2019 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress

Hardware-Accelerated Similarity Search with Multi-Index Hashing

Leandro Santiago*, Victor C. Ferreira *, Brunno F. Goldstein*,
Alexandre S. Nery‡, Leandro A. J. Marzulo†, Sandip Kundu§, Felipe M. G. França*
†Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro (UERJ), Brazil
Email: leandro@ime.uerj.br
‡Departamento de Engenharia Elétrica, Faculdade de Tecnologia, Universidade de Brasília, DF, Brazil
Email: anery@redes.unb.br
*Programa de Engenharia de Sistemas e Computação - COPPE, Universidade Federal do Rio de Janeiro (UFRJ), Brazil
Email: {vcruz, lsantiago, bfgoldstein, felipe}@cos.ufrj.br
§Department of Electrical and Computer Engineering University of Massachusetts Amherst, USA
Email: kundu@umass.edu

***Abstract*—Similarity search plays a major role in a large variety of database applications where high-dimensional data, usually derived from images and videos, are mapped onto indexed structures. Its essence relies on a k-nearest-neighbors (kNN) algorithm to find similar contents given a specific input data. An efficient way to store and speedup such searches is to apply kNN in Hamming space by encoding the data as binary and splitting it into multiple hash tables. This paper proposes an efficient Hardware-Accelerated Similarity Search co-processor architecture using Multi-Index Hashing as storage structure. The accelerator is specified in Verilog hardware description language and implemented in a Xilinx low-cost Zynq FPGA. Performance, circuit-area and power consumption results are presented, with the accelerator being up to 13× and 18× faster than the corresponding C/C++ code on the ARM host processor when running the SIFT and GIST datasets, respectively, while also requiring less power.**

I. Introduction

In the past few years, the urge for novel hardware architectures to efficiently run Artificial Intelligence and Machine Learning applications has led big companies like Google [1], Microsoft [2] and ARM [3] to invest on new hardware accelerators for these domain specific problems. Data surge has also driven new architectural trends, such as in situ processing, to satisfy modern data-intensive applications that demands higher throughput and lower latency that cannot be achieved only by traditional Von-Neumann CPUs. Consequently, Internet of Things (IoT) applications are increasingly processing a large amount of data through mobile devices, wireless sensors and other low-power devices.

Similarity Search is a class of data-intensive Machine Learning applications that can leverage from these kind of accelerators. It is essential for several important applications such as computer vision [4], image retrieval [5] and natural language processing. Conceptually, it is a simple search problem that tries to recover the most similar data from a database given a query as input. The *k-nearest neighbors* (kNN) search algorithm is a well-known specialization which only retrieves until k of the most identical contents [6].

Modern applications produce high dimensional real-valued feature vectors, imposing a challenge issue to appropriately manage them in practical time. To overcome this barrier, a wide range of techniques have been proposed to improve the efficiency in similarity search domain by compacting the high dimensional data into binary codes [7]–[11]. Binary code generation is performed by indexing high-dimensional vectors into the Hamming space preserving similarity in the original space. Selecting an adequate indexing structure is crucial to large scale search performance since it drastically reduces points in space where the kNN works. Degradation is also an important factor as most of the structures falls into the *curse of dimensionality* [12] diminishing to linear search. Hash based structures are well known for this kind of scenario and recent work has shown that it has significant speed-up opportunities especially when distance search is done in Hamming Space [6].

Multi-index hashing (MIH) is a fast kNN search technique in Hamming space that relies on multiple hash tables to store indexes mapped by disjoint sub-strings partitioned from a binary code to speed up the search [13]. The key idea arises from the fact that two similar binary codes also contain similar binary sub-codes which enables to run parallel near neighbors searches on these sub-codes in order to find all neighbors for the complete binary code.

This paper presents a hardware co-processor design to accelerate similarity search in hamming space using multi-index hashing structure. Since kNN search is a fundamental algorithm to solve a variety of problems, future IoT applications need a kNN search method in order to run high performance systems. Thus, the goal is to enable efficient MIH processing in a low-cost in-situ environment such as IoT applications that would benefit from the hamming space search which is very simple and efficient to implement on hardware. As MIH executes several look-ups in multiple hash tables, we identify that the Hamming distance calculation and partially sort of the candidate neighbors from each hash table are data-intensive and practical to be accelerated in the hardware.

978-1-7281-3024-8/19/$31.00 ©2019 IEEE
DOI 10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00138

The Register Transfer-Level (RTL) architecture is specified in Verilog hardware description language and implemented in the programmable logic of a low-cost Xilinx FPGA (XC7Z020). Performance, circuit-area and power requirement analysis shows that the proposed accelerator achieves good speedup compared to embedded ARM processor while consuming less power.

The rest of the paper is organized as follows. Background related to this work is presented in Section II. Section III describes our proposed architecture, followed by experiments and results analyzed in Section IV. In Section V, we discuss the applicability of our accelerator and indicate future works. Finally, we conclude this work in Section VI.

II. Background

This section describes the state-of-the-art researches on kNN algorithms and Multi-Index Hashing, including some basic background information about both techniques.

