Approximate Similarity Search with FAISS framework using FPGAs on the cloud

Dimitrios Danopoulos
Department of Electrical
and Computer Engineering
NTUA, Athens, Greece

Christoforos Kachris
Democritus University of Thrace
& ICCS-NTUA
Athens, Greece

Dimitrios Soudris
Department of Electrical
and Computer Engineering
NTUA, Athens, Greece

***Abstract*—Machine Learning algorithms, such as classification and clustering techniques, have gained significant traction over the last years because they are vital to many real-world problems. K-Nearest Neighbor algorithm (KNN) is widely used in text categorization, predictive analysis, data mining etc. but comes at the cost of high computation. In the era of big data, modern data centers adopt this specific algorithm with approximate techniques to compute demanding workloads every day. However, high dimensional nearest neighbor queries on billion-scale datasets still produce a significant computational and energy overhead. In this paper, we describe and implement a novel design to address this problem based on a hardware accelerated approximate KNN algorithm built upon FAISS framework (Facebook Artificial Intelligence Similarity Search) using FPGA-OpenCL platforms on the cloud. This is an original deployment of FPGA architecture on this framework that also shows how the persistent index build times on big scale inputs for similarity search can be handled in hardware and even outperform other high performance systems. The experiments were done on AWS cloud F1 instance achieving 98× FPGA accelerator speed-up over single-core CPU and 2.1× end-to-end system speed-up over a 36-thread Xeon CPU. Also, the performance/watt of the design was 3.5× from the same CPU and 1.2× from a Kepler-class GPU.**

***Index Terms*—big-data, similarity search, approximate KNN, cloud, FPGA, reconfigurable computing**

I. Introduction

The new age of big data requires trillion or even quintillion bytes of data to be processed every day. Emerging applications on the cloud are constantly learning from large scale data in all kinds of real world problems, such as data analytics and internet media. This large computational complexity motivates efforts to accelerate these tasks using hardware-specific optimizations by leveraging different architectures such as CPUs, GPUs, FPGAs. The increasingly growing demands for efficient and fast processing in the recent years can be addressed by heterogeneous computing systems such as Field Programmable Gate Arrays (FPGAs). This architecture is highly parallelizable and re-configurable and can be mapped well to classification or clustering problems, which are usually repetitive, such as KNN. Specifically, data centers tend to search for efficient ways for high performance and low energy cost solutions, and FPGAs play a mojor role in this evolution [1], [2].

Similarity search in data centers is characterized by large scale datasets and many times high dimensional inputs. Even very modern high performance computers cannot handle these requests resulting in a computer phenomenon known as the *curse of dimensionality* [3]. Approximate nearest neighbor algorithm (ANN) overcomes the high-dimensional problem that usually comes with arithmetic complexity and/or high data bandwidth demands [4]. In that way, the extremely intensive exhaustive distance calculation of the basic KNN algorithm become a less time and energy consuming task by implementing a partial query search. Exact results are impractical on billion-sized databases on the cloud so new techniques that index or compress the vectors with quantization methods have been introduced [5]. FAISS, an optimized library for efficient similarity search produced by Facebook [6], contains algorithms that can search in sets of vectors of any size using approximate methods. Nevertheless, the index building algorithms used for implementing KNN graphs are time consuming and therefore energy inefficient and often cannot readily scale to the billion-sized databases on the cloud especially when frequent re-training is needed due to statistically different data.

Asides from the software approaches, an underlying hardware solution is necessary to boost the performance on these kind of tasks. There is a lot of interest in using hardware accelerators such as FPGAs, for this set of problems. With the increasing popularity of data classification recently, FPGAs as parallel implementation platforms are very suitable as they can be programmed in such a way in order to reduce the total computational overhead. They provide fine grain programmable hardware resources with high performance and much lower power compared to multi-core systems. Moreover, as the energy footprint in big data centers that operate today on large-scale KNN tasks is crucial, the problem becomes challenging. In this paper, we present a novel implementation of FAISS framework using FPGAs and achieve a significant speed-up compared with other multi-core solutions. More specifically, in this work we make the following contributions:

- We implement a partition-based FPGA accelerator on Xilinx VU9P in order to speed-up the index building time for approximate KNN search used on FAISS.
- We make several dataflow and memory optimizations on FAISS framework in order to match and benefit from the FPGA hardware that we integrated.
- We ported and ran the full hardware accelerated framework on AWS cloud, achieving superior performance and power efficiency to other high performance systems.

