, compared to the state-of-the-art query accelerators, NVQuery can achieve 26.2× energy-delay product improvement while providing the similar accuracy.**

I. Introduction

Data management systems (DMS) are the standard tools for collecting and serving large amounts of information for web applications and end users. Over the past decade, data generation has grown exponentially due the diversity of collection sources [1], [2]. In addition, organizations collect large amounts of information for decision making and business analytics [3]. In the majority of scenarios, the execution time of DMS queries tends to increase linearly and sometimes exponentially as more records are stored in a single server instance. This has been one of the main challenges of DMS and its caused by the the hardware and software co-design limitations [4].

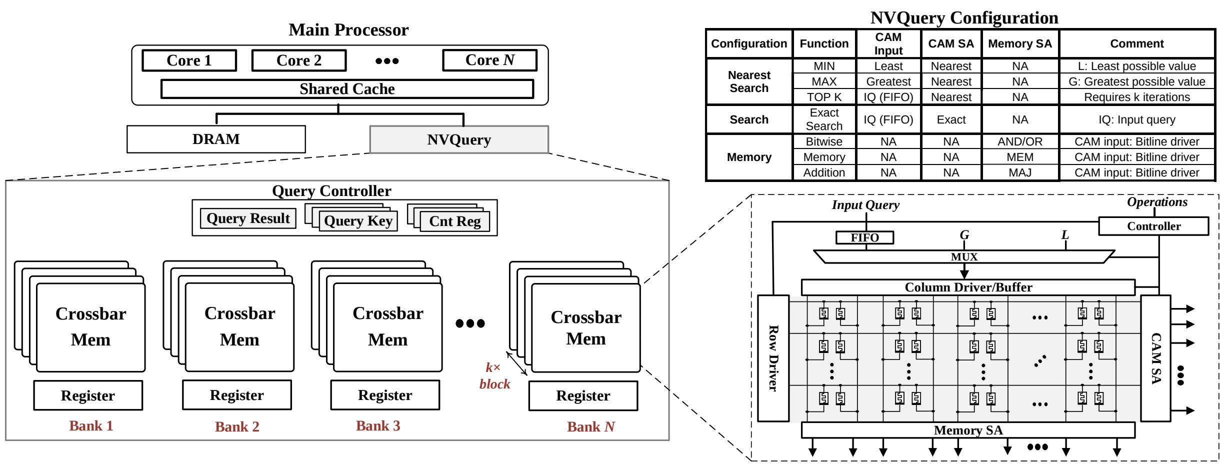
In the hardware aspect, several efforts have been made to accelerate computation by paralleling operations on a co-processor or GPUs [5], [6]. However, in most cases data movement has been a bottleneck due to the fact that large amounts of information tend to reside on slow storage devices such as disks. In most Structured Query Language (SQL) accelerator studies, this data overhead is not taken into account and as a consequence their results do not show a valuable improvement over general designs [7]. On the other hand, softwares have been developed to adapt to the nature of particular tasks. In the case of interactive data analysis, the focus is often less on exactness of the result and more on timeliness or responsiveness, which gives us the opportunity to approximate the result within a margin of error in order to accelerate the SQL query computation [4]. Query processing slows down significantly when running on the large data sets provided by connected devices. Data movement is the main bottleneck of current computing systems wherein the size of data increases over the cache capacity of the processing core [8]. Limited memory bandwidth makes the condition worse as data is delayed each time the main memory is accessed.

Near data computing and processing in-memory (PIM) are two efficient techniques which improve the cost of query processing [9]. Near-data computing puts the computing units close to the main memory, in order to avoid data movement cost in computation [10]. Although this technique improves the computation efficiency, it has some challenges including: (i) cost of large CMOS-based computing unit and (ii) cost of integrating the memory and logic in a single chip. The introduction of non-volatile memories has made it possible to process data in the memory itself, resulting in the concept of PIM [11], [12], [13], [14]. Resistive RAM (ReRAM) is one such memory and enjoys the benefit of low energy, high switching speeds, high density, and scalability. PIM processes data within memory, eliminating the need for integration between large processing cores and the memory. However, the existing PIM techniques support only simple functions like bit-wise or search operations [15], [16]. Not only it is too cumbersome to break down a simple query function like search into a series of bit-wise computations but it also minimizes the benefits of using PIM. To the best of authors' knowledge, this paper is the first attempt to implement an efficient PIM based query processor which supports a wide range of query functions.

