An Efficient Selection-Based kNN Architecture for Smart Embedded Hardware Accelerators

Hamoud Younes, Ali Ibrahim, Mostafa Rizk, Maurzio Valle, *Member, IEEE*

K-Nearest Neighbor (kNN) is an efficient algorithm used in many applications e.g. text categorization, data mining, and predictive analysis. Despite having a high computational complexity, kNN is a candidate for hardware acceleration since it is a parallelizable algorithm. This paper presents an efficient novel architecture and implementation for a kNN hardware accelerator targeting modern System-on-Chips (SoCs). The architecture adopts a selection-based sorter dedicated for kNN that outperforms traditional sorters in terms of hardware resources, time latency, and energy efficiency. The kNN architecture has been designed using High-Level Synthesis (HLS) and implemented on the Xilinx Zynqberry platform. Compared to similar state-of-the-art implementations, the proposed kNN provides speedups between 1.4× and 875× with 41% to 94% reductions in energy consumption. To further enhance the proposed architecture, algorithmic-level Approximate Computing Techniques (ACTs) have been applied. The proposed approximate kNN implementation accelerates the classification process by 2.3× with an average reduced area size of 56% for a real-time tactile data processing case study. The approximate kNN consumes 69% less energy with an accuracy loss of less than 3% when compared to the proposed Exact kNN.

***Index Terms*—Embedded Implementation, Hardware Accelerators, K-Nearest Neighbor, Approximate Computing, Tactile Sensing, Real-time Processing, Energy Efficiency, High Level Synthesis, FPGA**

I. INTRODUCTION

MODERN System-on-Chips (SoCs) are designed with heterogeneous architectures to support a variety of computationally intensive tasks in many application domains such as IoT systems, industrial automation, robotics, etc. These systems could consist of multi-core processors, or specialized hardware such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). The latter is used for accelerating complex operations and performing tasks concurrently compared to traditional processors.

K-Nearest Neighbor is a supervised classification algorithm used in a variety of applications such as pattern recognition, computer vision and machine learning [1]. However, kNN imposes significant computational workload since it has a linear scalability with the size of the dataset and the number of classes [2]. For embedded implementations, such workload demands significant memory requirements with high latency and power consumption [3], which makes kNN hardware acceleration a necessity. kNN algorithm involves independent operations e.g. the distance computation between a point A and point B is independent of that between points A and C, and kNN doesn't require the sorting of the entire distance vector to find the K-Nearest Neighbors. Such characteristics could be exploited to reduce the computational complexity of the algorithm using a pipelined architecture and tweaking the sorting process. Another solution for complexity reduction could be the use of Approximate Computing Techniques (ACTs). Approximate computing is the idea of reducing the accuracy to an acceptable limit to save energy, memory, and execution time without affecting the applications' overall quality [4]. Compared to Exact Computation Techniques (ECTs) i.e. performing arithmetic operations following the IEEE-754 and IEEE-854 standards for floating-point operations [5], the challenge of using ACTs is to maintain the approximation error acceptable with respect to the target quality of service (QoS) [6]. ACTs can be divided into two categories as shown in Figure 1: Processing-oriented and Data-oriented [6]. Processing-oriented techniques consist of "Computation Skipping" and "Computation Approximation". The former technique involves removing some processing tasks (e.g. image compression, encoding, etc.). The latter one replaces computationally intensive blocks with approximate ones that implements the same mathematical operations. Data-oriented techniques target dataset modifications as "Dataset Reduction" and "Data Format Modification". Dataset Reduction aims at decreasing the number of processed samples through Downsampling and Downscaling. Downsampling includes adjusting the sampling frequency of a processing block or truncating the size of an acquired sample. Downscaling reduces the dimension of the

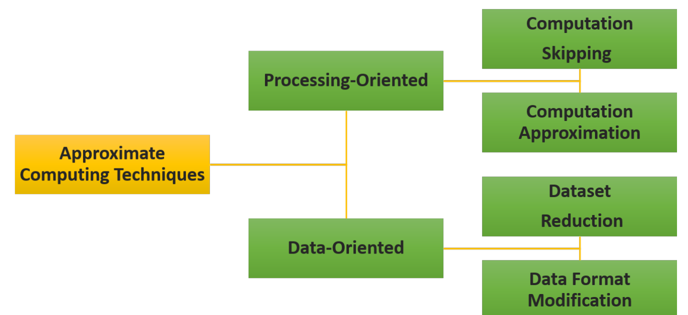

Fig. 1. Approximate Computing Techniques (ACTs)

Hamoud Younes, Ali Ibrahim, and Maurizio Valle are with the Department of Electrical, Electronic and Telecommunications Engineering and Naval Architecture, University of Genova, 16145 Genova, Italy E-mails: hamoud.younes, ali.ibrahim, maurizio.valle@unige.it

Hamoud Younes, Ali Ibrahim, and Mostafa Rizk are with the Department of Computer and Communication Engineering, Lebanese International University, Lebanon E-mails: hamoud.younes, ali.ibrahim, mostafa.rizk@liu.edu.lb

data by adjusting the matrix or vector size for vector-based applications. As for Data Format Modification, it changes the data from floating-point to fixed-point representation to simplify the involved arithmetic operations."
Approximate computing can lead to significant improvements from algorithmic to circuit level. Authors in [7] have presented an assessment of the different approximate computing techniques applied at the circuit, architecture, and algorithmic levels. Applying ACTs on supervised learning algorithms at the algorithmic level has been the focus of Younes *et. al* in [8] with a case study on the kNN algorithm. Authors reported that approximate kNN offers a computation boost in terms of $2\times$ speedup compared to Exact kNN.

To design a hardware accelerator for the kNN algorithm, an efficient architecture is proposed in this paper. The architecture introduces novel sorting process dedicated for kNN accompanied with several design optimizations. Moreover, ACTs are incorporated to further enhance the performance of the embedded implementation. For the rest of the paper, the term *"performance"* is used to report the characteristics of a kNN hardware implementation in terms of area, time latency, power consumption, and energy per classification. While, the term *"quality"* reflects the highest classification accuracy that a kNN architecture could achieve.

The main contributions of this paper could be summarized as follows:

- It proposes the design and implementation of a selection-based sorter (Selector) for the K-Nearest Neighbor (kNN) algorithm. The proposed selector overcomes similar state of the art solutions by reducing the occupied hardware area by up to 48% while providing a speedup up to $4.5\times$.
- It presents a novel kNN architecture that outperforms similar state of the art solutions in terms of occupied hardware area, time latency, and energy consumption. When validated on a touch modality classification, the proposed kNN achieves a speedup between $1.4\times$ and $875\times$ with 41% to 94% less energy consumption and 12% to 94% average hardware area reduction.
- It applies algorithmic level ACTs on the proposed architecture to improve its *performance*: the results show a 56.4% average area reduction, a speedup by $2.3\times$, and an energy reduction of about 69%. For a touch modality classification problem, an accuracy degradation of 2.6% was reported using the proposed approximate architecture.
- It demonstrates the feasibility of the implemented system for real-time touch modality classification when validated on Xilinx Zyqnberry platform. The proposed exact and approximate kNNs consume 6 μJ and 1.9 μJ respectively.

The rest of the paper is organized as follows: Section II reports on some of the efficient implementations of kNN presented in the literature. Section III explains the design and the high-level synthesis of the proposed selection-based kNN architecture. Also, it describes the selector integration into exact and approximate kNN classifiers. Section IV presents an overview of the tactile data processing case study and the experimental setup used to asses the *quality* of the proposed architecture. This Section also highlights the implementation tools and methodology with emphasis on the adopted design optimization techniques. Section V provides a *performance* analysis for the FPGA implementation of both the exact and approximate kNN classifiers. Also, a comparison with similar works from the literature is conducted. Section VI concludes the paper highlighting some future works.