II. Previous Work

A lot of research interest has been shown on the KNN algorithm especially for the acceleration and optimization both in software and hardware. Moreover, energy efficiency, the crucial criterion nowdays which concerns the big companies has not shown the equivalent importance. And as the big-scale data is processed on the cloud, developers need to find clever heterogenous architectures to port these applications.

Yuliang Pu et al. [7] presented a bubble sort enhanced KNN algorithm using the FPGA based computing system. This implementation focuses on the query search, however the algorithm is based on the exhaustive naive KNN implementation that lacks performance when compared with an approximate solution. Even the proposed FPGA accelerated solution of that work cannot surpass the performance of FAISS running in a traditional CPU. This is because FAISS is an already CPU/GPU optimized library for approximated similarity search which achieves fast query results while maintaining generally negligible reduction in accuracy.

Jialiang Zhang et al. [8] presented a method for PQ based approximated nearest neighbor search using OpenCL FPGAs. They focus on the codebook size reduction in order to limit the memory overhead over the dataset when doing query search. However, the cost of the training/clustering, which is a crucial part of the approximate KNN graph construction, is not reflected in the measurements. Secondly, they do not describe specifically the hardware implementation/architecture of the accelerator and they only show a part of the algorithm in pseudocode. Last but not least, their proposed solution does not proceed to an implementation of FPGAs on the cloud.

Hanaa M. Hussain at al. [9] present a K-means clustering method in FPGAs for processing large scale datasets. However, their implementation is inferior to this work's partition-based accelerator in terms of performance. They state that they achieve $0.0042\ secs/iteration$ for $N = 65500$ dataset with $K = 4$ clusters and $D = 9$ dimension. K-means, in the world of algorithm comlexity has an $O(n \cdot d \cdot i \cdot k)$ complexity so this translates in ~ 0.56 GFLOPs which is proportionately (of our FPGA resources) smaller than the performance we achieve in this work as we will see in the next chapters.

III. Background

In this section we describe how classic KNN search and ANN search on FAISS work. Different approximation techniques are used in the framework based on IVF (inverted indexing) methods in order to cluster dense vectors but in this work we chose to focus on the *IVFFlat* optimization because it is generally the fastest and more accurate technique used in the framework. Furthermore, we introduce the FPGA architecture and the environment used to develop the application dataflow and kernels.

A. KNN Search

KNN algorithm is one of the most popular machine learning algorithms used in pattern recognition scenarios, recommender systems and even financial research [9], [10]. The function returns the K nearest neighbor points to a specified object among a dataset. The "weights" of each object are defined by *D* attributes which is the data dimension. The plurality vote of this algorithm is often characterized by the distance between the query points. A commonly used distance is the *Euclidean distance* because very often it has intuitive meaning [11] and the computation scales. For discrete variables the overlap metric is used which is also known as *Hamming distance*. In Cartesian coordinates the Euclidean *n-space* is the following:

$$dist(x_i, y_i) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

The exhaustive query search using the above algorithm, especially for large-scale datasets, becomes quite high in terms of number of operations as we need to compute each distance between every point in the samples. So approximate solutions need to take place in order to find effectively the K most possible nearest neighbors.

B. ANN Search

The idea behind this approach is that, in many cases, an approximate nearest neighbor is almost as good as the exact one because small differences in the distance should not matter. This makes KNN computationally tractable even for large datasets as they reduce the number of distance evaluations performed in total. There are mainly two types of methods in ANN search that focus either on data and dimension reduction (or both) for the query search. One type is spatial clustering based method, such as K-means clustering [12] or hierarchical KD-trees [13]. These methods employ a partition algorithm technique by constructing a k-nearest neighbor graph via splitting-clustering the samples into regions. Another method for ANN search is hashing based methods such as Locality Sensitive Hashing (LSH) [14] which groups the points into "buckets" based on the distance metric. Near vectors are mapped into the same bucket and thus the algorithm retrieves the closest neighbors of the target vector.