A. K-Nearest Neighbors Search Algorithm

The Nearest Neighbor Search problem can be defined as finding the closest value of a query q in a set P of n objects represented as points in a space. K-Nearest Neighbors Search is a generalization of the Nearest Neighbor problem, with the number of closest points defined by $k > 1$. Searching for close neighbors when P is composed by high-dimensional objects is challenging and requires efficient storage/indexing strategies and distance calculation. Indexing strategies on high-dimensional data is an emerging topic that recent works have targeted with different types of techniques, such as: hierarchical k-means, kd-trees, multi-probe locality sensitive hashing (MPLSH) and multi-index hashing (MIH). Each data structure prunes the search space, drastically reducing the query time. Hierarchical k-means [14] splits the database into clusters that are stored in a tree fashion, with the tree leaves holding similar entries that are retrieved when a query reaches it. Kd-trees [15] works in a similar way, but each cluster is now a random slice of the database.

Multi-probe locality sensitive hashing (MPLSH) [16] and multi-index hashing (MIH) are two indexing structures that leverage the power of hashing as storage but with some improvements. MPLSH works with multiple hash tables, where each hash location stores similar entries. Similar to MPLSH, MIH [13], [17] stores vectors into a set of hash tables but using part of data as index. This technique, detailed in Section II-B, requires that all data be in binary representation. As a counterpart, binary representation is a straightforward and cheaper way to implement in hardware, making it a good candidate for the proposed hardware accelerator.

Distance can be calculated through a set of different metrics like Euclidean distance, Hamming distance [18], cosine similarity, learned distance metrics [19], Manhattan distance and Jaccard similarity. Hamming distance has shown to be a prominent way to calculate distance by taking advantage from the MIH structure [13].

B. Multi-Index Hashing

Multi-index hashing (MIH) is a fast technique to search for the nearest neighbors in Hamming Space [13]. MIH increases throughput by storing data d as binary code into m hash tables indexed with m disjoint sub-codes derived from d. The search splits the query into m sub-codes, seeking for neighbor candidates over each hash table, in parallel. All candidates are then validated using the original query to remove any false r-neighbors, hence enabling exact kNN search over sub-linear run-time.

The central idea behind MIH is that two similar binary codes also contain similar sub-codes. Instead of searching for neighbors through the whole binary code, matching can be done by its index. Reducing the search space not only saves time, but also reduces the amount of memory necessary to process the whole database. Consider the following premise for the MIH algorithm: two binary codes h and g both with b bits. Each one is partitioned into m disjoint binary sub-codes with s bits length, where $s = \lfloor \frac{b}{m} \rfloor$ or $s = \lceil \frac{b}{m} \rceil$ and b divisible by m. When h and g differ by r bits or less, then at least one sub-code s must differ by $\lfloor \frac{r}{m} \rfloor$. That is, there must be a sub-code k, $1 \leq k \leq m$, where

$$\| h^{(k)} - g^{(k)} \|_H \leq \lfloor \frac{r}{m} \rfloor \quad (1)$$

when $\| h - g \|_H \leq r$ with $\| \cdot \|_H$ denoting Hamming norm. The proof of the premise is derived from Pigeonhole Principle [13].

To exemplify the MIH working principle, suppose a 64-bit query q and that all neighbors which are at 16 Hamming distance from q are going to be searched. Also, suppose that each query is divided into 16-bit sub-strings, resulting on 4 sub-strings for each hash table. For each 64-bit binary code b, if b and q differ by at most 16 bits, it means that at least one of the associated sub-strings differs at least $\frac{16}{4} = 4$ bits, as generalized by Equation (1). Using this premise, the search is done only for $r = 4$ radius neighbors over each sub-string in the corresponding hash table. Thus, the number of lookups is reduced from $\binom{64}{16} \approx 4,9 \times 10^{14}$ to $4 \times \binom{16}{4} = 7,280$ when looking for 16 Hamming distance neighbors of the query q . As the lookups are executed, each matched sub-string is verified as true 16-distant neighbor. This approach provides high order of magnitude speedup with the ability to search millions of 128-bit codes within a second using a search radius of 30 bits. Moreover, the algorithm ensures to find exact neighbors without any approximation.

III. The Accelerator Architecture

Considering the MIH structure presented in Section II-B, we propose a MIH hardware co-processor to accelerate hamming distance calculation and partial sorting operations in an input stream of searched candidates.

A. Top View

The kNN search using MIH requires the delineation of a Hamming search radius r in order to retrieve the k nearest

neighbors from each hash table. Since different search queries with radius r may produce more candidates than k, the radius parameter must not be fixed. The search process iterates the radius parameter r starting from $r = 0$ until the top-k similar neighbors are found. Also, each iteration goes through an odd-even merge sort pipeline to partially sort the already found candidates from each hash table. At the end, the accelerator produces the top k candidates, sorted according to their hamming distance similarity.

Figure 1 shows how the MIH co-processor can be used to accelerate the kNN search for a 16-bit binary query partitioned in 4 sub-strings and a Hamming search radius $r = 0$. In this case, one sub-string represents itself a neighbor ($r = 0$). Each neighbor of sub-string is used to access a MIH hash table entry containing a list of indexes which point out to the 16-bit binary code stored in the database. The complete neighbor data is first recovered from the database through the indexes stored in each entry of the hash. Then, the accelerator calculates the Hamming distances between the query and the data streamed from the database. The distances are independently ordered for each hash entry and, at the end, they are ranked until the top-k candidates are found.