In this paper, we propose a novel non-volatile, memory-based query processing accelerator, called NVQuery, that supports several query functions within a memory instead of processing them in traditional core. NVQuery supports wide range of query functionalities including aggregation functions, prediction functions, bit-wise operations, addition, exact and nearest distance search operation. The configurable crossbar memory structure of our design supports these functionalities inside the memory. It exploits the analog characteristic of non-volatile memory to also enable the nearest distance search capability. Our experimental evaluation on SQL queries shows that, compared to the state-of-the-art query accelerators, NVQuery can achieve 26.2× energy-delay product improvement while providing similar accuracy.

II. Related Work

Several efforts have been made in order to accelerate DMS querying by using specialized hardware. GPUs in particular have been widely used to parallelize the 'SELECT' SQL queries with results that range from 20x to 70x speed up [17]. However, they do not take into consideration the data movement overhead of these tasks and assume only the computation cost. In addition, it has been demonstrated that the bandwidth and cache capacity of GPU devices are the main bottlenecks of database computations. For instance, work in [18] examines multiple GPU systems and acknowledges that



NVQuery Configuration

Configuration	Function	CAM Input	CAM SA	Memory SA	Comment
Nearest Search	MIN	Least	Nearest	NA	L: Least possible value
	MAX	Greatest	Nearest	NA	G: Greatest possible value
	TOP K	IQ (FIFO)	Nearest	NA	Requires k iterations
Search	Exact Search	IQ (FIFO)	Exact	NA	IQ: Input query
Memory	Bitwise	NA	NA	AND/OR	CAM input: Bitline driver
	Memory	NA	NA	MEM	CAM input: Bitline driver
	Addition	NA	NA	MAJ	CAM input: Bitline driver

Fig. 1. Proposed NVQuery architecture with N banks and k blocks, crossbar implementation of memory banks, and supported configurations.

unless the full working set of data can fit into the memory on a GPU, PCI Express bus will be a bottleneck.

As a consequence, researchers have worked on optimizing the data movement through memory. In the areas of distributed computation, Al-Kiswany *et al.* describe StoreGPU, a distributed storage system that uses pinned, non-pageable memory on the host system to reduce the impact of data transfer [19]. Gelado *et al.* in [20] introduce an asymmetric distributed shared memory that defines two types of memory updates which determine when to move data on and off the GPU. These optimization efforts only focus on conventional memory technologies and the computation still occurs on computing units. A query service by Google called, BigQuery is capable of searching through petabytes of data [21]. The latency is minimized by paralleling queries over multiple servers and columnar storage of data over multiple memories or memory banks. However, this distribution of queries and data results in huge energy requirements. Also, it does not support data manipulation queries and has huge latency bottleneck with updating data.

Approximating the results of SQL query has been used to reduce the required waiting time by producing results within acceptable error bounds. The most famous querying framework based on approximation is sampling-based approximate querying (SAQ) [22], [23], where the computation is performed over a small random subset of the data. The error in the estimate is specified using a confidence interval or error bars. However, SAQ suffers from several shortcomings such as ignoring the tails of the data and being useless to complex queries. Poti *et al.* [4] proposes deterministic approximate querying (DAQ) schemes that formalize a determinist approach to approximate the results by taking advantage of the bit value representations. Their approach reads the table records, starting from the most significant bit, one by one and adjust deterministic error bounds with respect to the bits not seen yet. Also, DAQ evaluations confirm efficient approximation giving estimates with less than 1% error with a speedup of 6x for SQL predicate queries and 2-4x for aggregation.

In contrast, in this paper we propose a novel computer architecture which completely addresses the data movement issue of the query processing systems by doing computations inside the memory.