C. FAISS framework operation

Faiss contains several methods for similarity search on dense vectors of real or integer number values and can be compared with L2 distances or dot products. Vectors that are similar-close to a query vector are those that have the lowest L2 distance or equivalently the highest dot product with the target-query vector. The framework is built upon different indexing schemes for storing vectors and the distance function is calculated using severel metrics such as the previously mentioned. The indexing structures differ from each other, ranging from exact search to product quantization techniques which are shown to be more effective than binary codes. This, of course, comes at a cost of accuracy but also determines the trade-off between other important measurements such as speed of search time, training time, memory, etc. In this subsection we will analyze some basic indexing techniques in FAISS so as to have an overall understanding of the framework architecture

before proceeding to choose which one to accelerate via the FPGA hardware. Our implementation, described on the next section, chooses IVFFlat indexing as a use case for demonstration (best for high-accuracy regimes) but the design can be easily applied to other indexes such as IndexIVFPQ. Both methods, especially the latter, have a little less precision than exaustive search but are proven to scale to billions of vectors in sufficient a memory on a single server.

1) IVFFlat indexing: Several studies have exploited the properties of *Voronoi* diagrams to improve variations of the nearest neighbor search [15]. Voronoi diagram is a partitioning of a plane into regions based on distance to points in a specific subset of the plane [16]. FAISS constructs the *IndexIVFFlat* index by defining Voronoi cells from a codebook C_{coarse} in the d-dimensional space, and each database vector falls in one of the cells. At search time, only the database vectors y contained in the cell that the query x is mapped to along with a few neighboring ones are compared against the query vector. So, two parameters are essential to the query process method: *ncells*, the number of cells, and *nprobe*, the number of cells (out of *ncells*) that are visited to perform a search. The number of cells stands for the number of inverted lists which might be denoted also as $nlist$. This probing process is a partition-based method based on Multi-probing (a reminiscent variant of best-bin KD-tree [17]). The database vectors are assigned to the cells thanks to a hashing function, specifically K-means (closest query to centroid) and stored in an inverted file structure. As a result, IVFFlat effectively reduces the search space, and achieves a substantial speed-up over the exhaustive search. By all means, in order the results to be accurate, they must fall into the visiting Voronoi cells.

2) IVFPQ indexing: This generalizes the inverted index idea by combining inverted indices with product quantization so as to avoid exhaustive search. The natural product quantization method is denoted as PQ and its non-exhaustive version is denoted as IVFPQ [18]. The vectors are still stored in Voronoi cells, but their size is reduced to a configurable number of bytes *m* (dimension must be a multiple of m). The compression is an additional level of quantization, that is applied on sub-vectors of the vectors to encode. In this case, since the vectors are not stored exactly, the distances that are returned by the search method are also approximations and often have less accuracy than IVFFlat for example. Also, if this indexing has to handle uniform data, it is very hard because there is no regularity that can be exploited to cluster or reduce dimensionality. Nevertheless, IndexIVFPQ is a very useful indexing structure for large-scale search which can also be deployed into our FPGA design.

3) LSH method: Another very popular cell-probe method is probably the original Locality Sensitive Hashing method (LSH). It reduces the dimensionality of high-dimensional data by hashing input items into similar buckets. Points that are close to each other under a specific distance metric (i.e. Euclidean) are mapped to the same bucket with high probability. Faiss implements this algorithm, however, it lacks some characteristics that are found in the other algorithms such as memory optimizations. The large number of hash functions leads to extra memory, something which is prohibitive for big-scale datasets on the cloud and thus not ideal for this work.

D. FPGA OpenCL framework

The OpenCL specification in FPGAs consists of host code and kernels where host lies on a x86-64 CPU which handles the application's dataflow. Our FAISS application which runs on the host leverages the hardware kernels that translate the OpenCL hardware abstraction to an FPGA implementation on the device fabric. For this task we utilized the SDAccel environment [19] which offers software and hardware emulation capabilities to verify and debug the application before synthesizing and running on the FPGA device. Then, by analyzing the system reports and checking the device data tracing and application timeline for potential bottlenecks in our design up to the precision of clock cycles we could finalize the optimized application with the hardware integrated into FAISS framework. Cloud service providers, such as Amazon AWS, who specialize in heterogeneous accelerator clouds for big data and machine learning offer FPGA-based boards and with this way we deployed our application on the cloud.