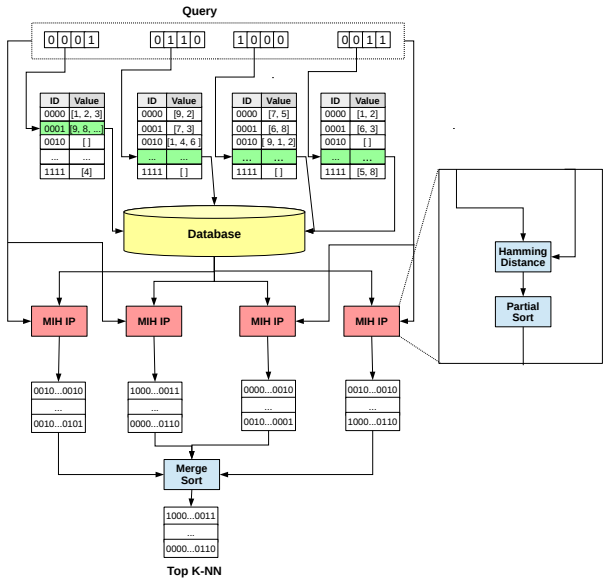

Fig. 1. K-nearest neighbor search with Multi-Index Hashing (MIH) hardware accelerator (MIH Intelectual Property - IP). It presents a search for a 16-bit binary query partitioned in 4 sub-strings and a Hamming search radius $r = 0$.

At certain conditions, kNN search in MIH may require multiple search radius for all queries to find the most k similar neighbors. In addition, huge databases are likely to map several data onto the same hash table entry. Thus, the MIH data-intensive computation part consists of the hamming distance calculation and the partial sorting of candidates processed in each hash table. As data-intensive processing applications can benefit from higher memory bandwidth, this work proposes a MIH accelerator that implements specialized circuit for hamming distance calculation and partial sorting. Each hash table can be connected to an independent co-processor that quickly provides a sorted small list representing the top nearest neighbors and, due to this, the merge sort executed for all hash table candidates can be optimized considering the produced ordered list.

B. Accelerator Design

To support the data-intensive execution phases, we designed the MIH accelerator to communicate with external systems via Advanced extensible Interface (AXI) Stream protocol, which is part of the Advanced Microcontroller Bus Architecture (AMBA4). The accelerator is composed of two main cores: the AXI Stream Slave, which receives the query and the stream of neighbor data, and AXI Stream Master, which sends back the sorted distances and the corresponding neighbor index. To simplify it, we described the architecture to support 64-bit binary string hamming distance calculation and 64 sorted neighbor distances and indexes. For different configurations, the architecture's components have to be extended to supply the specification. Figure 2 depicts the MIH accelerator architecture.

The AXI Stream slave core keeps a double buffer to store sorted distances and indexes. Initially, the first 64-bits is stored in the *query* register while the subsequent data are stored in the *rx_data* register. As soon as data is received, the hamming distance is calculated. Afterwards, the data index (receiving order) is sent to the sorter component. Both Hamming distance and sorter components are detailed in Section III-C. When the first buffer is full (64 distances), the core waits until the ordering is completed. To fill the second buffer, the distances are submitted to ordering only when the incoming data value is less than that of the last distance in the first sorted buffer. Applying that rule on the second buffer, we avoid to sort far neighbors, early eliminating the false nearest neighbors. When the second buffer is full, the core starts the merge process between both buffers that updates the smaller distances adding them into first buffer and reinitialize the second buffer. The merge process is detailed in Section III-D. After merging, the logic comes back to insert good distances on the second buffer. These processes continue until receiving the last neighbor data.

AXI Stream master core starts when slave core has finished. First, the indexes are returned and shortly thereafter the distances are sent. Since each one has a double buffer, two pointers are handled to choose the smaller distances between them. The total of indexes and distances sent are limited to buffer length (64).

C. Specialized Components

The co-processor's efficiency comes from the fast specialized circuit to calculate Hamming distance and the sorting network to sort the indexes and distances received by slave core. Hamming distance between binary codes a and b can be efficiently obtained by counting number of $1's$ in $a \oplus b$. There are several techniques to perform bit-counting operation, which is known as *popcount* (population counts) operation. We