II. State-of-the-art

Several architectures have been proposed in the literature for implementing the kNN algorithm on hardware platforms. One of the first architectures and implementations is presented in [9]. The architecture is based on a neural network with SIMD-style architecture, which imposed excessive response time while performing complex operations. Authors in [10] designed flexible IP cores based on linear array architecture for the FPGA implementation of the kNN algorithm. The cores achieved a very high throughput when validated on a medium size FPGA device, very large size classification problems, and with thousands of reference data vectors. A novel dynamic partial reconfiguration (DPR) architecture of the kNN algorithm is presented in [11]. The architecture is characterized by an efficient reconfiguration time for different values of K. Speedups between $68\times$ and $76\times$ were recorded when compared to General Purpose Processor (GPP) implementation. Pu et. al designed a kNN-specific bubble sort algorithm to take advantage of the FPGA parallel pipeline structure using OpenCL [12]. The overall implementation showed an enhanced *performance* compared to conventional GPU implementations. Authors in [13] adopted a Bitonic sorting algorithm to implement the kNN algorithm on both FPGA and GPU. Using OpenCL coding style and some HLS directives, the results showed that an FPGA implementation could be comparable to a GPU in terms of execution time. Another HLS based implementation has been reported in [2]. With a reduced number of comparators in the sorting process and by utilizing the memory-mapped AXI4-Master Interface of the Xilinx ZC706 FPGA board, a $35.1\times$ speedup over a GPP based implementation is noticed. In [14], several architectures are presented, where each one adopt a single or multiple HLS optimization directives. Speedups between $3.8\times$ and $58\times$ could be achieved while using the quick sort algorithm and the BCW_9 dataset. Both hardware and software designs using Xilinx MicroBlaze platform were used to verify the bubble sort based architecture of the kNN algorithm in [15]. The hardware design achieved $127\times$ speedup compared to its software counter counter-part. To reduce the impact of the memory-access constraint in kNN classification problems, an HLS based kernel is proposed in [16]. The kernel employs two data access reduction methods: low precision data representation and principal component analysis based filtering (PCAF). The kernel performed to an equivalent 56-thread CPU server while greatly reducing external memory-accesses.

A common characteristic of the most existing efficient kNN architectures is the use of the conventional sorting algorithms without optimizations for the kNN algorithm. The efficiency of these algorithms is affected by the: i) need to sort all the

vector's elements, ii) large number of required comparators, and iii) increased complexity/latency for large vector size. In this work, we propose a kNN architecture that avoids such sorting algorithms and adopts a selection-based one with a re-configurable division ratio for complexity and latency trade-offs (see Section III.C). Moreover, existing kNN implementations focus on acceleration gains compared to CPU-based implementations. This work reports a detailed comparison with FPGA-based kNN implementations that sheds the light on different implementation strategies and their effect on the kNN *performance*.

III. K-NN PROPOSED HARDWARE ARCHITECTURE

A. *k-Nearest Neighbor Algorithm Overview*

The kNN algorithm classifies an input sample according to the class of the majority of K-nearest samples. For an input sample, the kNN classifier 1) calculates the distance between the input sample and all the samples in the training set $T_i \in T$, 2) sorts the distances in ascending order, and 3) selects an output class based on the minimum distance towards K neighbors. The three main considerations for kNN are:

- The number of nearest neighbors K. A 1-NN classifier is naïve, while a large value of K might result in over-fitting. Thus, the value of K is determined using a trade-off between accuracy and complexity.
- The distance metric could be Chebyshev, Manhattan, Cosine, Euclidean, etc [17].
- The sorting algorithm could be bubble, select, quick, etc [18].

B. *Selection-based kNN Architecture*

The block diagram of the proposed hardware architecture is shown in Figure 2. The "kNN Classifier" block has been designed in HLS (coded in C++), whereas the other blocks are existing IP blocks embedded in Vivado. The SDRAM memory is used to store the training set, which is more suitable than the Block RAM (BRAM) of the FPGA for platforms with limited number of BRAMs or applications with large datasets. The AXI Interconnect IP handles the read and write operations from and to the memory. It adopts an AXI smart connect IP to use one AXI port for 1) writing to the data acquisition block and 2) reading the classification result from the class determination block. The Zynq processing system IP is the main block of the design which uses a processor system reset IP to drive all blocks with a common clock and reset signals. The kNN HLS IP starts operating once the Data acquisition block receives the training samples. To reduce the access overhead imposed by DRAM, we fetch the samples in bursts to reduce the number of memory accesses. Such technique has shown its efficiency in [13] and [19]. First, the distance between a testing sample and all training samples is performed. Then, the K minimum distance valus are determined using the proposed selector (see subsection C), and finally a class is assigned for the testing sample using the class majority of the samples corresponding to the K minimum distance values.

C. *Nearest Neighbors Selector*

The proposed architecture aims to replace the conventional sorter block with a "Selector" which finds the K-minimum distances without sorting the entire vector [20] as depicted in Algorithm 1. While coding the selector in HLS, the minimum K-distance values are saved in the same vector to be sorted, thus decreasing the memory footprint.

The selector operations can be detailed in three steps:

- Step 1: The distance vector V of size S is divided into two vectors $V1$ and $V2$. The suitable division ratio ($a\%$:$b\%$) is determined via a software simulation. Start by decreasing the size of V until the classification accuracy drops to obtain the value of a. Hence, $b = S - a$.
- Step 2: K-registers are initialized with a maximum value (e.g. 1000). Each distance value $V1[i]$ is compared to the content of register 1. If it is smaller, $V1[i]$ occupies the register, and the old content in register 1 is shifted to register 2. Then, the content in register 2 is shifted to occupy register 3. Consequently, the content in register i is shifted to occupy register $i+1$. Else, $V1[i]$ is compared

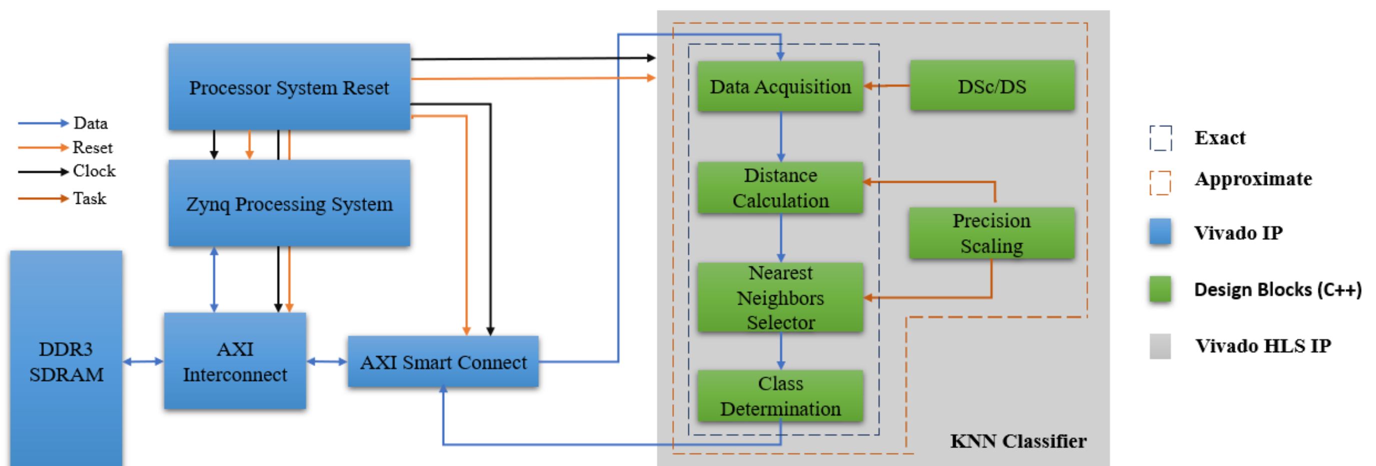

Fig. 2. The Proposed Hardware Architecture

Algorithm 1: Nearest Neighbors Selector

Input: Vector V with size S, Division ratio a:b, Number of neighbors K
Output: V with the K-minimum elements at the first K indices

$S1 \leftarrow a \times S/100$
for $i \leftarrow$*K to* $S1$ **do**
  if $V[i] <= V[0]$ **then**
    $V[K-1] \leftarrow V[K-2]$
    $\vdots$
    $V[0] \leftarrow V[i]$
  else if $V[i] <= V[1]$ **then**
    $V[K-1] \leftarrow V[K-2]$
    $\vdots$
    $V[1] \leftarrow V[i]$
  $\vdots$
  else if $V[i] <= V[K-1]$ **then**
    $V[K-1] \leftarrow V[i]$
for $j \leftarrow$*S1 to* S **do**
  if $V[j] >= V[K-1]$ **then**
    break
  else
    if $V[j] < V[K-2]$ **then**
      $\vdots$
      if $V[j] < V[0]$ **then**
        $V[0] \leftarrow V[j]$
      else
        $V[1] \leftarrow V[j]$
    else
      break

to the next register, and so on. At the end of step 2, the K minimum distance values are saved in the K registers in the order $min1 < min2 < ..... < minK$.