IV. Implementation

In this section we first describe the framework analysis in order to profile FAISS and specify which parts of the algorithm are going to be implemented on hardware, selecting IVFFlat as a use-case for profiling. Next, we represent the new framework which uses an in-memory FPGA format for minimum latency overhead over the memory transactions. Last, we specify the hardware accelerator and all the optimizations done on the host and kernel side so as to achieve maximum throughput and reduce the overall latency of the design.

A. Framework profiling

We first had to select an efficient ANN implementation, so we identified from previous work and ours that the IVF indices, especially IVFFlat, perform the best among the other indexing techniques used in FAISS. In this algorithm, using several profiling tools (i.e. valgrind/callgrind , gprof), we determined the memory and compute bottlenecks with the use of call-graph techniques. The functions that consume the most execution time are good candidates to be offloaded and accelerated into FPGAs (see Figure 1). Our work shows that index creation for the dataset consumes a lot of time, even orders of magnitude from a modest query search of thousands of vectors. This comes from the fact that ANN search is very agile but comes with a cost of long training-clustering times. Particularly, the training algorithm contains a lot of MultiplyAccumulate operations (MAC) which are implemented with optimized BLAS routines on the default FAISS CPU implementation. These algorithms are good candidates for hardware parallelization because they have a high number of $\frac{total\ operations}{bytes\ transferred}$.

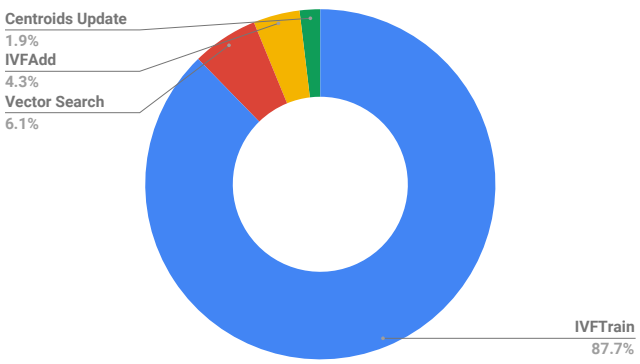

Fig. 1. FAISS profiling with IVFFlat indexing

B. Optimization schemes

For an efficient hardware implementation we had to take into account specific principles that comprise the design methodgy for FPGAs. These include as a first step to design the host and kernel side of the accelerator function so as to map well to the FAISS framework and at the same time keep the correctness of the application. Furthermore, concerning the memory, optimizing data movement is crucial between the host from/to global memory and global memory from/to kernels [20], [21]. Last, we created an optimized custom logic for the FPGA kernels to achieve a low latency design.

1) Defining the Accelerator: They key transformations for high level synthesis start from the host code. In order to define a scalable function for acceleration we implemented a custom column-major GEMM (General Matrix Multiplication) routine where the first matrix is transposed:
$C = alpha * T(A) * B + beta * C$
In this way, more efficient hardware is implemented accessing both arrays with consecutive elements in the 2nd dimension as the data is stored. Also, it is essential to know before-hand if possible, how large the dimensions of the matrices are going to be. In our case, FAISS defines the first input of GEMM as an $nlist \times vector\_dim$ matrix while the second one as a $vector\_dim \times centroid\_points$. The integer $nlist$ is usually small (compared with the database) and depicts the number of inverted lists as stated previously. As a rule of thumb this number is usually multiples of $\sqrt{dataset}$ (i.e. IVFFlat4096, etc). Furthermore, the $vector\_dim$ which is the dimension of the vectors, is not a large number as well (below one thousand) because compressing is used for higher data dimensions. The larger matrix dimension is $centroid\_points$ which are defined as the points to sample for clustering each time. So our design will be based on the above facts for maximum optimization.