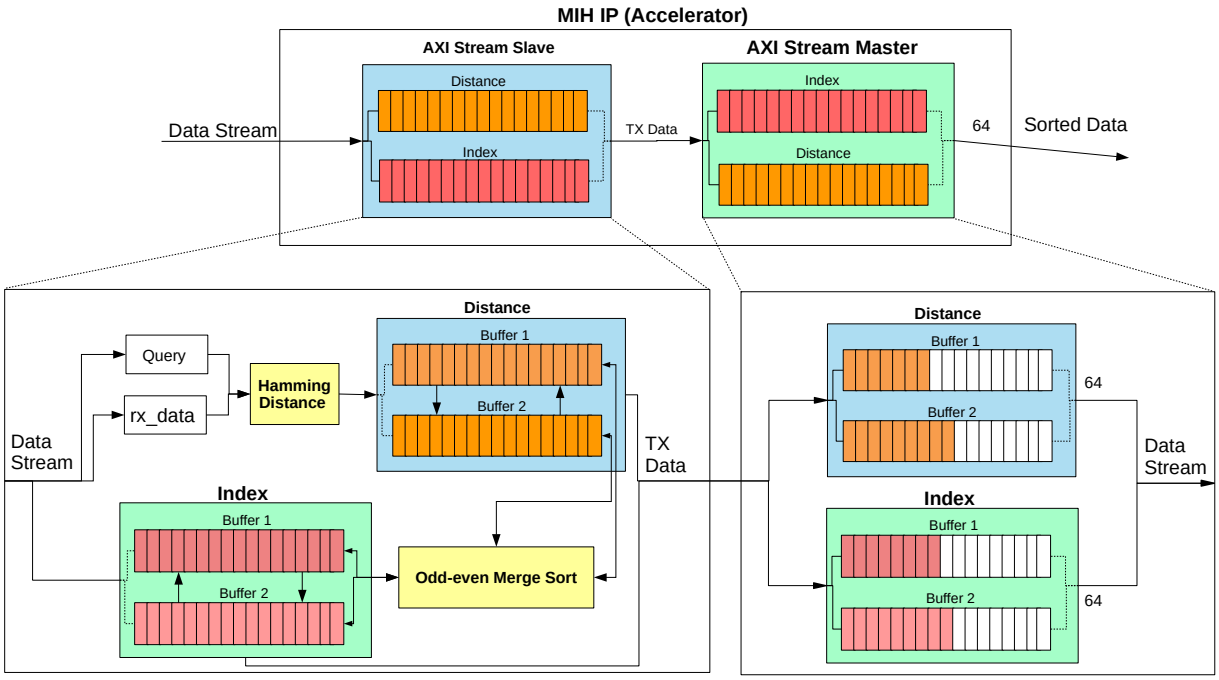

Fig. 2. Overview of the MIH Accelerator. The AXI Stream Slave interface receives the query and the stream of neighbor data (64-bit each), while the AXI Stream Master sends back the top-64 sorted distances and the corresponding neighbor index.

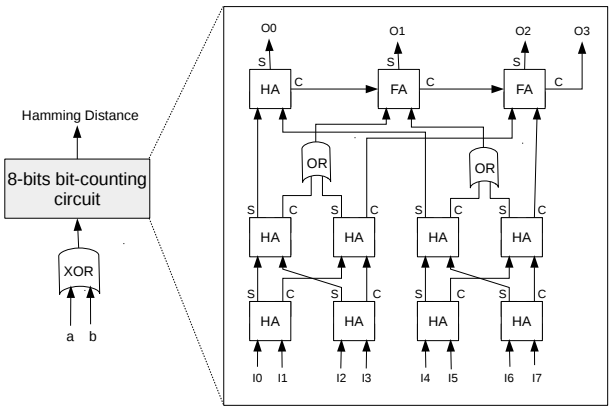

Fig. 3. 8-bits Hamming Distance counting component, as proposed in [20].

implement the hamming distance component with bit-counting circuit proposed in [20], as shown in Figure 3.

The bit-counting circuit can be recursively generated by extending the 8-bits counting circuit, following the method proposed in [20]. The base circuit consists of 3 layers connecting *Half Adder* (HA) and *Full Adder* (FA) components with *OR* logic gate. In [20], the authors inform that their circuit requires fewer resources and achieves low delay than other hardware solutions. Table I summarizes the number of gates required to build such bit-counting circuit for different input sizes. Our Hamming distance component is implemented to take only one cycle.

TABLE I
BIT-COUNTING CONSUMPTION. EACH FULL ADDER CONSUMES 5 GATES AND HALF ADDER 2 GATES.

Bit length	# FAs	# HAs	# OR gates	Total Gates
8	2	9	2	30
16	7	19	4	77
32	18	39	8	176
64	41	79	16	379

The partial sorting component is specified through sorting network algorithms, which are very efficient sorting approaches to implement and run on hardware. These algorithms are adequate to sort a short fixed sequence of numbers using fixed steps of comparisons and swaps operations. The circuit is composed of horizontal wires and vertical comparators where the unsorted elements are applied at left side and sorted two-by-two in each stage. The comparator between two wires swaps the item to keep the smaller input on the upper output wire while the larger is placed on lower output wire. In [21], several sorting network implementations have been evaluated on FPGA and the authors conclude that Batcher's odd-even merge sort network is the best cost benefit in terms of hardware resource and throughput. Batcher's algorithm describes a recursive non-adaptive procedure to generating the sorting network for a number of items with size of power of 2, taking $O(\log(n)^2)$ depth to sort n elements utilizing $O(n\log(n)^2)$ resources. Consequently, we embedded this sorting network

TABLE II
BATCHER'S ODD-EVEN MERGE SORT PIPELINE CONSUMPTION.

Bit length	# Pipeline Stages	# Comparator-and-Swap
4	3	5
8	6	19
16	10	63
32	15	191
64	21	543

into our proposed MIH accelerator.