- Step 3: Each distance value $V2[i]$ is compared to the highest minimum i.e. $minK$ as shown in Figure. 3 (K=3). If it is larger, the minimums obtained from $V1[i]$ are not updated and a new value of $V2$ is fetched. Else, $V2[i]$ is compared to the other minimums to reach a register to occupy. Once $V2[i]$ occupies a register, the old value of that register is shifted to occupy the register of the next minimum.

The advantage of this architecture compared to the one presented in [14], is that if $V2[i]$ is greater than $minK$, step 3 will not be executed. Thus, the K minimum distances are the output of step 2. This will result in a reduced selection time for hardware implementations. Moreover, the architecture in [14] selects the K minimum distances in a single step, which imposes hardware complexity and increased time latency for large datasets. While in the proposed architecture, the selection is performed in two smaller steps with a high probability that the third step will not be executed ($V2[i] > minK$).
Although the proposed selector finds the K-nearest neighbors without sorting the entire vector, a comparison with two sorters reported in the literature, i.e. QuickSort [12] and Bitonic Sorter

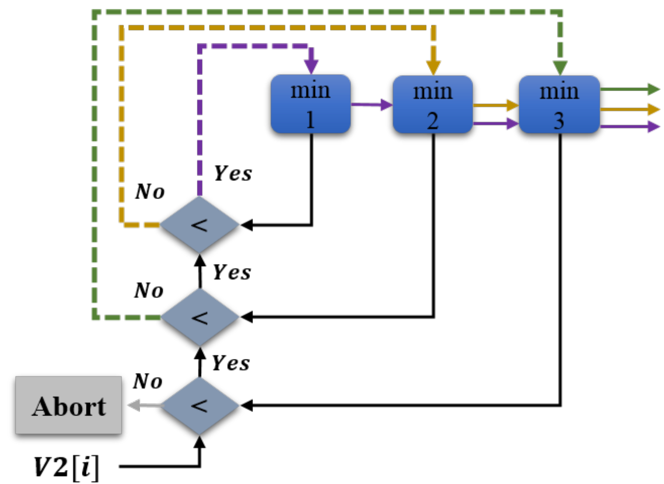

Fig. 3. Sorting Process Step 3 (K=3): Dashed Line (new value), Solid Line (old value), Colored Lines (concurrent operations)

[21], has been carried out. In general:

- Bitonic sorting is a recursive algorithm that sorts a Bitonic sequence in a parallel operating fashion. A Bitonic sequence is a sequence of M elements in which L elements out of M are sorted in ascending form, and the other $M - L$ elements are sorted in descending order [22]. If the sequence is not Bitonic, an additional task is imposed before the ability to sort the vector. The proposed selector can operate on any vector form.
- QuickSort selects one of the elements in the sequence to be the pivot and divides the sequence into two sequences one with all elements less than the pivot while the other contains all the elements greater than the pivot. The process is recursively (add more burden on the hardware) applied to each of the sub sequences. QuickSort has a worst-case complexity of $O(n^2)$ when the given sequence is sorted; this resembles the best-case scenario for the proposed selector as it will select the K minimums faster (the minimums occupy the first K registers). Then all the comparisons fail thus no shift operations are performed. Consequently, step 3 is not executed at all.
- The number of comparators required by the selector depends on the number of neighbors K, while it depends on the size of the vector N in the case of Bitonic and QuickSort. In machine learning applications, usually, it is valid that K (the number of Nearest Neighbors in kNN) $\ll$ N (size of training vector). Given that the selector doesn't sort the complete vector, the number of comparisons is decreased.

Both the sorters presented in [12],[21], and the selector

TABLE I
HLS SYNTHESIS RESULTS OF DIFFERENT SORTERS

Sorter	FF	LUT	Clock Cycles
Proposed Selector	84	176	15 (division ratio 6:4)
[12]	106	226	17
[21]	123	514	96

have been coded in C++ using Vivado HLS. The distance vector V has been used as a testing vector for the three implementations. In this paper, the best K value and division ratio were determined to be 3 and 6:4 (for the case study presented in section IV.B) i.e. there is a need to sort only 60% of the vector and step 3 can be aborted without affecting the selection process accuracy. The obtained synthesis results for finding the three minimum numbers in V are presented in Table I. Results show that the selector occupies less hardware area than the implementations of both sorters. Specifically, an average reduction of 21.4% and 48.7% is reported compared to the sorters in [12] and [21] respectively. Concerning the time latency of the sorting process, the selector is faster than the sorter in [21] by $4.5\times$. Compared to the sorter in [12] that adopts one of the fastest sorting algorithm (QuickSort), the *performance* depends on the selection of the division ratio in step 1 and the number of neighbors *K*. In fact, if all the minimum numbers are located in *V2*, or if the required value of *K* is very large; the selector is now sorting all the elements of *V* resulting in a slower sorting process. Hence a speedup of $\pm 1.2\times$ (for 6:4 and 5:5 ratios respectively) has been observed for the given task.

D. Approximate kNN Blocks

In [23], we have presented a complete assessment of using algorithmic level ACTs on a kNN classifier. The assessment included the degradation in accuracy to the gain in memory and execution time on Intel i7 CPU. In [8], the studied techniques have been formulated into a general approach that has been tested for the FPGA implementation of two machine learning classifiers. The reported approach has been adopted in this paper where a trade-off between the classifiers *performance* and *quality* has been considered. The trade-off resulted in our selection for the ACTs presented in the proposed approximate kNN architecture. All the adopted techniques belong to the data-oriented approximate computing category [8]. The adopted ACTs are Dataset Reduction (Downsampling (DS) and Downscaling (DSc)), and Data Format Modification (DFM). DS means varying the signal sampling frequency during signal acquisition. Since tactile data used in this work are from an already available dataset, the sampling frequency can't be changed. As a consequence, DS is applied offline on the dataset by reducing the number of samples. DSc is applied by adjusting the sample size as shown in subsection IV.A. DFM reduces the sample resolution. This can be achieved through the use of fixed-point or mixed precision instead of floating-point data representation. In this paper, fixed-point representation is adopted, and the precision is determined as a trade-off between resolution (32, 24, 16, and 8-bit) and classification accuracy." The approximate kNN classifier starts operating once the Data Acquisition block receives the training samples (after DS/DSc has been applied offline) from the memory. Then, the same steps performed by the kNN HLS IP are executed.