2) Optimizing data movement: An important thing we noticed is that GEMM runs multiple times in each iteration of the training (iterations are usually no more than 20-25 as little improvement is done) accessing blocks of the B matrix. We know from theory that each iteration uses the same clustering data (total centroid points) over training so we conclude that B matrix is reused over each iteration. In order to avoid the same data to be transferred to global memory during each iteration we transferred the total number of centroid points to DDR one time only in the first iteration. In this way, we avoided memory requests of the same data and also achieved faster burst data transfers to global memory as bigger chunks of memory are transferred in higher rates. Furthermore, to maximize data throughput we implemented a 512-bit user interface on each kernel side which is the maximum memory bandwidth supported in our FPGA using OpenCL vector datatypes. Particularly, we used *float16* datatype in order to maintain maximum accuracy for the dataset and not append another layer of approximation into ANN search. Along with the use of all four DDRs of the device for read-write operations we achieved optimal memory transfer rates between host-DDRs and DDRs-kernels. It is worth mentioning that careful selection of SLR assignment to kernels was took into account. Our design was placed in a way that did not exceed SLR resources and kernels were put in the same SLR as the memory banks with which they had the most connections. This avoids SLR crossing and limits critical paths, something that leads to inefficient synthesis and more power consumption. Last, we allocated matrix blocks in physically contiguous memory using on-chip BRAMs, which are physically located near the computation of kernels. Hence, we guaranteed a fast communication allowing one-cycle read-writes of 512 bit data with the most efficient data movers.

3) Optimizing Kernel: In order to achieve large throughput we had to enable a high degree of fine-grained parallelism in application execution within the PL fabric. We avoided data dependencies and increased the level of parallelism in the algorithm of the hardware kernels. Using appropriate OpenCL directives such as pipeline or unroll, we constructed a highly parallel and pipelined architecture with minimum latency that performed the MAC operations very efficiently. We achieved initiation interval II=1 in every loop and by using the dataflow directive the kernel consumed all the data as soon as they arrived at memory interfaces and wrote back to DDRs as soon as they were ready. Also, it was more efficient to create larger and fewer compute units (CU) to access chunks of global memory than creating multiple smaller CUs because this results in absurdly heavier usage of FPGA resources and area due to which design fails timing. The design for a single kernel fully parallelizes up to 192 output data, thus in every cycle it produces 192 output elements. This translates to:

$$2 \cdot 192 \cdot kernel\_freq \text{ FLOPs}$$

C. Final System Design

In order to integrate the hardware accelerator with FAISS and port it into the FPGA on the cloud we had to export it as a shared library and then link it with the rest of the FAISS framework. Through the SDAccel Development Environment we were able not only to create the dynamic library but also the final FPGA binary that the device would load from. We made specific modifications on the Makefile of FAISS so as to direct the SDAccel compiler to successfully compile the whole framework using our in-memory FPGA OpenCL model.