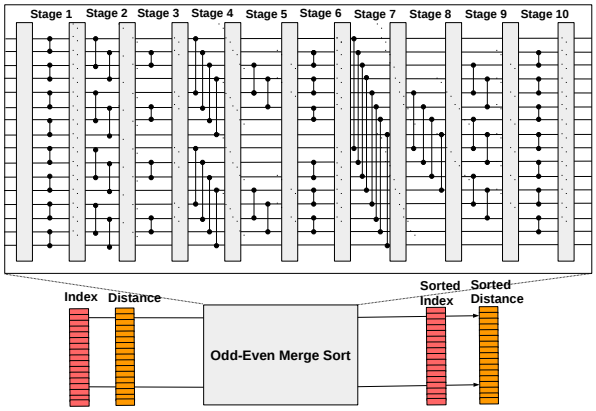

Fig. 4. Odd-even merge sort component, exemplifying the sort network with 16-bit inputs, 16-bit outputs and 10 pipeline stages.

Figure 4 details the partial sorting component, exemplifying the odd-even merge sort network with 16-bit inputs and 16-bit outputs. We adapt the circuit to realize indexes sorting synchronized with distance order checking. The network is implemented as a pipeline circuit where the distance and index buffers are sent together at first stage. The whole sorting is finished in the last stage (stage 10). To sort 64 elements the pipeline consists of 21 stages with the total of 543 comparators as shown in Table II.

D. Buffer Merge Strategy

As addressed in Section III-B, the merge of buffer 1 and 2 occur when both are full. This process finalizes by storing sorted elements between both buffers into buffer 1 and re-initializing buffer 2 with maximum input values.

Buffer merge process occurs in three steps as demonstrated in Figure 5. Both buffers have 64 items totaling 128, where the smaller 64 will be inserted in buffer 1 and the remaining will be discarded. In step 1, the concatenation of first 32 elements of buffer 1 and 2 is sent to the odd-even merge sorting network, which takes 21 cycles to complete. In the next step, the last 32 elements of both buffers are concatenated and sent to the sorting network component. It also takes 21 cycles to finish, however after step 1 has concluded, we can obtain the sorted results for step 2 in the next cycle due to the pipeline circuit implementation. To summarize, these two steps are completed in 22 cycles. Since the elements of buffer 1 and 2 are initially sorted, the first 32 items produced by step 1 are correctly ordered. In the last step, the concatenation of the last 32 elements of step 1's buffer and the first 32 from step 2's buffer is submitted, taking more 21 cycles. The first 32 items are updated as last 32 elements into buffer 1. Thus, the buffer merge process always takes 43 cycles.

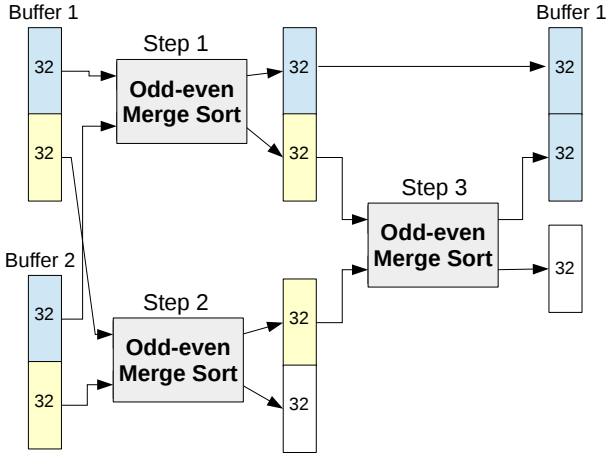

Fig. 5. Buffering merge process of buffer 1 and 2. The process is concluded in 43 cycles - 22 cycles corresponding to steps 1 and 2 plus 21 cycles comes from step 3.

IV. EXPERIMENTS

All experiments and results presented on this section were executed and implemented on a ZYNQ XC7Z020-1CLG400C Xilinx FPGA attached on the Pynq-Z1 board, which contains 512 MB DDR RAM and several I/O interfaces. The Zynq FPGA architecture embeds a Cortex-A9 multi-core processor around its reconfigurable logic. The proposed MIH accelerator specified in Verilog is synthesized and implemented through Xilinx Vivado tools, version 2018.1. We use a Linux Pynq Image v2.3 to execute Python 3.0 and implement the kNN search application based on MIH to access the hardware accelerator. As the boards are not attached to any machine, they work as a standalone systems.

A. Datasets

To evaluate our proposed MIH co-processor implementation, we conduct the experiments on two datasets: 80M 384D GIST descriptors from 80 million tiny images [4] and 1B 128D SIFT descriptors from BIGANN dataset [22]. Since these datasets are not binary, we use the hyperplane Locality Sensitive Hashing (LSH) [23] to generate 64-bit binary codes. Each experiment requires three set of data: a training set to adjust the LSH parameters, a base set to populate MIH structure and the query set that contains the query data. For GIST, we randomly split the whole data into $300K$ items for the training set, 1000 items for query set and the remaining for the base set in similar way as applied in [13]. SIFT descriptors are already divided into training, base and query set.

To binarize the datasets, we subtract the mean of training set with the input code, select a set of coefficients from

TABLE III
DATASET CONFIGURATIONS.

Dataset	# training	# base	# query
SIFT10K	25,000	10,000	100
SIFT1M	100,000	1,000,000	10,000
GIST10K	300,000	10,000	1,000
GIST100K	300,000	100,000	1,000
GIST1M	300,000	1,000,000	1,000

normal distribution and calculates the dot product between normal coefficients and normalized input, finalizing with the quantization. These generated binary data were saved into a 32GB microSD card which is plugged into the Pynq board and later read by the application.