IV. Selection-based kNN Implementation and Case Study

A. Case Study: Tactile Data Processing for Electronic Skin Systems

Electronic skin system is an artificial system developed to mimic human skin behaviour or to implement intelligent tasks in applications such as robotics, prosthetic, etc. Intelligence involves the use of learning algorithms for tactile data processing in tasks such as surface texture, object compliance, touch modality, etc. [24], [25]. Touch modality classification allows the integration of gesture-based actions to be performed by robots or humans with prosthetic hands. For a medical purpose of caring for patients with mild mental impairment; a humanoid equipped with artificial skin has been trained to discriminate between nine touch modalities (scratch, tickle, rub, etc.) [26]. Recognition rates up to 96.7% has been achieved using four machine learning algorithms including kNN, SVM, DT, and Logitboost. Authors in [27] and [23] have adopted a touch modality classification problem that involves three patterns: brushing a paint brush, rolling a washer, and sliding a finger on 4×4 tactile sensory array. SVM and Extreme Learning Machine (ELM), kNN, and Deep CNN based on transfer learning have been chosen as learning algorithms. A classification accuracy of 90%, 89.8%, and 76.9% has been recorded respectively.

In this paper, the kNN algorithm is adopted for the design of an embedded tactile data processing architecture due to the: 1) high level of parallelization of the kNN algorithm, which makes it adequate for hardware acceleration, 2) high classification accuracy with a reduced computational complexity compared to state-of-the-art algorithms operating on the same task [28], [29], and 3) ability of complexity reduction without affecting the application *quality* using approximate computing techniques [23].

The dataset collected in [30] has been selected for the validation of the proposed kNN architecture. The experimental setup is shown in Figure 4 and can be described as follows:

- Dataset: The dataset contains records for the two touch modalities performed by 70 participants. Each modality was recorded from a 4×4 tactile sensor for 10 seconds at 3 kHz sampling frequency. Thus, each raw data sample can be modeled in the form of a tensor of size $4\times4\times30{,}000$. The touch modalities were performed on both the horizontal and vertical directions for two trials, resulting in a dataset of 840 samples.
- Simulation Software: An open-source machine learning simulation tool called "Weka" has been used [31]. Weka involves a collection of learning algorithms that can be applied to a pre-defined dataset or invoked from a Java code. The tool has options for classification, clustering, regression, etc.
- Classification Task: the binary problem "Sliding a finger" vs "Washer Rolling". For Weka simulation an Attribute-Relation File Format (ARFF) file is required. Thus, a header describing the features and the possible output class of each sample is added to the original tactile dataset.

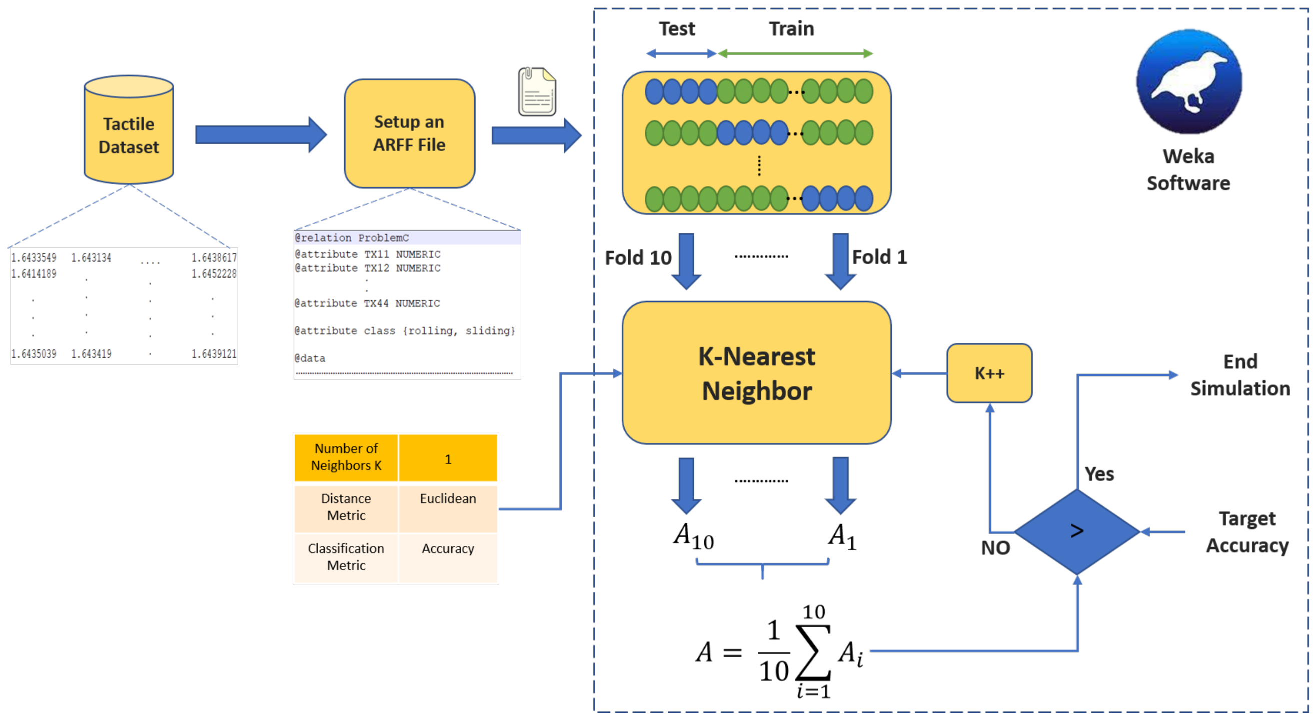

Fig. 4. Experimental Setup

- kNN Characteristics: A 10-fold cross-validation simulation using the Weka tool has been carried to determine the best value of K. The adopted distance between two tactile samples $T1$ and $T2$ is the squared euclidean distance written as:

$$d(T_1, T_2) = \sum_{0}^{16} (TX_{ij} - TX_{mn})^2 \tag{1}$$

  where TX_{ij} and TX_{mn} are the taxels inside a 4×4 tactile sample.
- Classification Metric: The kNN algorithm has been assessed by calculating the classification accuracy i.e. the ratio of correctly classified samples to the total number of available samples.

The ARFF file was loaded into Weka and classification using kNN is performed. A kNN classifier with 3-nearest neighbors resulted in the highest classification accuracy of 89.8%. This result was achieved based on a model selection approach. Based on the best kNN model obtained, a novel efficient kNN architecture is proposed in this paper. Furthermore, approximate computing techniques have been applied to enhance the overall *performance* of the proposed architecture without affecting the *quality* of the application.

The best obtained kNN model with K=3 is referred to as Exact kNN. As for Approximate kNN, it employs the following techniques to the Exact architecture: 1) Downsampling which applies an approximate window on the touch modality, where only the data that corresponds to the interval [a,b] seconds is considered. First, the interval [a, b] is selected such that $0.1 < a \ll 1$ and $9.5 < b < 10$. Then, the values of a and b are varied, while calculating the classification accuracy. The interval [3.5, 7] provided the highest accuracy among others. Thus, each modality tensor can be written as $\phi = 4 \times 4 \times 10,500$ (touch readings that belong to the interval [3.5, 7] seconds), and 2) downscaling which reduces the tensor size to $\phi = 4 \times 4 \times 1$, where the last dimension is the mean of the 10,500 readings according to the equation:

$$mean = \frac{\sum TX_{ij}}{10500} \tag{2}$$

where TX_{ij} is the individual taxel inside the 4×4 tactile sample. Figure 5 shows the initial and obtained touch modalities after applying DS and DSc.

It is worth mentioning that the tensor representation of data has been adopted by [30] since it preserves the initial structure of the data, which is still valid after applying approximate computing techniques. This is evident in Figure 5 (b), (c) where the two touch modalities can still be differentiated. For both the Distance Calculation and Nearest Neighbor Selection blocks, the operations are implemented in 24-bit fixed-point representation with a $< 6, 18 >$ precision. The adopted precision is based on a trade-off between complexity and classification accuracy.

