FPGA Acceleration of Approximate KNN Indexing on High-Dimensional Vectors

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*—Accurate and efficient Machine Learning algorithms are of vital importance to many problems, especially on classification or clustering tasks. One the most important algorithms used for similarity search is known as K-Nearest Neighbor algorithm (KNN) which is widely adopted for predictive analysis, text categorization, image recognition etc. but comes at the cost of high computation. Large companies that process big data on modern data centers adopt this technique combined with approximations on algorithm level in order to compute critical workloads every second. However, a significant computation and energy overhead is formed further with the high dimensional nearest neighbor queries. In this paper, we deploy a hardware accelerated approximate KNN algorithm built upon FAISS framework (Facebook Artificial Intelligence Similarity Search) using FPGA-OpenCL platforms. The FPGA architecture on this framework addresses the problem of vector indexing on training and adding large-scale high-dimensional data. The proposed solution uses an in-memory FPGA format that outperforms other high performance systems in terms of speed and energy efficiency. The experiments were done on Xilinx Alveo U200 FPGA achieving up to 115× accelerator-only speed-up over single-core CPU and 2.4× end-to-end system speed-up over a 36-thread Xeon CPU. Also, the performance/watt of the design was 4.1× from the same CPU and 1.4× from a Kepler-class GPU.**

***Index Terms*—approximate KNN, nearest neighbor index, machine learning, FPGA, hardware accelerator**

I. Introduction

In the era of big data, modern data centers require a demanding workload of many terabytes of data or more to be processed every day. Emerging machine learning applications on the cloud that are constantly learning from real-world large scale data have shown great advancement. In recent years, this computational complexity motivated efforts to enhance these tasks using hardware-specific optimizations by leveraging different heterogeneous architectures combining CPUs, GPUs, FPGAs. The increasingly growing demands for efficient and fast processing of the new generation algorithms today can be addressed by high performance computing systems such as Field Programmable Gate Arrays (FPGAs). This architecture shows immense parallelization and re-configurability and can be mapped well to repetitive tasks, such as KNN. Specifically, many companies 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].

KNN 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 large workload that casual KNN demands in terms of arithmetic complexity and/or high data bandwidth [4] by implementing a partial query search. However, the optimizations do not focus only on optimizing the query time even though it has a major importance. The new data structures and algorithms used to speed up KNN queries focus on the training of the data points, usually with specific indexing techniques so that thousands of vectors can be searched without accessing all the dataset. This is because exact results are impractical on billion-sized databases so new techniques that index or even compress the vectors with quantization methods have been introduced [5]. The index building algorithms used for implementing KNN graphs are time consuming and therefore energy inefficient and often cannot readily scale to the enormous sizes of input vectors especially when frequent re-training is needed due to statistically different data. FAISS, is an optimized library for this kind of similarity search produced by Facebook [6] and contains approximate algorithms that can handle big-scale inputs.

FPGAs as parallel platforms can reduce the total computational overhead with an underlying hardware solution which boosts the performance on these kind of tasks. An implementation with fine granularity can take advantage of the FPGA hardware resources for high performance and much lower energy footprint, something very crucial for data centers that operate today on these large scale tasks. 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 speed-up the index creation and addition of data points for the approximate KNN search by implementing an FPGA accelerator for Xilinx Alveo U200.
- We introduced an in-memory FPGA model for FAISS framework that can benefit directly from our hardware.
- We successfully integrated and ran the full hardware accelerated framework on a U200 OpenCL-FPGA achieving superior performance and power efficiency to other high performance systems.

II. Previous Work

KNN algorithms have been explored by many researchers especially for the software as well as hardware optimizations that can be applied on this specific task. Additionally, nowadays, big companies investigate new power efficient methods for computation which was of little importance in the past. 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

FAISS framework uses an IVF (inverted indexing) method before doing similarity search by clustering dense vectors. In this section, we describe how a typical KNN and approximate KNN on FAISS work and then we introduce our FPGA implementation and the tools used for developing the environment to host this architecture and application dataflow.