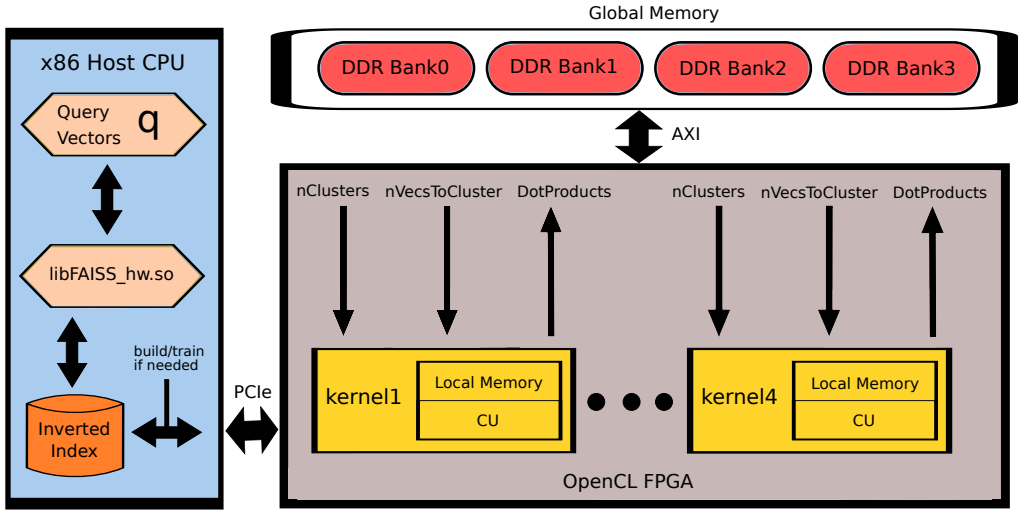

Fig. 2. Software and Hardware Dataflow

As a result, all cluster points resided in FPGA global memory the whole time and our hardware function accessed part of the global memory when needed. The already optimized CPU BLAS function called "sgemm" was replaced by our custom function which manipulates carefully all the input data with OpenCL task synchronization via command queues on the host side. The general application dataflow of FAISS can be observed on Figure 2. To conclude, host code optimization (concurrency from OpenCL Command Queue), buffer management regarding data exchange between the host and kernels, general pipelining on the FPGA, and synchronization between host and kernels were all precisely constructed according to FPGA design principles for high performance.

V. Experiments

In order to evaluate the design we confirmed the correctness of the accelerator and measured its performance. Then after the integration with FAISS we tested the final system's accuracy and performance on real-world data and measured the power efficiency compared with other systems such as CPU, GPU.

A. System setup

The mapping of the final system was done on cloud FPGAs based on the Amazon AWS F1 platform. This consists of a Xilinx Virtex Ultrascale+ VU9P Acceleration development board with four DDR4 channels along with a host system of 8 vCPUs and 122GiB RAM. For a fair end-to-end comparison with a high performance CPU we chose a c4.8xlarge instance which comes with 36-vCPU Xeon and 60 GiB of RAM, and has the same price tag (per hour) as an f1.2xlarge ($\sim 1.6/h$). Also, for the final system evaluation we compare the perfomance/watt of the same CPU and a Kepler-class K40 GPU. Our FPGA hardware design uses all DDRs and makes good use of each SLR's resources with routing across the three SLRs being the primary bottleneck preventing further scaling. Table I shows the resource utilization of a single kernel on the FPGA device.

TABLE I
FPGA Resource Utilization per Kernel

Utilization summary				
Name	***BRAM***	***DSP***	***FF***	***LUT***
Total	502	1004	157131	89096
Percentage (%)	11	14	6	7

B. Accelerator Performance

First, the proposed accelerator was verified for correctness and evaluated on AWS F1 instance. In order to test properly the hardware function, we simulated the FAISS inputs for different dataset scenarios, generating random cluster data for a number of Voronoi cells. On Figure 3 we observe the FPGA accelerator efficiency measured in percentage of GFLOPs for a number of Voronoi cells.

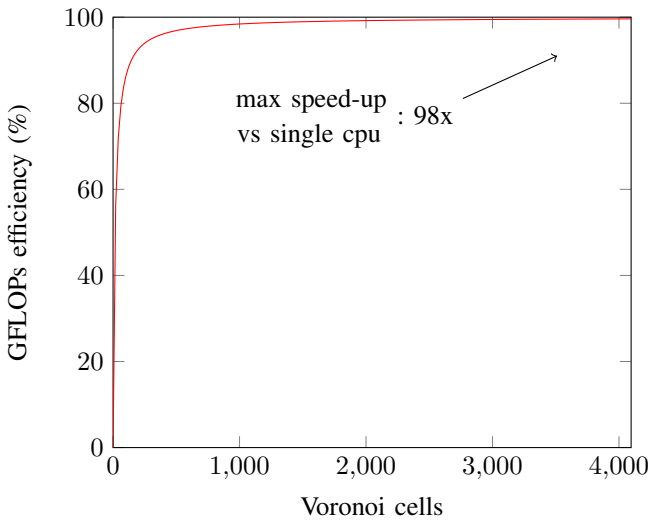

Fig. 3. Hardware accelerator efficiency

The arrow shows the maximum speed-up achieved when compared with a single-core CPU. This comes from the fact that for more cells the impact of data transfer is less evident. It's worth mentioning here that the lower efficiency value which happens to be from $\sim$ 500 cells and below does not impact the overall performance of the algorithm. Usually in real-world datasets, especially in larger ones used in data centers, the Voronoi cells are multiples of thousands for satisfactory clustering, even for a modest 1-million dataset.