B. Hardware and Software Design Tools

The Zynqberry TE0726-03M [32] was adopted for implementation. Zynqberry is a small-sized platform in the form of a Raspberry Pi compatible System-on-Chip (SoC) module integrating a Xilinx Zynq-7010 with a 512 MB SDRAM memory. The Zynq SoC has a hybrid structure of combining a dual-core ARM Cortex-A9 processor as a Processing System (PS) and an FPGA as Programmable Logic (PL) in a single SoC. Moreover, for comparison purposes, the architecture has

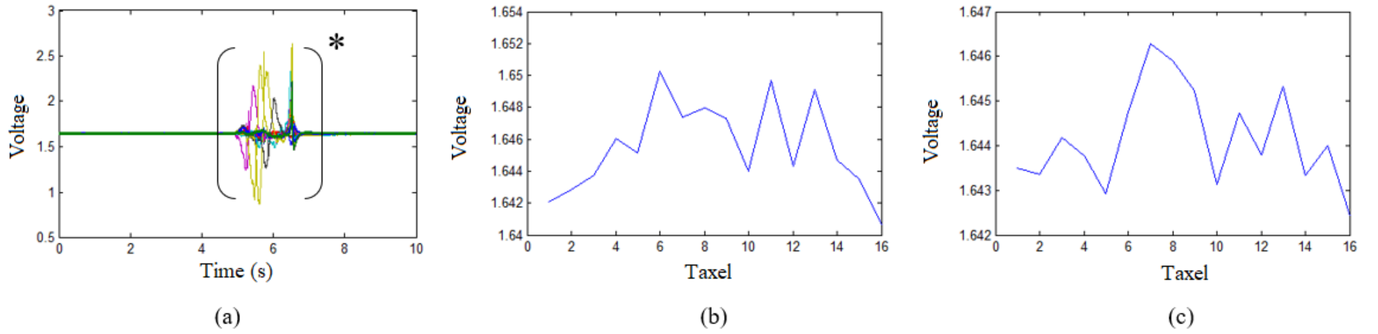

Fig. 5. Touch Modalities: (a) Rolling with DS, (b) Rolling after DSc, (c) Sliding after DSc, * Window

been also implemented on the Virtex-7 FPGA and the NVIDIA GTX 1650 GPU.

As for the software tools, Vivado HLS 2018.3 and Vivado 2018.3 were used. Vivado HLS allows the design of an embedded system on FPGA using a high-level programming language such as C and C++ compared to traditional hardware description languages (HDL). The use of HLS decreases the FPGA development time and effort. Also, it offers a set of optimization directives that can be used to enhance the design *performance*. Once the design is completed, it can be exported as a Register Transfer Level (RTL) Intellectual Property (IP) block. The latter is imported into Vivado and can be connected to the processing system (PS) via built-in IPs; thus, it is possible to obtain the hardware resources, time latency, and power consumption of the whole design.

C. Implementation Methodology

Once the whole design code is completed in HLS and design optimizations are applied, a co-simulation is performed. This simulation runs both the C++ and the RTL simulations together to verify a matching output. Then, the design is exported as an IP block. The latter is imported into Vivado 2018 and connected to the Zynq processor and other IP blocks as seen in Figure 2. First, a behavioral simulation is performed to verify the functionality of the design. Then, synthesis and place and route occur to finalize implementation. At this point, the generated report contains the occupied area percentage (BRAM, DSPs, etc.) and the number of clock cycles passed to generate an output. Concerning power consumption estimation, Vivado offers two methods: Vector-based and Vector less. This estimation can be performed at any stage between post-synthesis to post routing. For a credible estimation, a post-implementation functional and timing simulation is used to generate a Switching Activity Interchange File (SAIF) to be used for a vector-based estimation post-routing.

D. Design Optimization

Targeting the real-time functionality on a small-sized platform such as Zynqberry, the proposed architecture has been optimized to ensure an acceptable balance between time latency and hardware requirements. This has been achieved with several design optimizations as shown in Fig. 6. Such optimizations are facilitated with the use of Vivado HLS directives [33] as depicted in Algorithm 2. These optimizations are summarized as follows:

Algorithm 2: kNN Design Optimization

```
#pragma HLS ARRAYMAP
variable=Distance Instance=AllPatterns horizontal
variable=Modality Instance=AllPatterns horizontal
/* M: nb of features */
function UDC(T_1, T_2):
for i ←0 to M do
    #pragma HLS UNROLL factor=4
    Execute (1)
/* Tactiles: trainingset, q: testing
   point */
/* N: nb of training points */
for i ←0 to N do
    #pragma HLS INLINE
    #pragma HLS UNROLL factor=6
    Distance[i]= UDC(p, Tactiles[i])
    Modality[i]=Tactiles[i][16]
/* K: nb of Neighbors */
NearestNeighbors=Selector(K, Modality)
for i ←0 to K do
    #pragma HLS UNROLL
    if NearestNeighbors[i] == 1 then
        modality1count++
    else
        modalitycount2++
```

1) BRAM Resources Reduction: The BRAM size is 18K in the FPGA, if many arrays have a size less than 18K, it is better to combine them into a single array. Since kNN is a supervised algorithm, the class of each query must be known. Thus, we can benefit from this directive to combine the “Distance” and “Modality” arrays into a single array as shown in Figure 6(a). Thus, when the selector block finds the three minimum neighbors, the class of each selected neighbor is available at the same instant. This process is referred to as ”Array Map” where the horizontal option means that the two arrays are combined into a single array with more elements (see Algorithm 2).

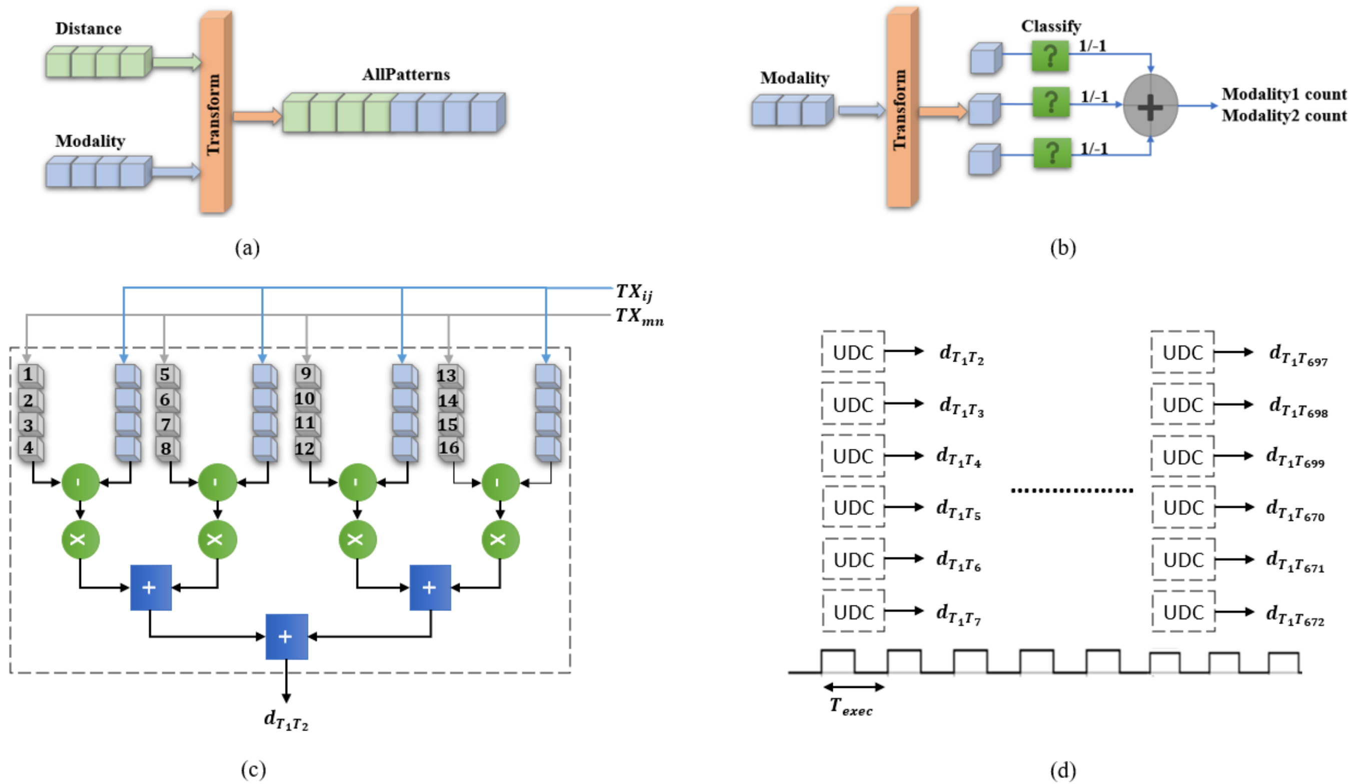

Fig. 6. Design Optimization: (a) BRAM Resources Reductions, (b) Unrolled Class Determination, (c) One Unrolled Distance Calculation (UDC) Block, (d) Complete Unrolled Distance Calculation