TABLE I
NVQUERY SUPPORTED FUNCTIONALITIES

	Notation	Functions
Aggregation	$F(S_I) \rightarrow S_O$	MIN, MAX, Average, Count
Bit-wise Operations	$F(S_I) \rightarrow S_O$	AND, OR, XOR (Combination of AND, OR)
Addition	$F(S_I) \rightarrow S_O$	In-memory addition
Comparison	$= \ \leq \ \geq$	Bit-wise and value-wise comparison
Predict	p	Exist, Search condition, Top

III. NVQuery Accelerator

A. Overview of NVQuery Architecture

Fig. 1 shows the general architecture of the proposed NVQuery. The proposed NVQuery integrates with DRAM and enables the main processor to accelerate query processing. NVQuery can also be used as a secondary storage to improve the effective DRAM capacity. NVQuery consists of N banks, where each has k memory blocks.Each memory block can be configured as memory or query accelerator.

Our design is a heterogeneous architecture, where the NVQuery co-operates with main processor in order to find the query result. In NVQuery, each memory block returns a result of the query, independent from other blocks. Therefore, to find the result of a query in the whole data set, the main processor receives output response of each memory block (a total of $N \times k$ values instead of the entire data). Finally, it processes data to find the result of query over entire data set. In this way, the load on memory bandwidth due to query processing and its related costs are significantly reduced.

B. Supported Functionalities

In this section, we describe the functions supported by the proposed non-volatile query processor, called NVQuery. NVQuery can support several essential query processor functions inside the memory and avoid costly data movement across memory hierarchy. Table I lists the NVQuery support functionalities. NVQuery supports a large number of essential functions including aggregation (`MIN, MAX, Average, SUM` and `Count`), comparison (equality or non-equality), and boolean (such as `AND`, `OR`). In addition, NVQuery can process prediction functions such as `Exist` and `Top` in memory.

We map all query functionalities explained in Fig. 1 to NVQuery which can work in three main configurations: (i) look-up table (LUT) with capability of exact search, (ii) nearest distance search, and (iii) memory. We propose a new memory architecture which can process data locally without reading it. In each of these configurations, our design processes query operations without approximating the result. In the following subsections, we explain how each query operation can be supported in memory.

1) Exact Search: The most common operation in many query processors is looking up for a set of data which matches with input query. A typical search query involves a brute-force search through a LUT till the data is located. This is usually implemented in one of the two ways, (i) word-by-word search and (ii) bit-by-bit search. A word-by-word search looks through every stored word in the LUT sequentially and finds a match. In the worst case, it involves processing each and every element present in the LUT. The bit-by-bit search scans through one bit (but same index) for multiple words at a time. The first iteration analyses a particular bit index of every word in the LUT, looking for a match with the corresponding entry in the input query. The following iterations are performed only on the words filtered by previous iterations. This approach does not analyse all the elements since the size of candidate pool decreases after each iteration. The exact search operation supports `Exist, Search functions` and is further extended to implement `Count` function in the query processor. The `Count` output is given by the number of hits for an exact search query. Our design adds a counter block to NVQuery in order to support this query.

2) Nearest Distance Search: NVQuery can be configured to perform the closest distance search operation inside the memory. The bit-by-bit search described above can be used to implement this functionality. Here, the nearest data is the one which remains selected for the maximum number of iterations. Our design exploits this functionality to support aggregation functions like `MIN` and `MAX` and prediction functions like `Top k`. Running these queries on traditional core has a time complexity of $O(\log n)$. However, our hardware can find `MIN, MAX` queries in a single cycle and `Top k` in k cycles.

`MIN`: This query runs on a set of stored data to find the minimum value. To perform this query in LUT, NVQuery block adopts the nearest distance search configuration and searches for the data which has the closest distance to the minimum possible value. Fig. 2 shows an example of running `MIN` query in nearest distance LUT for unsigned numbers. Our design searches for an entry which has the closest distance to zero. In the case of signed values, this number is the largest possible negative number (single one followed by a chain of zeros).

`MAX`: As shown in Fig. 2, to find the data with the maximum value, we search for the entry which has the least distance from the largest positive number. For unsigned values, the largest value is a chain of ones (1111...1), while in the case of signed numbers, this value is represented by a zero followed by a chain of ones (0111...1).

`Top k`: To search for k values closest to the input data, we perform the nearest distance search for k iterations. After each iteration, our design deactivates the selected word and repeats the nearest distance search on the remaining words. This approach gives a set of k nearest values arranged in the order of their proximity to the input. Our design also supports bit-wise/value-wise comparison by searching for the exact and nearest values.