B. Performance

To analyze the potential performance of the MIH accelerator, we implemented three versions of the MIH kNN search: C (entirely in software), Python (entirely in software) and Python with MIH IP. All versions utilize only one core of the dual-core ARM Cortex-A9 host processor. Since the dataset is binarized to 64-bit binary codes, we also configure the MIH to partition data in 4 and 8 hash tables. Due to the limited resources provided by the given Zynq FPGA chip (xc7z020), we use parts of the datasets: SIFT10K, SIFT1M, GIST10K, GIST100K and GIST1M. SIFT10K and SIFT1M are already available on the SIFT dataset. For the GIST dataset, we randomly pick 10K, 100K and 1M points from the binary base set corresponding to GIST10K, GIST100K and GIST1M, respectively. Table III summarizes the dataset configurations we use to measure the performance.

For each configuration, we obtain the mean of 10 runs to calculate the speedups. Figure 6 shows the performance of the MIH accelerator compared to Python code running on the ARM host processor. For all datasets, the MIH accelerator achieves better results than the corresponding Python version. The number of hash tables in MIH influences the distribution of indexes that are sent to the accelerator and the number of radius searches to find the k nearest neighbors. In smaller datasets, the average number of indexes submitted to MIH is less than 64. Since the co-processor uses a sorting network, as presented in Section III-C, even receiving less than 64 data, it takes 21 cycles to complete the ordering. The MIH configuration with 8 hash tables obtained best results with the best case accelerating up to 33 times.

Figure 7 presents the performance of the MIH accelerator compared to C code running on the ARM host processor. Smaller datasets result in slowdown, because the co-processor takes more time (21 cycles) to sort small list of indexes. Furthermore, the data distribution in 8 hash tables configuration create many small lists of indexes for each hash table location, so that the C version also takes advantage of the cache hit. The best results were obtained with 4 hashes tables, with the best case presenting approximately 19 times speedup.

The performance analysis shows that the MIH accelerator potentially improves the performance of kNN search applications, especially considering that the Zynq ARM processor operates at 667 MHz, while the accelerator operates at 100 MHz. The bottleneck of MIH accelerator is strictly related to the number of indexes stored in each hash table location which impact the partial sorting operation. A large amount indexes result in great speedup. As the experiments were performed using a single accelerator for all hash tables due to the limited number of resources available on the Zynq FPGA, the results could be improved if one dedicated MIH accelerator could be instantiated for each hash table.

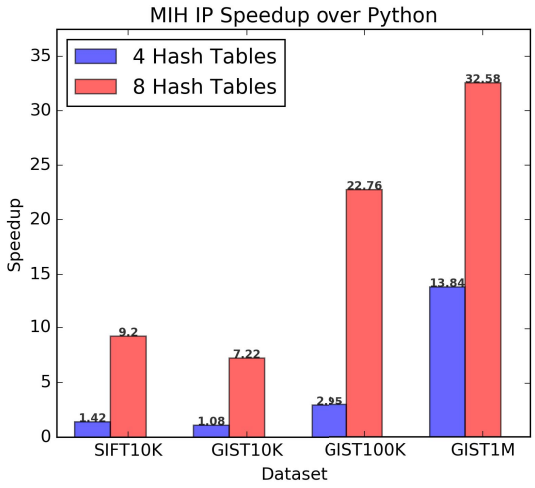

Fig. 6. Speedup of MIH IP compared to Python version.

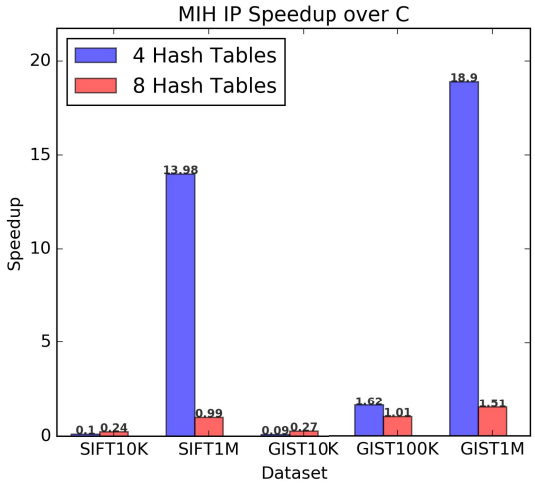

Fig. 7. Speedup of MIH IP compared to C version.

C. Utilization Cost

This section presents the results regarding resource consumption on the implementation of the Hamming distance and sorting algorithm on the Zynq FPGA. The values of the Direct

TABLE IV
APPLICATION UTILIZATION ON THE PYNQ BOARD.

Resource	Utilization	Available	Utilization %
LUT	47228	53200	88.77
LUTRAM	1934	17400	11.11
FF	61564	106400	57.86
BRAM	18	140	12.86
BUFG	1	32	3.13

Memory Access (DMA) are also taken into account, since it is needed due to Axi-Stream communication protocol.