C. Final System and Evaluation

For the final system performance evaluation we first compared the end-to-end execution with the 36-thread Xeon CPU mentioned previously achieving $\sim 2.1\times$ speed-up. Next, for the purposes of demonstration we used a million-scale dataset, specifically the SIFT 1M which is a well-known dataset and can fit easily in any RAM. In Figure 4 we make an accuracy evaluation on this dataset with our FPGA design in order to make an assessment of the quality of our algorithm on actual real-world data. KNN models are usually evaluated using the *recall* measure which is the ratio of correctly predicted positive observations to all observations in the actual class. However, for the quality of our custom Inverted Index method we use the "$R-recall\ at\ R$" also known as *intersection*, which the fraction of the R found nearest neighbors that are within the ground-truth R nearest neighbors (we use R=100).

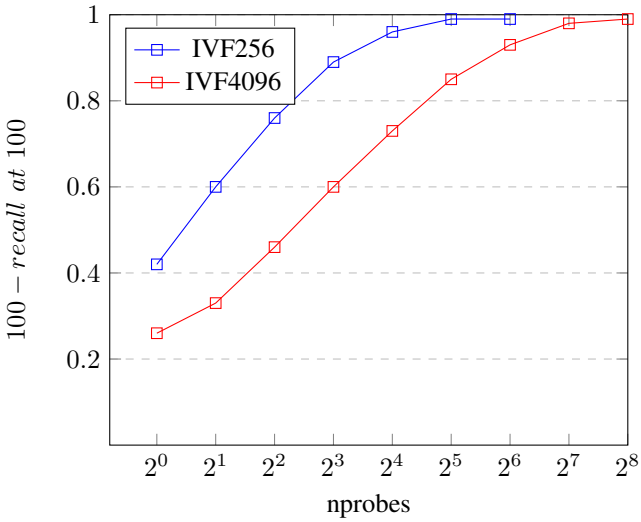

Fig. 4. Inverted Index accuracy for different probe values on SIFT 1M

In the above figure, we evaluate the accuracy of two Inverted Index (Flat) methods using different probe values. The construction of the IVF4096 index takes much longer as we have a larger number of cells to train (4096 vs 256). However, for the same accuracy values between each method, the query search is much faster on the IVF4096. This is because a smaller fraction of the database is compared to the query ($nprobe/ncells$). For example in order to maintain 0.6 accuracy we have to probe $\frac{2}{256}$ cells on IVF256 whilst we select $\frac{8}{4096}$ cells on IVF256.

Next, we proceeded to make a live measurement of our FPGA board on the cloud using AWS metric tools. We then compared the performance/watt of our hardware design with the maximum theoretical performance/watt of the Xeon CPU and a K40 GPU. The results are illustrated in Figure 5 with a clear advantage over our FPGA architecture. We used the following equation (1) for the other devices in order to make a comparison of the maximum power efficiency they can achieve compared with our FPGA design.

$$power\ efficiency = \frac{device\ FLOPS}{device\ TDP} \quad (1)$$

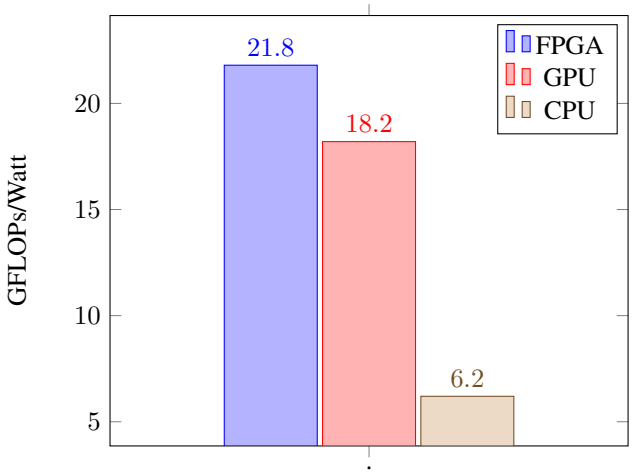

Fig. 5. Power Efficiency comparison

VI. Conclusion

Hardware accelerators can improve significantly the performance of machine learning applications. However, many frameworks such as FAISS do not support the transparent utilization of such acceleration modules. In this study, we implemented a novel scheme for the seamless utilization of FPGA hardware in FAISS framework for big-data similarity search used on the cloud.