A. Typical KNN

One of the most broadly used machine learning functions for classification scenarios, recommender systems and even financial research is KNN [9], [10]. This algorithm returns the K nearest neighbor points to a specified object (query) among a dataset. Each object is defined by the data dimension D which represents the "weights" or "attributes" of a specific vector. The plurality vote of this algorithm is often characterized by the distance between the query points. *Euclidean distance* is commonly used as the distance metric because very often it has intuitive meaning [11] and the computation scales. Also, textitHamming distance is many times used when then input has discrete variable. 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. Approximate KNN

Practice says that an approximate nearest neighbor is almost as good as the exact one because distance errors are often negligible. This makes KNN search computationally tractable even for large scale datasets and high dimensions as the number of distance evaluations performed in total is greatly reduced due to approximation techniques. 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. 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 order to select which parts of FAISS are worth to be implemented on harwdare, we needed to make a full framework profiling. The first part of this section describes this procedure where IVFFlat index is selected as a use-case for profiling. Next, we specify the algorithm of the hardware function 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. The new custom framework which uses an in-memory FPGA format and integrates our accelerators for minimum latency overhead over the memory transactions is described at the last part of this section.

A. Framework profiling

Previous work and our own experience showed that IVF indices performs well in general in terms of speed and accuracy. Especially IVFFlat, performs the best among the other indexing techniques used in FAISS for typical scenarios. Then, we determined the memory and compute bottlenecks with the use of call-graph techniques that are provided from several profiling tools such as valgrind/callgrind. The functions that consume the most execution time are good candidates to be offloaded and accelerated into FPGAs (see Figure 1). The profiling outcome showed that index creation along with the data addition to the index results for the most computation workload, even orders of magnitude from a modest query search of thousands of vectors. This comes from the fact that approximate nearest neighbor search is very agile but comes with a cost of long training times particularly if there are many cluster points to sample the dataset. It's worth mentioning that the training algorithm containes many MultiplyAccumulate operations (MAC) that FAISS implements them with CPU optimized BLAS routines by default. It is common for these algorithms to be mapped for hardware parallelization as they have a high number of $total\_operations$ per $bytes\_transferred$.

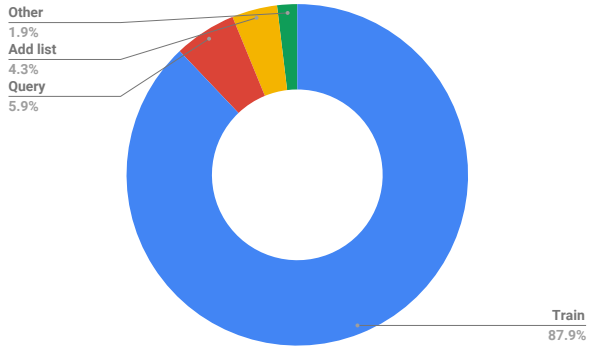

Fig. 1. IVF indexing (Flat) workload distribution

B. Optimization schemes

Specific hardware design principles had to be taken into account for an efficient FPGA implementation of the accelerator. These, as a first step, comprise the design of the x86 host and kernel side of the accelerator function so as to map well to FAISS and at the same time keep the correctness of the application. Moreover, as memory is often characterized as the number one priority, we optimized data movement between the Linux host from/to global memory and global memory from/to kernels [20], [21]. Last, we designed the FPGA kernels so that our custom logic can achieve the lowest latency possible.

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. We then determined the dimensions of the matrices of our function where it is applied, particularly in the index creation and data addition. 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 for maximum optimization our algorithm design and dataflow will be based on these facts.

2) Optimizing data movement: An important thing we noticed is that GEMM runs multiple times in each iteration of the training accessing chunks of the B matrix. Thus, our designed is going to be based on a blocking technique. Also, 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 by avoiding data dependencies. 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. We further generalized our implementation to host multiple FPGAs by distributing evenly the workload and the dataflow is synchronized automatically for any number of kernels. The design for a single kernel fully parallelizes up to 192 output data achieving the maximum of 300 Mhz, thus in every cycle 192 output elements are produced, translating into $2 \cdot 192 \cdot 300$ MHz = 115 GFLOPS.

C. Final System Design

In order to integrate the hardware accelerator with FAISS and port it into the FPGA we had to export it as a shared library and then link it with the rest of the FAISS framework. We made specific modification on the Makefile variables so as to direct the SDAccel compiler to successfully compile the whole framework by loading the OpenCL runtime libraries.