2) Parallelization: To exploit the capabilities of the FPGA, the operations of the distance calculation and the class determination blocks are executed in parallel with a small unroll factor. In HLS terms this is known as "Unrolling", where unrolling a loop creates multiple copies of its body in the RTL design, which allows some or all of its iterations to occur in parallel. This optimization has been applied to (1) denoted as Unrolled Distance Calculation (UDC) and the Class Determination blocks leading to accelerating the calculation and output class decision. However, executing all the operations in parallel leads to a high power consumption and increased resource requirements. To avoid the negative impact of unrolling on the hardware cost and power consumption, the loops are partially unrolled (unroll factors of 4 and 6) as it is shown in Algorithm 2, and the design is implemented at an operating frequency of 100 MHz which is lower when compared to similar work [21]. Each touch modality sample has 16 features, thus an UNROLL factor equals to 4 is used. This means that the distance between each 4 features is calculated in a single time interval as shown in Figure 6(c). Similarly, Figure 6(d) shows how the UDC block is used to calculate the distance between the testing sample and all training samples. An UNROLL factor equals to 6 is used, thus 112 timing intervals are required to finish all the distance calculations for a training set size of 80%. The distance from a testing sample to $(80 * 840/100) = 672$ training samples is calculated in batches of 6 calculations per timing interval i.e $672/6 = 112$ intervals. As for Class Determination, since $K = 3$ and we have a binary classification, the process could be fully unrolled as shown in 6(b).
3) Function Inline: The inlined function is treated as a part of the calling function that is calling it rather than a separate entity. This optimization is applied for the distance calculation function. Thus, whenever the classification function is called, the distance calculation is executed within it and it no longer appears as a separate level of hierarchy in the RTL design. Thus improving the overall latency of the classification task.

V. Implementation Results and Assessment

The *performance* and *quality* of the proposed exact and approximate implementations are assessed on the touch modality classification problem mentioned in section IV. The assessment involves three case studies: (1) Proposed Exact kNN versus approximate kNN, (2) Exact kNN on FPGA versus GPU, and (3) Exact kNN versus similar works. For cases (1) and (3), the time latency T is calculated according to the equation:

$$T = N \times 1/f_{max} \tag{3}$$

where N is the number of clock cycles obtained in post-implementation reports and f_{max} is the maximum operating

frequency the design can achieve. The Joule per classification energy E is calculated as:

$$E = T \times P \tag{4}$$

where T is the time latency and P is the dynamic power consumed by the programmable logic (PL) of the Zynqberry reported by Vivado i.e the power consumed by the simulated kNN architecture to compute a classification of an input sample (excluding the static power of the processing system (PS) as it is device dependent).

For case (2), Exact kNN architecture has been coded using Python language running inside the CUDA computing platform. The GPU power estimation was obtained using NVIDIA System Managment Interface (NVSMI). The latter is a command utility that can be issued on any Python development environment with the CUDA libraries imported [34].

Case 1: Exact versus Approximate kNN

The implementation results on Zynqberry of the proposed Exact and Approximate kNN are shown in Table II. Exact kNN occupies 12% of the hardware resources, consumes 0.236 W, and classifies an input sample within 25.7 μs. The obtained time latency verifies the real-time classification of a touch modality in less than 400 ms [35]. Applying downsampling and downscaling have decreased the input size from $4 \times 4 \times 30,00$ (a 10s sample) to $4 \times 4 \times 1$ (a 3.5s sample) offering a 65% reduction in data size. Such reduction led to a significant decrease in the hardware resources and time latency. Using the 24-bit fixed-point representation instead of 32-bit floating-point one provides a 25% reduction in the word length of data exchanged between the different blocks of the kNN architecture and the complexity of the arithmetic computations. Consequently, the dynamic power consumption has been reduced. Thus, the proposed approximate kNN offers an average hardware resource reduction up to 56.4%, by accelerating the classification of a test sample by 2.3× with an energy reduction of about 69% compared to the proposed Exact kNN. For the whole design, an accuracy degradation of 2.6% is reported. The proposed approximate kNN provides real-time classification of touch modalities with a reduced time latency of 11.2 μs. These results motivate the use of approximate computing techniques.

Case 2: Exact kNN on FPGA versus GPU

TABLE II
IMPLEMENTATION RESULTS OF THE PROPOSED EXACT AND APPROXIMATE CLASSIFIERS ON ZYNQBERRY

Implementation	Exact	Approximate
Classification Accuracy	89.8%	87%
Frequency (MHz)	100	
BRAM	4	4
DSP48E	10	4
FF	3493	1612
LUT	2825	1264
Time Latency (μs)	25.7	11.2
Dynamic Power Consumption (W)	0.236	0.164

TABLE III
EXACT KNN PERFORMANCE ON FPGA AND GPU

Exact kNN	
Time (FPGA/GPU)	25.7μs/80ms
Energy (FPGA/GPU)	6μJ/1.13J
Time Ratio	0.00032
Energy Ratio	5.37E-6

Using the CUDA platform and NVSMI tool, the GPU implementation of Exact kNN provided a classification time of 80 ms while consuming 14.12W for the touch modality classification problem. Table III shows a comparison between the FPGA and GPU implementations in terms of execution time and energy consumption. The results are significantly in favor of the FPGA, where the acceleration of the proposed kNN architecture on FPGA could be achieved with a fraction of the energy consumed using GPUs. This can justified due to two possible reasons:

- GPUs use DRAM for the communication between the different blocks of the kNN architecture (referred to as kernels) [36], which is slower than using a hybrid structure as proposed in Figure 2 where BRAMs are used to communicate between different blocks and DRAM is used only for dataset storage.
- The proposed kNN architecture exploits the parallelism capabilities of the FPGA. Thus, the ”if-then-else” conditions are executed in parallel. On the other hand, the ”then” and ”else” parts are executed serially on GPUs resulting in a significant time latency increase. Such issue is known as ”thread divergence” [37].

Case 3: Comparison with similar works

Comparing two kNN implementations is not a straight forward task due to the large number of differences such as: the number of nearest neighbors (K), dataset size (N), number of features per sample (f), development environment (HLS or HDL), hardware device used, etc. To achieve a fair comparison with Exact kNN, three similar architectures have been selected. These three architectures have been chosen such that they all have:

- Used HLS for development since the comparison with an HDL implementation is not feasible.
- Achieved a high acceleration gain (i.e speedup) with respect to equivalent CPU-based kNN implementation, so the architecture resembles an efficient accelerator.
- Used similar (and different) values of K, N, and f to generalize the comparison.