3) Bit-wise Operations and Addition: A traditional processor implements bit-wise logic operations in the main core. The operands are fetched from the main memory and brought through the memory hierarchy all the way up to the core. The core then performs the required computations. On the other hand, our design implements these operations in the memory itself, avoiding the need to transfer data from memory to the computing core. For executing these operations, NVQuery is set into memory configuration and the output is obtained from memory SA. This operation can support the following queries: `AND, OR, XOR` and `Average`. Note that our design supports average query by using a counter and sending the data to main processor.

IV. Hardware Support

This section explains the hardware implementation of NVQuery and the way in which it supports the functions described in Section III-B. NVQuery is designed using a crossbar non-volatile memory architecture. The crossbar is configured in such a way that a set of two storage elements in the crossbar corresponds to one bit data. Data 0 is stored as $\{R_{HIGH}\ R_{LOW}\}$, while 1 is stored as $\{R_{LOW}\ R_{HIGH}\}$. However, our architecture does not use any access transistors for these elements, hence called 0T-2R. Implementations like 2T-2R require access transistors. This makes the design unsuitable for a crossbar memory, reducing the area density benefit of non-volatile memories. Moreover, the presence of transistors introduces non-linearity to the system. On the other hand, 0T-2R doesn't need access transistors and can be implemented on a conventional crossbar memory, making it more area efficient.

As shown in Fig. 1, the crossbar memory in NVQuery is supported by many peripheral components. The controller receives the input query and generates the appropriate control signals. It is also responsible for collecting the output of the block and forwarding it for further processing. The multiplexer, controlled by the controller, selects the inputs which drive the bitlines of the crossbar memory. These can be data from input query (in case of search operations) or greatest (corresponding to `MAX` where bitlines are driven by the largest positive number) or least (corresponding to `MIN`). The column driver drives the bitlines of the crossbar. It not only provides the execution voltages for different operations but also maps the input query to the required bitline voltage levels. Row driver is responsible for charging wordlines (also called match-lines due to the nature of operations). It is also responsible for selecting/activating different words, for example word selection after a bit-by-bit search iteration. It also provides a limited set of voltage options essential to the working of crossbar. The crossbar is equipped with sense amplifiers (SAs) on both the wordlines (CAM SA) and the bitlines (memory SA). Fig. 3 shows these SAs. The CAM SAs are responsible for detecting charging and discharging behaviors of wordlines. The nMOS-capacitor circuit acts as a latch. The inverter-diode-NOR circuit deactivate the wordlines as soon as the first edge is detected or the sampling signal for *Exact* is set. As a result, the latch is set only for the wordlines which discharge before this deactivation. The memory SAs are buffers with special resistors to support bit-wise and memory

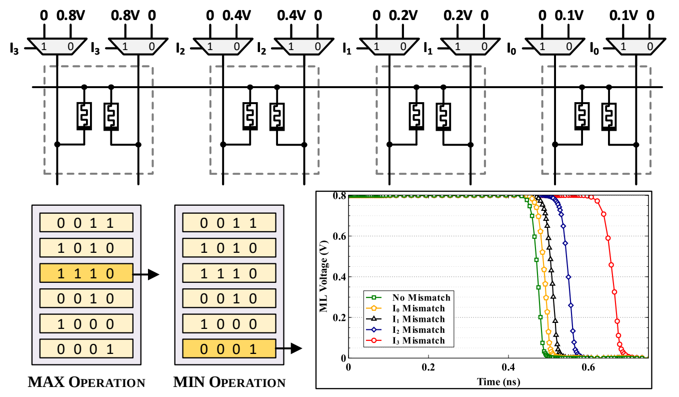


Fig. 2. NVQuery in nearest distance search configuration, the corresponding discharging characteristics, and `MAX` and `MIN` operations.