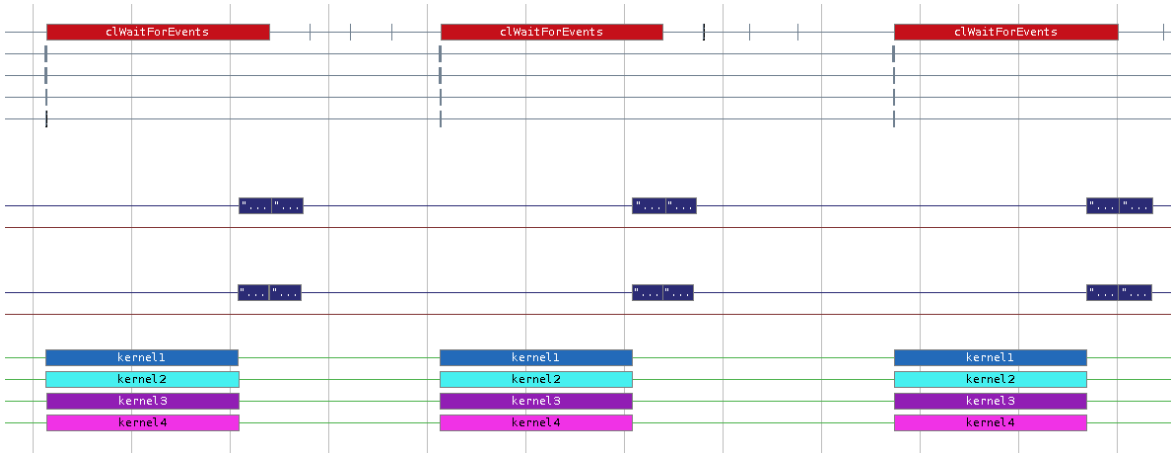

Fig. 2. Hardware Dataflow

The FPGA API hides beneath the FAISS framework and can be activated by simply changing the build target option from 'SW' to 'HW'. The in-memory FPGA model of FAISS maps the appropriate FAISS functions to FPGA memory and gives access of the device specifications across all framework source files so that any function can instantly run in HW without delay. Additionally, all cluster points resided in FPGA global memory the whole time and hardware accelerators in FAISS 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. An example hardware dataflow can be observed on Figure 2 where concurrency from OpenCL Command Queues and synchronization between host and kernels were all precisely constructed to run concurently according to FPGA design principles for high performance.

V. Experiments

First, the proposed accelerator was verified for correctness and then we measured its performance. Also, after the final 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 OpenCL-FPGAs, particularly an Xilinx Alveo U200 datacenter card with four DDR4 channels along with a host system of Xeon CPU. For a fair end-to-end comparison with a high performance CPU we chose a c4.8xlarge instance from AWS Cloud which comes with 36-vCPU Xeon and 60 GiB of RAM, and has the same price tag (per hour) as an f1.2xlarge instance which consists of a similar FPGA, the VU9P. 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	1036	156137	89206
Percentage (%)	11	15	6	7

B. Accelerator Performance

For the evaluation of the design we first confirmed the correctness of the accelerator and evaluated it on Alveo FPGA. In order to test properly the hardware function, we simulated the FAISS variables for different dataset scenarios, generating random cluster data for various list sizes. On Figure 3 we observe the FPGA accelerator-only speed-up which reaches up to $\sim 115\times$ for large list sizes.