TABLE IV
TESTBENCH IMPLEMENTATION SETTINGS FOR THE EXACT KNN AND THREE SIMILAR ARCHITECTURES

Architecture	S_1 [14]	S_2 [15]	S_3 [21]
K	10	3	5
N	699	150	300×10^3
f	9	4	2
Dataset	BCW_9	Iris	Weather
Device	AVNET ZedBoard	Virtex-7	Virtex-7
Frequency (MHz)	100	100	240

Table IV presents the existing implementation settings for the three architectures. Denote by $S_i = [K_i, N_i, f_i, Dataset_i]$ the settings used in the first [14], second [15] and third implementation [21] respectively. Exact kNN is implemented using the settings S_i i.e the proposed kNN architecture is implemented and validated using the testbench reported in each architecture. The implementation results are shown in Table V with the original implementation results of each architecture.

kNN-S_1 achieves a $12.3\times$ classification speedup with 94% less energy consumption while requiring a 61% less hardware resources compared to the kNN presented in [14]. This is due to two main reasons: 1) the selector used on kNN-S_1 is an enhanced version of the one used in [14] where the division factor plays a key role in decreasing the sorting time as presented in section IV.C, and 2) the aim of the kNN architecture in [14] is to attain the highest speedup possible for real-time embedded applications. This has been accomplished by combining the UNROLL, PIPELINE, and DATAFLOW directives. Such directives are known for speedup gains due to the level of parallelism introduced at the expense of a noticeable increase in the hardware resources. The latter was not an issue when using the relatively large FPGA in the ZedBoard platform. Meanwhile, in kNN-S_1 only the UNROLL directive is used with an unrolling factor that balances the speedup and complexity for the target application, while achieving more speedup with the use of the selector.

When compared to the kNN implementation in [15], kNN-S_2 provides a huge acceleration gain of $875\times$ with 94% less required hardware resources. Such gain is due to the design choices adopted by the authors in [15] such as: 1) using the euclidean distance metric, which compared to (1), has an added complexity due to the square root operation, 2) applying the normalization of the data on-chip, which presents a complexity overhead, and 3) performing the sorting operation using a single comparator and multiple BRAMs to compare each pair of data points, this process is very slow compared to the proposed selector. Although no power/energy details were provided for the kNN in [15], kNN-S_2 is expected to be more efficient due to the 90% reduction in the number of DSPs.

The implementation requirements of kNN-S_3 exceeds the capacity of the FPGA fabric in Zynqberry and thus the design couldn't be routed to achieve the 240MHz operating frequency. Thus, for comparison reasons only, kNN-S_3 is implemented on the target device used in [21] i.e Vertix-7 knowing that implementing kNN on Zynqberry has achieved the real-time and low power consumption demands for the target touch modality application as reported in Table II. kNN-S_3 offers a speedup of $1.4\times$ with 41.5% and 12.3% reduction in hardware resources and energy per classification respectively. Such results are justified with the lower number of FF and LUTs required by kNN-S_3. This is expected since the kNN in [21] uses the Bitonic sorter, which is outperformed by the proposed selector as shown in Table I. Compared to kNN-S_1 and kNN-S_2, the gain achieved by kNN-S_3 is relatively lower since the kNN in [21] exploits the high optimization capabilities of OpenCL for extensive computations and large datasets.

VI. Conclusion

This paper introduces an efficient novel architecture for the hardware acceleration k-Nearest Neighbor algorithm using a selection-based sorter. The architecture has been coded in HLS, synthesized, and routed on the Zynqberry platform. Two efficient implementations were provided based on exact and approximate computing. The implementations exploit the parallelism nature of the kNN algorithm along with the use of ACTs to achieve real-time classification with relatively low power consumption. Compared to similar state-of-the-art work, the proposed Exact kNN offers acceleration gain between $1.4\times$ and $875\times$ with lower energy per classification between 41% and 94% depending on the used settings. Compared to GPU-based implementations, the proposed kNN-FPGA implementation offers efficient and faster classification for the target application. Such results pave the way towards embedding intelligence using a small-sized platform such as the Zynqberry for applications with low power and real-

TABLE V
Implementation Results Comparison

Architecture	kNN-S_1	[14]	kNN-S_2	[15]	kNN-S_3	[21]
Device	Zynqberry				Virtex-7	
Frequency (MHz)	100				240	
BRAM	4	-	4	293	500	512
DSP	7	9	5	47	12	12
FF	2002	9484	827	-	21677	23892
LUT	1607	8845	1407	-	11416	11838
Time Latency (ms)	22×10^{-3}	0.27	12×10^{-3}	10.5	0.88	1.24
Energy per classification (mJ)	4.84×10^{-3}	70×10^{-3}	-	-	1.86	3.17
Average Resources Reduction (%)*	61%		94%		12.3%	
Speedup	12.3x		875x		1.4x	
Energy Reduction (%)	94%		-		41.5%	
Classification Accuracy (%)	96.2%		93.3%		86.5%	

* Calculated for the available resources only e.g. reduction in BRAM and DSP for kNN-S_2 compared to [15], i.e. Reduction = (BRAM-Reduction + DSP-Reduction)/2, where BRAM-Reduction= 100(1-4/293) and DSP-Reduction= 100(1-5/47).

time requirements. The implementation on different hardware platforms, under different settings, and operating on different dataset size, verifies the efficiency of the proposed selection-based kNN architecture. Future work will involve the investigation of the use of circuit-level approximate computing techniques that are reported to permit noticeable gains in the *performance* of machine learning hardware implementations.

ACKNOWLEDGEMENT

The authors acknowledge partial financial support from TACTIle feedback enriched virtual interaction through virtual realITY and beyond (TACTILITY) project: EU H2020, Topic ICT-25-2018-2020, RIA, Proposal ID 856718.

REFERENCES

[1] J. Sun, W. Du, and N. Shi, "A Survey of kNN Algorithm," *Inf Eng Appl Comput*, vol. 1, May 2018.

[2] Zhe-Hao Li, Ji-Fang Jin, Xue-Gong Zhou, and Zhi-Hua Feng, "K-nearest neighbor algorithm implementation on FPGA using high level synthesis," in *2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, (Hangzhou, China), pp. 600–602, IEEE, Oct. 2016.

[3] J. Saikia, S. Yin, Z. Jiang, M. Seok, and J.-s. Seo, "K-Nearest Neighbor Hardware Accelerator Using In-Memory Computing SRAM," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, (Lausanne, Switzerland), pp. 1–6, IEEE, July 2019.

[4] V. Kumar and R. Kant, "Approximate Computing for Machine Learning," in *Proceedings of 2nd International Conference on Communication, Computing and Networking* (C. R. Krishna, M. Dutta, and R. Kumar, eds.), vol. 46, pp. 607–613, Singapore: Springer Singapore, 2019.

[5] F. de Dinechin, M. D. Ercegovac, J.-M. Muller, and N. Revol, "Digital Arithmetic," in *Wiley Encyclopedia of Computer Science and Engineering* (B. W. Wah, ed.), p. ecse578, Hoboken, NJ, USA: John Wiley & Sons, Inc., Mar. 2009.

[6] E. Nogues, D. Menard, and M. Pelcat, "Algorithmic-Level Approximate Computing Applied to Energy Efficient Hevc Decoding," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2016.

[7] A. Ibrahim, M. Osta, M. Alameh, M. Saleh, H. Chible, and M. Valle, "Approximate Computing Methods for Embedded Machine Learning," in *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, (Bordeaux), pp. 845–848, IEEE, Dec. 2018.

[8] H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Algorithmic Level Approximate Computing for Machine Learning Classifiers," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, (Genoa, Italy), pp. 113–114, IEEE, Nov. 2019.

[9] S. Lucas, "A fast exact parallel implementation of the k-nearest neighbour pattern classifier," in *1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36227)*, vol. 3, (Anchorage, AK, USA), pp. 1867–1872, IEEE, 1998.

[10] E. S. Manolakos and I. Stamoulias, "Flexible IP cores for the k-NN classification problem and their FPGA implementation," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, (Atlanta, GA, USA), pp. 1–4, IEEE, Apr. 2010.

[11] H. Hussain, K. Benkrid, C. Hong, and H. Seker, "An adaptive FPGA implementation of multi-core K-nearest neighbour ensemble classifier using dynamic partial reconfiguration," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, (Oslo, Norway), pp. 627–630, IEEE, Aug. 2012.

[12] 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*, (Vancouver, BC, Canada), pp. 167–170, IEEE, May 2015.

[13] F. Muslim, A. Demian, L. Ma, L. Lavagno, and A. Qamar, "Energy-efficient FPGA Implementation of the k-Nearest Neighbors Algorithm Using OpenCL," pp. 141–145, Oct. 2016.