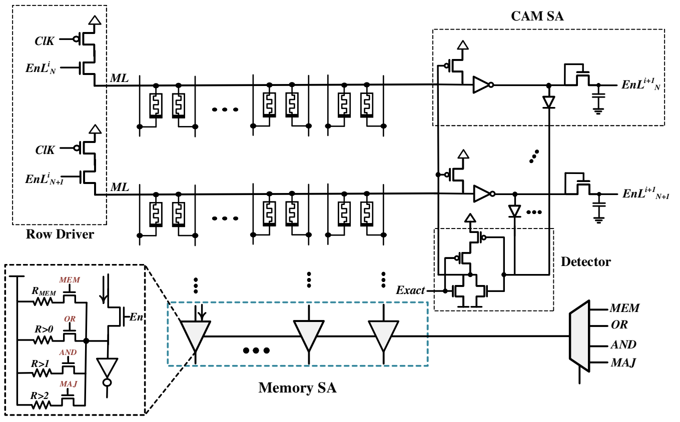


Fig. 3. Circuit level implementation of CAM SA, Memory SA, and Row Driver.

operations as described in Section IV-C. We now explain the way in which NVQuery enables different functions discussed in Section III-B.

A. Exact Search

To implement the LUTs discussed in Section III-B1, NVQuery uses content addressable memory (CAM) configuration of crossbar. Fig. 3 shows the structure of non-volatile crossbar CAM, capable of searching for stored data which exactly matches the input query. During search operation, all the match-lines (MLs) pre-charge to V_{dd}. The input buffer (column driver) distributes the query point to all CAM rows using vertical bitline. Any cell with the same stored data as input query discharges the ML. The sense amplifier, connected to the horizontal ML, determines the equality of the input and stored data by sampling the ML voltage.

B. Nearest Distance Search

CAM has been extensively used to implement search operations. Different versions of CAM implementations (*e.g.* TCAM) on different types of hardware (crossbar, 2T-2R, 3T-1R, *etc.*) have been active topics of research recently. However, majority of the previous work revolves around exact and nearest hamming distance search operations. Hamming distance is a good criterion when considering hyper-dimensional vectors where the index of a bit does not matter. Only the face value of a bit and the total number of mismatches between the stored data and the input query are considered. Such a comparison is not practical for many real life applications where a query to the processor is dependent on the binary weighted values of the stored data.

To support such queries, some researchers have proposed the division of a memory block into stages [24]. In such an architecture, the first m most significant bits of data are stored in the first stage, the next m significant bits in the second stage and so on. Then, a search is performed sequentially, starting from the first stage. The output of a stage selects the rows to be activated in the following stage. This increases the weight of the initial stages with respect to the later stages. However, the m bits in a stage are treated as having the same binary weight. This leads to inaccurate results in many cases. In this work, we address this issue by introducing a new method to assign binary weights to the bits within a stage.

For a search in conventional CAM, the match-lines (MLs) are pre-charged to V_{dd} and then bitlines are driven with V_{dd} or 0 depending upon the input query. The MLs of rows with more number of matches discharge earlier. The line to discharge first is the one with minimum mismatch with the input query. To give binary weight to the bits, we modify the bitline driving voltage. Suppose a stage contains m bits $(m-1:0)$. The bitlines which were earlier driven with V_{dd} and now driven with a voltage $V_i = V_{dd}/2^{(m-1-i)}$ where i denotes the index of a bit in the stage. Fig. 2 shows CAM in nearest search configuration for a stage size of 4 bits. As shown by our results, a match in the most significant bit results in faster ML discharging current than lower indices. We exploit this difference and design a CAM which can find the binary value nearest to the input query.

However, as the number of bits increases, the bitline voltage V_i becomes very small. We limit the minimum available voltage source output to $100mV$. Moreover, the maximum voltage that can be applied is limited by the threshold voltage of the non-volatile elements. This ensures that the data in the memory is preserved. This upper bound is set to $1.8V$. Hence, the allowable voltage levels include $0.1V, 0.2V, 0.4V, 0.8V$ and $1.6V$, restricting the stage size to 5 bits. In this work, we split the CAM into multiple stages of 4-bits each for simplicity and then search for the nearest distance row in a serial manner, starting with the stage containing the most significant bits.