Our results showed than our hardware accelerator outperforms a 36-thread Xeon CPU and has better performance/watt when compared with the same CPU and also with a Kepler-class GPU. Thus, SW/HW codesign is in fact a valid solution for the cloud computing workloads, in this case for the persistent indexing times of approximated KNN algorithms. The superior performance and performance/watt of our design can leverage the use of FPGA hardware on the cloud and in big data-centers as power efficiency plays a very important role today with the constantly increasing workload demands.

As Future Work, in order to solve the memory issue of billion-scale datasets which was our only restriction, the distributing of the application to a number of FPGAs is needed. Our algorithm was designed in such a way that the host application can easily distribute the dataset and thus the workload on a number of FPGAs on the cloud which due to limited infrastructure at that time we could not accomplish.

VII. Acknowledgment

This project has received funding from the Xilinx University program and the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under grant agreement No 2212-Hardware Acceleration of Machine Learning Applications in the Cloud.

References

[1] C. Kachris and D. Soudris, "A survey on reconfigurable accelerators for cloud computing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–10, Aug 2016.

[2] S. Mavridis, M. Pavlidakis, I. Stamoulias, C. Kozanitis, N. Chrysos, C. Kachris, D. Soudris, and A. Bilas, "Vinetalk: Simplifying software access and sharing of fpgas in datacenters," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Sep. 2017.

[3] N. Kouiroukidis and G. Evangelidis, "The effects of dimensionality curse in high dimensional knn search," in *2011 15th Panhellenic Conference on Informatics*, pp. 41–45, Sep. 2011.

[4] A. Andoni, P. Indyk, and I. P. Razenshteyn, "Approximate nearest neighbor search in high dimensions," *CoRR*, vol. abs/1806.09823, 2018.

[5] M. Norouzi and D. J. Fleet, "Cartesian k-means," in *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, (Washington, DC, USA), pp. 3017–3024, IEEE Computer Society, 2013.

[6] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *arXiv preprint arXiv:1702.08734*, 2017.

[7] Y. Pu, J. Peng, L. Huang, and J. Chen, "An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 167–170, May 2015.

[8] J. Zhang, J. Li, and S. Khoram, "Efficient large-scale approximate nearest neighbor search on opencl fpga," pp. 4924–4932, 06 2018.

[9] H. M. Hussain, K. Benkrid, A. T. Erdogan, and H. Seker, "Highly parameterized k-means clustering on fpgas: Comparative results with gpps and gpus," in *2011 International Conference on Reconfigurable Computing and FPGAs*, pp. 475–480, Nov 2011.

[10] Q. Chen, D. Li, and C. Tang, "Knn matting," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 869–876, June 2012.

[11] S. Liu, X. Liang, L. Liu, X. Shen, J. Yang, C. Xu, L. Lin, , and S. Yan, "Matching-cnn meets knn: Quasi-parametric human parsing," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1419–1427, June 2015.

[12] K. Fukunage and P. M. Narendra, "A branch and bound algorithm for computing k-nearest neighbors," *IEEE Trans. Comput.*, vol. 24, pp. 750–753, July 1975.

[13] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, Sept. 1975.

[14] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), pp. 518–529, Morgan Kaufmann Publishers Inc., 1999.

[15] M. Sharifzadehand and C. Shahabi, "Approximate voronoi cell computation on geometric data streams," 03 2019.

[16] Z. C. Y. L. Kangshun Li, Wei Li, *Computational Intelligence and Intelligent Systems*. Springer Singapor, first ed., 2018.

[17] J. Kybic and I. Vnuko, "Approximate best bin first kd tree all nearest neighbor search with incremental updates," vol. 10, pp. 420–2, 08 2010.

[18] Y. Chen, T. Guan, and C. Wang, "Approximate nearest neighbor search by residual vector quantization," in *Sensors*, 2010.

[19] Xilinx, Inc, "Sdaccel development environment."

[20] D. Danopoulos, C. Kachris, and D. Soudris, "Acceleration of image classification with caffe framework using fpga," in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pp. 1–4, May 2018.

[21] J. Yinger, E. Nurvitadhi, D. Capalija, A. Ling, D. Marr, S. Krishnan, D. Moss, and S. Subhaschandra, "Customizable fpga opencl matrix multiply design template for deep neural networks," in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 259–262, Dec 2017.