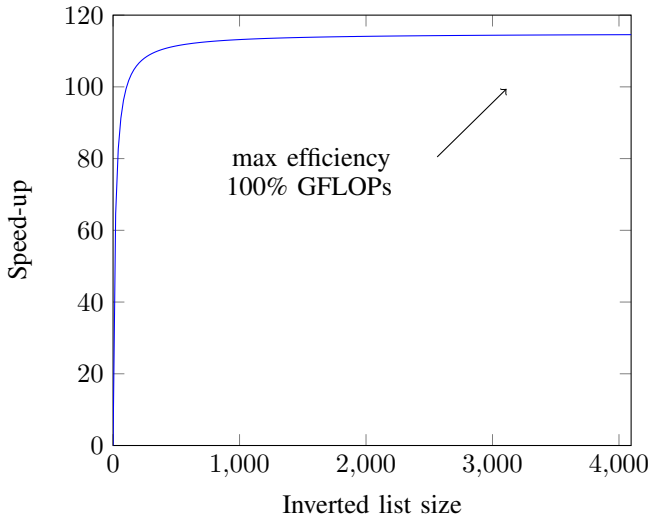

Fig. 3. Hardware accelerator performance vs single CPU

The arrow shows the maximum efficiency achieved in percentage of GFLOPs. This comes from the fact that for bigger lists the impact of data transfer of cluster data is less evident. It's worth mentioning here that the lower efficiency value which happens to be from ~ 500 list size 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 list sizes 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 $\sim 2.4\times$ 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).

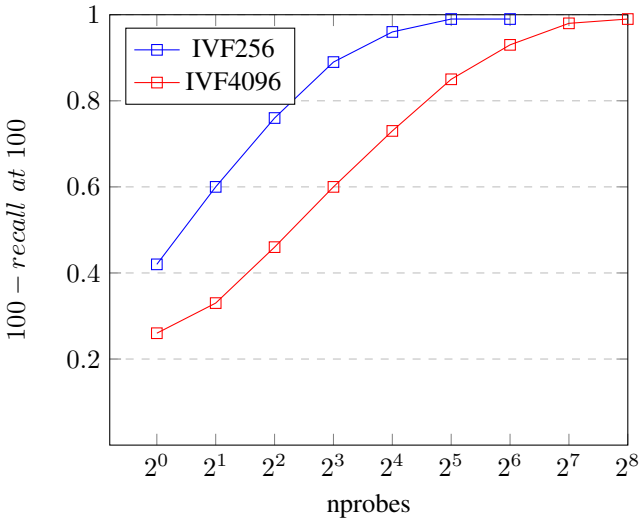

Fig. 4. Inverted Index accuracy for different probe values on SIFT 1M

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 real-world measurement on the FPGA board. 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 CPU and GPU devices in order to make a comparison of the maximum power efficiency they can ever achieve compared with our FPGA design.

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

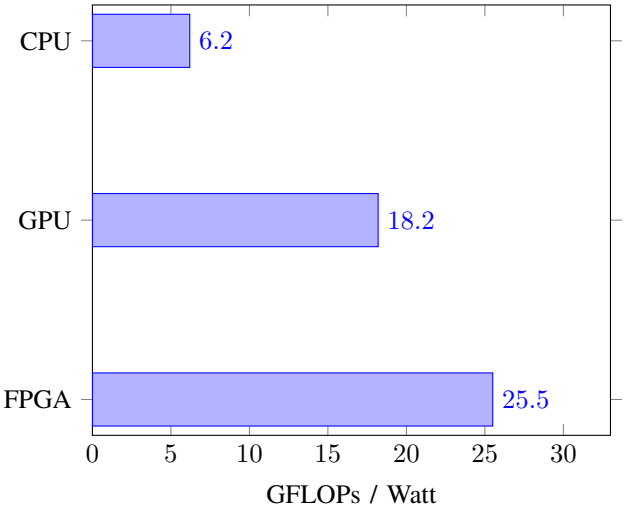

Fig. 5. Power Efficiency comparison

VI. Conclusion

Hardware accelerators looks as a promising solution for the increase of the performance and power efficiency in machine learning applications. The main challenge though is the programming efficiency where many frameworks such as FAISS do not support the transparent utilization of FPGA acceleration modules. The quantitative and qualitative comparison in this work showed that our proposed 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, especially for the cloud computing workloads, in this case for the persistent indexing creation and addition times of approximated KNN algorithms.

The superior performance and performance/watt of our design can leverage the use of FPGA hardware on frameworks or libraries on the cloud, in our case on FAISS framework especially with our easily-distributed API to multiple FPGAs. Power efficiency has gained significant traction over the last years, especially from data centers with the constantly increasing workload demands every day. Thus, heterogeneous cloud computing services coexist and many times prefer FPGAs as the platform that combines high performance and low power for a wide variety of fields.

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] 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.

[21] 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.