Table IV presents the resource consumption divided into Look-Up Tables(LUTs), Look-Up Tables RAMs Flip-Flops(FF), Block RAMs (BRAM) and BUFG which is responsible for generating clock. We can see that FF is the most used resources, not in percentage, but in raw value. This is due to using multiple pipelines, forcing data to be stored in small register-like variables, which in may have been mapped onto Flip-Flops. The BRAM resources amounted to 12.86% utilization and are most likely due to store multiple 64-bit values for the partial sorting component. LUTs are the most used resources ratio-wise achieving 88.77% as there are many logical and arithmetic operations occurring in parallel as seen on Figures 2 and 3, which means the hardware needs to be replicated instead of reused. Also, the LUT resource is the main building block of several architecture components.

D. Power

This section presents power requirement estimations generated by Vivado power analysis report. The analysis was executed on *vectorless* mode with a default toggle rate set to 12.5% and static probability set to 0.5. Although the *vectorless* power analysis is not accurate, it gives a reasonable estimation about the overall circuit power consumption.

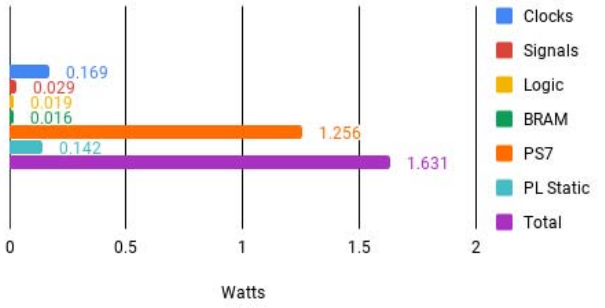

Fig. 8. Vivado 2018.1 Power consumption report based on *vectorless* mode, with a default toggle rate set to 12.5% and static probability set to 0.5.

Figure 8 displays the average power consumption in Watts, separated by FPGA components. Total consumption is equivalent to 1.63 Wats per cycle, which can be considered small when compared to common general purpose ASICs. PS7, which is the processing system (ARM), amounts for the most consumption overall, while other components have a very small impact.

V. DISCUSSION

As the main goal of this work is to provide a design of hardware accelerator to improve kNN performance using MIH structure, we identify that our accelerator is not limited to this kind of structure. MIH is one type of Inverted Multi-Index (IMI) structure which has been widely researched to provide efficiency in similarity search applications [24]–[26], where the key idea is to replace the standard quantization techniques by inverted indices with product quantization [24]. Our accelerator can be easily modified to support the variations of this structure such that Distance-Computation-Free search proposed in [26]. Eghbali *et al.* have elaborated Hamming Weight Tree (HWT) structure [27] which builds a tree structure to map the query with bucket represented as a leaf, containing similar neighbors. Even as in MIH, HWT might be used for kNN search and consists of calculating HD for each candidate into the leaf and sort them to find the top-k most similar neighbors among the leaves. The MIH co-processor can be straightforwardly applied to work with HWT structure.

Moreover, our MIH accelerator supports fixed configurations in term of number of candidates to return and to sort by using the sorting network component. We setup to use 64 items to be able to run the empirical experiments discussed in Section IV-B. Typically, the kNN search uses 100-NN until 1000-NN neighbors parameters. To fit these parameters embedded in the sorting network is impracticable due to resource constraints. Kobayashi *et al.* have proposed a sorting accelerator that combines Sorting Networking and Merge Sorter Tree algorithms with a data compression mechanism [28]. Upgrading our sorting network component with their sorting accelerator is planned to future work as we believe that would be able to increase the number of parameters enabling 1000-NN search in our hardware design.

Overall the MIH co-processor has the potential to accelerate kNN search consuming around $0,375$ Watts (excluding the ARM processor) and running at 100 MHz.

VI. CONCLUSION

Binary compression techniques have been proposed to compact the high-dimensional data into binary codes in order to keep acceptable accuracy when performing kNN search applications. Multi-Index Hashing provides fast indexing solution by mapping database indexes m times into m hash tables, which enables the search for binary codes in sub-linear runtime in Hamming space. Its data-intensive tasks are focused in Hamming distance calculation and the ordering over neighbor candidates from each hash table. As MIH is based on binary data, kNN search mechanisms can be easily enabled on IoT environments through MIH structure providing a simple and efficient implementation on hamming space.

In this work, we proposed a MIH hardware co-processor that rely on specialized Hamming Distance calculation and Odd-even Merge Sort Network circuits. We discussed the FPGA implementation by using AMBA4 AXI Stream protocol and showed the synthesis results. We evaluated the proposed hardware running at 100 MHz frequency and reported that it

can achieve interesting speedup in comparison with embedded ARM processor when the dataset is large. The complete co-processor occupies around 88% of the total Look-Up Tables and has low power consumption (around 0.375 Watts, excluding the ARM processor) in a small-sized, low-cost FPGA. The power consumption estimation is based on Vivado's *vectorless* mode, with a default toggle rate set to 12.5% and static probability set to 0.5. In the future, we intend to meticulously measure the architecture's power consumption, in order to get more precise power consumption results.

Finally, it is important to emphasize that although the proposed MIH accelerator was implemented and evaluated using a FPGA chip, it could also be implemented as an Application-Specific Integrated Circuit, especially because its RTL architecture is already specified in Verilog hardware description language.

ACKNOWLEDGMENT

This study was financed in part by the Coodernao de Aperfeioamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001 and CNPq.

REFERENCES

[1] D. P. Kaz Sato, Cliff Young, "An in-depth look at Google's first Tensor Processing Unit (TPU)," https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, 2017, [Online; accessed 12-October-2018].