C. Bit-wise Operation and Addition

Although a search based CAM can accelerate several functionalities in NVQuery, it cannot support a major part of queries such as addition, average, and all bit-wise operations. In order to make NVQuery a general design for query processing accelerator, we modify the sense amplifiers in the vertical bitlines to support bit-wise operations. Fig. 2 shows the sense amplifier in a single NVQuery bitline to support bit-wise operations. In this mode, each block works as memory instead of CAM, where one of the vertical bitlines in each CAM cell is activated. The tail of the shared bit-line is connected to a sense amplifier. Since our design supports AND and OR functions, the sense amplifier has two main parts: one for AND operation and a simple sense amplifier to support OR. These circuits work on the basis of the leakage current through the vertical bitline. When several rows in memory are active, each row leaks current through vertical bitlines depending upon the resistance value. If the stored bit is 1 (low resistance), this

TABLE II
APPROXIMATION IN 16-BIT ADDITION

Approximated Bits	4	8	12	14	16
Error (%)	0.006	0.098	1.56	6.25	25
Energy (pJ)	3.52	2.41	1.3	0.75	0.197
Latency (ns)	182	133	84.7	60.5	36.3

current is large, while in the case of 0, leakage is significantly small. The goal of OR operation is to identify the presence of at least one high (1) bit in all activated rows. Therefore, we use a special resistor such that in the case of at least single high bit, it turns the output signal to one. However, for AND operation the goal is to find a case such that at least one input is not 1. In that case, the AND circuitry uses an appropriate sense resistance.

Interestingly, prior work shows that crossbar memory can further support addition within the memory [25], [26]. This approach breaks down an operation into a series of `NOR` operations. The logic family used in the paper executes `NOR` in crossbar memory with a latency of just 1 cycle. This functionality is supported by NVQuery due to its regular structure (unlike CAMs with access transistors), enabling it to perform data computations within memory. This addition can be partly implemented to support `XOR`. In the case when approximate results are acceptable, the sense amplifier at the bitlines can be used to improve the performance of NVQuery. The truth table for 1-bit full adder shows that the sum bit (S) can be obtained by inversion of the carry bit (C) in 75% of the cases. The sense amplifier calculates C (majority) in one step by simply using an appropriate sense resistance. S is obtained by inverting C. This introduces a worst case error of 25%. However, this error is reduced significantly by approximating only some LSBs depending upon the level of accuracy desired. The MSBs are calculated accurately using the techniques described in [25]. Table II shows the error corresponding to different number of approximated bits for an 8-bit addition. By calculating the carry bit correctly, the proposed approximation approach limits the effect of an error to one bit and does not propagate it.

Addition is extended to implement average function. The output of successive additions is sent to the processor, where the average is obtained by bit-shifting or simple division.

V. EXPERIMENTAL RESULTS

A. Experimental setup

For detailed evaluation of the proposed NVQuery, we run circuit-level simulations in HSPICE with 45nm TSMC technology. We use VTEAM [27] model of memristors with I_{ON}/I_{OFF} ratio of 10^3 for non-volatile memory crossbar design. We develop software-based cycle-accurate simulator (based on C++) which emulates the functionalities of the designed NVQuery. This allows us to speed up the simulation time significantly and verify the proposed design with diverse practical data sets. The simulator exploits accurate models of the hardware, e.g., time and power extracted from the aforementioned circuit-level simulation to evaluate the efficiency of the proposed design. We compare NVQuery performance and energy efficiency with with state-of-the-art query processing approaches running on the same technology node. We evaluate two popular approaches, sampling-based approximate querying (SAQ) [23] and deterministic approximate querying (DAQ) [4] on Intel i7 7600 CPU with 8GB memory. For measurement of the processor power, we use Hioki 3334 power meter. We use a dataset consisting a table of `Census` of 10 million tuples using 32-bit unsigned integers to compare the efficiency of different techniques. This data is popularly used to model populations of various kinds ranging from cities and organizations to word frequencies in natural language corpora. The SQL server contains a single table with one 10GB column of randomly generated records. In the rest of the paper, power and performance results have been reported for 1000 queries from aggregation and prediction functions over five randomly generated datasets.

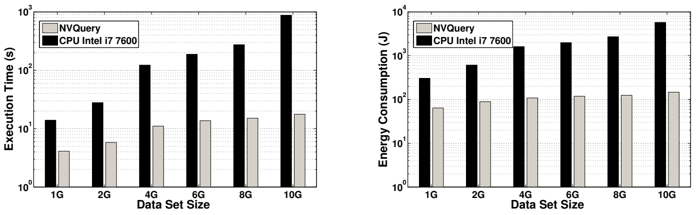


Fig. 4. Energy consumption and performance of query processing running on traditional core and the proposed NVQuery.