[14] D. Jamma, O. Ahmed, S. Areibi, and G. Grewal, "Hardware accelerators for the K-nearest neighbor algorithm using high level synthesis," in *2017 29th International Conference on Microelectronics (ICM)*, (Beirut, Lebanon), pp. 1–4, IEEE, Dec. 2017.

[15] M. A. Mohsin and D. G. Perera, "An FPGA-Based Hardware Accelerator for K-Nearest Neighbor Classification for Machine Learning on Mobile Devices," in *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, (Toronto ON Canada), pp. 1–7, ACM, June 2018.

[16] X. Song, T. Xie, and S. Fischer, "A Memory-Access-Efficient Adaptive Implementation of kNN on FPGA through HLS," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, (Abu Dhabi, United Arab Emirates), pp. 177–180, IEEE, Nov. 2019.

[17] K. Chomboon, P. Chujai, P. Teerarassammee, K. Kerdprasop, and N. Kerdprasop, "An Empirical Study of Distance Metrics for k-Nearest Neighbor Algorithm," in *The Proceedings of the 2nd International Conference on Industrial Application Engineering 2015*, pp. 280–285, The Institute of Industrial Applications Engineers, 2015.

[18] You Yang, Ping Yu, and Yan Gan, "Experimental study on the five sort algorithms," in *2011 Second International Conference on Mechanic Automation and Control Engineering*, (Inner Mongolia, China), pp. 1314–1317, IEEE, July 2011.

[19] J. Vieira, R. P. Duarte, and H. C. Neto, "kNN-STUFF: kNN STreaming Unit for Fpgas," *IEEE Access*, vol. 7, pp. 170864–170877, 2019.

[20] D. Jamma, O. Ahmed, S. Areibi, G. Grewal, and N. Molloy, "Design exploration of ASIP architectures for the K-Nearest Neighbor machine-learning algorithm," in *2016 28th International Conference on Microelectronics (ICM)*, (Giza, Egypt), pp. 57–60, IEEE, Dec. 2016.

[21] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.

[22] Q. Mu, L. Cui, and Y. Song, "The implementation and optimization of Bitonic sort algorithm based on CUDA," *arXiv:1506.01446 [cs]*, June 2015.

[23] H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Data Oriented Approximate K-Nearest Neighbor Classifier for Touch Modality Recognition," in *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, (Lausanne, Switzerland), pp. 241–244, IEEE, July 2019.

[24] N. Jamali and C. Sammut, "Majority Voting: Material Classification by Tactile Sensing Using Surface Texture," *IEEE Transactions on Robotics*, vol. 27, pp. 508–521, June 2011.

[25] J. Kwiatkowski, D. Cockburn, and V. Duchaine, "Grasp stability assessment through the fusion of proprioception and tactile signals using convolutional neural networks," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (Vancouver, BC), pp. 286–292, IEEE, Sept. 2017.

[26] M. Kaboli, A. Long, and G. Cheng, "Humanoids learn touch modalities identification via multi-modal robotic skin and robust tactile descriptors," *Advanced Robotics*, vol. 29, pp. 1411–1425, Nov. 2015.

[27] P. Gastaldo, L. Pinna, L. Seminara, M. Valle, and R. Zunino, "A Tensor-Based Pattern-Recognition Framework for the Interpretation of Touch Modality in Artificial Skin Systems," *IEEE Sensors Journal*, vol. 14, pp. 2216–2225, July 2014.

[28] M. Alameh, A. Ibrahim, M. Valle, and G. Moser, "DCNN for Tactile Sensory Data Classification based on Transfer Learning," in *2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, (Lausanne, Switzerland), pp. 237–240, IEEE, July 2019.

[29] H. Younes, A. Ibrahim, M. Rizk, and M. Valle, "Algorithmic-Level Approximate Tensorial SVM Using High-Level Synthesis on FPGA," *Electronics*, vol. 10, p. 205, Jan. 2021.

[30] P. Gastaldo, L. Pinna, L. Seminara, M. Valle, and R. Zunino, "Computational Intelligence Techniques for Tactile Sensing Systems," *Sensors*, vol. 14, pp. 10952–10976, June 2014.

[31] *Weka 3 - Data Mining with Open Source Machine Learning Software in Java*.

[32] *TE0726 Resources - Public Docs - Trenz Electronic Wiki*.

[33] X. Xilinx, *Vivado Design Suite User guide, High-Level Synthesis.*

[34] NVIDIA, *NVIDIA System Management Interface*. June 2012. Publication Title: NVIDIA Developer.

[35] P. P. Lele, D. C. Sinclair, and G. Weddell, "The reaction time to touch," *The Journal of Physiology*, vol. 123, pp. 187–203, Jan. 1954.

[36] L. Sánchez, J. Ranilla, and A. Cocaña-Fernández, "Eecluster: An energyefficient tool for managing hpc clusters," *Annals of Multicore and GPU Programming*, vol. 2, no. 1, pp. 15–24, 2015.

[37] S. Cook, "Memory Handling with CUDA," in *CUDA Programming*, pp. 107–202, Elsevier, 2013.

Hamoud Younes He received the B.S and M.S. degrees in Computer and Communication Engineering from the Lebanese International University, in 2012 and 2015 respectively. Currently, he is a Ph.D. student at the Department of Electric, Electronic, Telecommunication Engineering and Naval Architecture, University of Genoa. His research interests involve embedded electronic systems, FPGA implementation, embedded machine and deep learning, approximate computing, and energy efficient embedded computing.

Ali Ibrahim received the M.S. degree in industrial control from the Doctoral School of Sciences and Technologies, Lebanese University, in 2009, and the dual Ph.D. degrees in electronic and computer engineering and robotics and telecommunications from the University of Genova and the Lebanese University in 2016. He has been a Post-Doctoral Researcher with the Department of Electric, Electronic, Telecommunication Engineering and Naval Architecture, University of Genova from 2016 to 2018. Currently, he is an assistant professor in the department of Electrical and Electronics Engineering at the Lebanese international University in Lebanon and also an associate researcher Department of Electric, Electronic, Telecommunication Engineering and Naval Architecture, University of Genoa. His research interests involve Embedded machine learning, FPGA implementation, and interface electronics for electronic skin systems, approximate computing, and techniques and methods for energy efficient embedded computing.

Mostafa Rizk received the Maitrise degree in electronics, the M.Sc. degree in biomedical physics, and the M.Sc. degree in signal, telecom, image, and speech from Lebanese University, Beirut, Lebanon, in 2007, 2008, and 2010, respectively, the Ph.D. degree in sciences and technologies of information from Telecom Bretagne, Brest, France, in 2014, and the Ph.D. degree in electronics and communication from Lebanese University in 2015. He has been a Post-Doctoral Researcher with the University of Southern Brittany, Lorient, France, and with the Lab-STICC Laboratory CNRS, Lorient. He is currently an Assistant Professor with Lebanese International University, Beirut, and an Associate Researcher with IMT Atlantique, Brest, France. His current research interests include hardware/software implementations and digital circuit design, network-on-chip design, and new MPSoC architectures based on emerging nonvolatile memory technologies.

Maurizio Valle (MV) received the M.S. degree in Electronic Engineering in 1985 and the Ph.D. degree in Electronics and Computer Science in 1990 from the University of Genova, Italy. From December 2019, MV is full professor of Electronics at the DITEN, University of Genova where he leads the Connected Objects, Smart Materials, Integrated Circuits – COSMIC laboratory. MV has been and is in charge of many research contracts and projects funded at local, national and European levels and by Italian and foreign companies. Professor Valle is co-author of more than 200 papers on international scientific journals and conference proceedings. He is IEEE senior member and member of the IEEE CAS Society. His research interests include bio-medical circuits and systems, electronic/artificial sensitive skin, tactile sensing systems for prosthetics and robotics, neuromorphic touch sensors, electronic and microelectronic systems.