[2] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 1–14.

[3] "ARM Project Trillium," https://www.arm.com/products/silicon-ip-cpu/machine-learning/project-trillium, 2018, [Online; accessed 12-October-2018].

[4] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 11, pp. 1958–1970, Nov. 2008. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2008.128

[5] M. K. Alsmadi, "An efficient similarity measure for content based image retrieval using memetic algorithm," *Egyptian Journal of Basic and Applied Sciences*, vol. 4, no. 2, pp. 112 – 122, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2314808X16300628

[6] V. T. Lee, A. Mazumdar, C. C. del Mundo, A. Alaghi, L. Ceze, and M. Oskin, "Application codesign of near-data processing for similarity search," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, vol. 00, May 2018, pp. 896–907. [Online]. Available: doi.ieeecomputersociety.org/10.1109/IPDPS.2018.00099

[7] P. Li, M. Wang, J. Cheng, C. Xu, and H. Lu, "Spectral hashing with semantically consistent graph for image indexing," *IEEE Transactions on Multimedia*, vol. 15, no. 1, pp. 141–152, Jan 2013.

[8] Y. Lv, W. W. Y. Ng, Z. Zeng, D. S. Yeung, and P. P. K. Chan, "Asymmetric cyclical hashing for large scale image retrieval," *IEEE Transactions on Multimedia*, vol. 17, no. 8, pp. 1225–1235, Aug 2015.

[9] J. Song, Y. Yang, Z. Huang, H. T. Shen, and R. Hong, "Multiple feature hashing for real-time large scale near-duplicate video retrieval," in *Proceedings of the 19th ACM International Conference on Multimedia*, ser. MM '11. New York, NY, USA: ACM, 2011, pp. 423–432. [Online]. Available: http://doi.acm.org/10.1145/2072298.2072354

[10] J. Wang, S. Kumar, and S. Chang, "Semi-supervised hashing for scalable image retrieval," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, June 2010, pp. 3424–3431.

[11] D. Zhang, J. Wang, D. Cai, and J. Lu, "Self-taught hashing for fast similarity search," in *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '10. New York, NY, USA: ACM, 2010, pp. 18–25. [Online]. Available: http://doi.acm.org/10.1145/1835449.1835455

[12] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," 1998, pp. 604–613.

[13] M. Norouzi, A. Punjani, and D. J. Fleet, "Fast exact search in hamming space with multi-index hashing," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, vol. 36, no. 6, pp. 1107–1119, June 2014. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TPAMI.2013.231

[14] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration." in *VISAPP (1)*, A. Ranchordas and H. Arajo, Eds. INSTICC Press, 2009, pp. 331–340.

[15] C. Silpa-Anan and R. Hartley, "Optimised kd-trees for fast image descriptor matching," in *2008 IEEE Conference on Computer Vision and Pattern Recognition*, June 2008, pp. 1–8.

[16] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 950–961. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325958

[17] M. M. Esmaeili, R. K. Ward, and M. Fatourechi, "A fast approximate nearest neighbor search algorithm in the hamming space," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 12, pp. 2481–2488, Dec 2012.

[18] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, "Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 12, pp. 2916–2929, Dec 2013.

[19] E. P. Xing, A. Y. Ng, M. I. Jordan, and S. Russell, "Distance metric learning, with application to clustering with side-information," in *Proceedings of the 15th International Conference on Neural Information Processing Systems*, ser. NIPS'02. Cambridge, MA, USA: MIT Press, 2002, pp. 521–528. [Online]. Available: http://dl.acm.org/citation.cfm?id=2968618.2968683

[20] A. Dalalah, S. Baba, and A. Tubaishat, "New hardware architecture for bit-counting," in *Proceedings of the 5th WSEAS International Conference on Applied Computer Science*, ser. ACOS'06. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2006, pp. 118–128. [Online]. Available: http://dl.acm.org/citation.cfm?id=1973598.1973623

[21] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on fpgas," *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1007/s00778-011-0232-z

[22] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," *CoRR*, vol. abs/1102.3828, 2011. [Online]. Available: http://arxiv.org/abs/1102.3828

[23] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 2002, pp. 380–388.

[24] A. Babenko and V. Lempitsky, "The inverted multi-index," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 6, pp. 1247–1260, June 2015.

[25] X. Lin, Y. Shen, L. Cai, and R. Ji, "The distributed system for inverted multi-index visual retrieval," *Neurocomputing*, vol. 215, pp. 241 – 249, 2016, sI: Stereo Data. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231216306427

[26] J. Song, H. T. Shen, J. Wang, Z. Huang, N. Sebe, and J. Wang, "A distance-computation-free search scheme for binary code databases," *IEEE Transactions on Multimedia*, vol. 18, no. 3, pp. 484–495, March 2016.

[27] S. Eghbali, H. Ashtiani, and L. Tahvildari, "Online nearest neighbor search in binary space," in *2017 IEEE International Conference on Data Mining (ICDM)*, Nov 2017, pp. 853–858.

[28] R. KOBAYASHI and K. KISE, "A high performance fpga-based sorting accelerator with a data compression mechanism," *IEICE Transactions on Information and Systems*, vol. E100.D, no. 5, pp. 1003–1015, 2017.