B. NVQuery Efficiency

Here we highlight the advantage that NVQuery can provide in computing each query function. Table III compares the energy savings and performance speedup of running different queries on proposed NVQuery as compared to a digital ASIC design. Each energy is reported when 10 queries run on 1k dataset. The selected dataset is small so that the reported values compare the computation energy without data movement cost. The digital system is designed using System Verilog in 45nm ASIC flow. The result shows that NVQuery improves the computation cost of all queries significantly. Specifically, queries such as `MAX, MIN` and/or `TOP k` can be processed in a single cycle, instead of processing in $O(n)$ or $O(logn)$ time. Our evaluation shows that our design can provide 11.8× energy improvement and 26.85× performance speedup on average compared to digital approach for nearest distance search-based queries. Similarly, our design can achieve on average 13.7× and 92.1× (5.8× and 0.9×) energy savings and performance speedup over exact search (memory functionalities, e.g. addition). Although, the performance of in-memory addition is less than that of digital-based design, but considering the cost of data movement, it makes sense to process data locally in-memory. In large size query processing, the data movement dominates the computation cost, which motivates us to perform in-memory computations to avoid data movement issue.

We also compare the efficiency of the proposed NVQuery with the state-of-the-art query accelerators SAQ [23] and DAQ [4] using 8G dataset size. We select those configurations of SAQ and DAQ which result in the best EDP improvement. Our experimental evaluation shows that, NVQuery can achieve 105.0× and 26.2× EDP improvement as compared to SAQ and DAQ designs in exact mode. The main advantage of NVQuery comes from addressing data movement issue.

C. NVQuery & Dataset Size

While running real dataset, the main advantage of NVQuery comes from addressing the data movement issue. Fig. 4 shows

TABLE III
ENERGY CONSUMPTION AND PERFORMANCE SPEEDUP OF QUERIES IN NVQUERY NORMALIZED TO DIGITAL DESIGN OVER 1K DATA

	Nearest search		Search	Memory	
Queries	*MAX/ MIN*	*Top 1*	*Search/ Count*	*Addition/ Average*	*Bit-wise*
Energy Improv.	9.5×	14.1×	13×	5.8×	46.7×
Speedup	24.2×	29.5×	92.1×	0.9×	122.6×

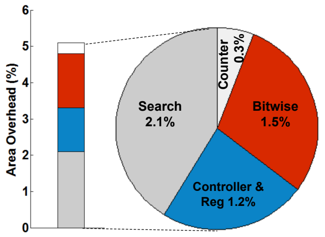


Fig. 5. Area overhead as compared to conventional crossbar memory

the average energy consumption and performance of running query processing on traditional core and NVQuery when the data set size changes from 1GB to 10GB. Our evaluation shows that the NVQuery has higher advantage in processing the nearest distance search and related functions such as `MIN`, `MAX` or `Top` queries. However, to see the average NVQuery improvement, we generate the same amount of queries running on the dataset. Our evaluation shows that increasing the data size significantly increases the energy and execution time of traditional cores. However, this increment is minor in NVQuery as it can locally process the data. As our result in Table III shows, NVQuery not only avoids the overhead of data movement, but also provides much cheaper computation than traditional cores. This difference is more prominent when the size of the dataset passes 8GB, which is the available main memory size in our tested platform. In such case, the traditional cores require to bring data up from the hard disk, which significantly slows down the computation. Comparing the energy and performance of NVQuery for 10G data shows that, our design can achieve 34.7× energy savings and 49.3× performance speedup as compared to traditional processor running the same query tasks.

NVQuery has both memory and query processing functionalities. We added peripheral circuitry to crossbar memory to support nearest distance exact search operation, bitwise/addition operations, counter and controller. Fig. 5 shows that proposed NVQuery has up to 5.1% area overhead compared to the conventional crossbar. The search circuitry takes 2.1% extra area. Counter and bit-wise circuits add 0.3% and 1.5% area overhead to design. Finally, the controller and registers take the rest 1.2% area overhead.

VI. CONCLUSION

In this paper we propose a novel memory architecture which can accelerate query processing inside the memory. NVQuery supports a large range of query functionalities inside the memory. Our design exploits the analog characteristic of the non-volatile memory to design a configurable memory architecture which can look for exact or nearest distance values. Our result shows that NVQuery not only improves the cost of each query processing, but also completely addresses the data movement issue by locally processing the data in memory. Our experimental evaluation shows that, in comparison with the state-of-the-art query accelerators, NVQuery can achieve 26.2× energy-delay product improvement while providing similar accuracy.

VII. ACKNOWLEDGMENT

This work was supported by NSF grant #1527034 and Jacobs School of Engineering UCSD Powell Fellowship.

REFERENCES

[1] 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.
[2] M. Chen *et al.*, "Big data: a survey," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.
[3] H. Chen *et al.*, "Business intelligence and analytics: From big data to big impact.," vol. 36, no. 4, pp. 1165–1188, 2012.
[4] N. Potti *et al.*, "Daq: a new paradigm for approximate query processing," *VLDB Endowment*, vol. 8, no. 9, pp. 898–909, 2015.
[5] M. Imani *et al.*, "Acam: Approximate computing based on adaptive associative memory with online learning," in *ISLPED*, pp. 162–167, 2016.
[6] M. Samragh *et al.*, "Looknn: Neural network with no multiplication," in *DATE*, IEEE, 2017.
[7] C. Gregg *et al.*, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *ISPASS*, pp. 134–144, IEEE, 2011.
[8] J. LeFevre *et al.*, "Miso: souping up big data query processing with a multistore system," in *SIGMOD/PODS*, pp. 1591–1602, ACM, 2014.
[9] S. H. Pugsley *et al.*, "Comparing implementations of near-data computing with in-memory mapreduce workloads," *IEEE Micro*, vol. 34, no. 4, pp. 44–52, 2014.
[10] R. Balasubramonian *et al.*, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.
[11] M. Hoseinzadeh *et al.*, "Reducing access latency of mlc pcms through line striping," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 277–288, 2014.
[12] M. Hoseinzadeh *et al.*, "Spcm: The striped phase change memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, p. 38, 2016.
[13] M. Saremi, "Carrier mobility extraction method in chgs in the uv light exposure," *Micro & Nano Letters*, vol. 11, no. 11, pp. 762–764, 2016.
[14] N. Khoshavi, S. Salehi, and R. F. DeMara, "Variation-immune resistive non-volatile memory using self-organized sub-bank circuit designs," in *Quality Electronic Design (ISQED), 2017 18th International Symposium on*, pp. 52–57, IEEE, 2017.
[15] M. Imani *et al.*, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *ASPDAC*, pp. 757–763, IEEE, 2017.
[16] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *HPCA*, IEEE, 2017.
[17] P. Bakkum *et al.*, "Accelerating sql database operations on a gpu with cuda," in *3rd GPGPU workshop*, pp. 94–103, ACM, 2010.
[18] D. Schaa *et al.*, "Exploring the multiple-gpu design space," in *IPDPS*, pp. 1–12, IEEE, 2009.
[19] S. Al-Kiswany *et al.*, "Storegpu: exploiting graphics processing units to accelerate distributed storage systems," in *HPDC*, pp. 165–174, ACM, 2008.
[20] I. Gelado *et al.*, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 347–358, ACM, 2010.
[21] J. Tigani and S. Naidu, *Google BigQuery Analytics*. John Wiley & Sons, 2014.
[22] S. Acharya *et al.*, "Aqua: A fast decision support systems using approximate query answers," in *VLDB*, pp. 754–757, Morgan Kaufmann Publishers Inc., 1999.
[23] S. Agarwal *et al.*, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *EuroSys*, pp. 29–42, ACM, 2013.
[24] M. Imani *et al.*, "Resistive cam acceleration for tunable approximate computing," *IEEE TETC*, 2016.
[25] N. Talati *et al.*, "Logic design within memristive memories using Memristor-Aided loGIC (MAGIC)," *IEEE TNano*, vol. 15, pp. 635–650, July 2016.
[26] M. Imani *et al.*, "Ultra-efficient processing in-memory for data intensive applications," in *DAC*, IEEE, 2017.
[27] S. Kvatinsky *et al.*, "VTEAM: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, pp. 786–790, Aug 2015.