Received November 5, 2019, accepted November 21, 2019, date of publication November 25, 2019, date of current version December 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2955864

kNN-STUFF: kNN STreaming Unit for Fpgas

JOÃO VIEIRA, (Member, IEEE), RUI P. DUARTE, (Member, IEEE), AND HORÁCIO C. NETO
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, 1000-029 Lisboa, Portugal

Corresponding author: João Vieira (joaomiguelvieira@tecnico.ulisboa.pt)

This work was supported by the National Funds through Fundação para a Ciência e a Tecnologia (FCT) under Grant PTDC/EEI-HAC/31819/2017 and Grant UID/CEC/50021/2019 project ''Synthetic Aperture Radar Robust Reconfigurable Optimized Computing Architecture (SARRROCA)''.

ABSTRACT This paper presents kNN STreaming Unit For Fpgas (kNN-STUFF), a modular, scalable and efficient Hardware/Software implementation of k-Nearest Neighbors (kNN) classifier targeting System on Chip (SoC) devices. It takes advantage of custom accelerators, implemented on the reconfigurable fabric of the SoC device, to perform most of the classifier's workload, whereas the processor coordinates the accelerators and runs the remaining workload of the kNN algorithm. kNN-STUFF offers a highly flexible framework, where the designer has the possibility to define the number of parallel instances of the classifier and the parallelism within each instance. This capability allows creating the most suitable implementation for a target device of any size. Results show that kNN-STUFF, with 24 accelerators, attains performance improvements up to 67.4×, when compared to an optimized (-O3) software-only implementation of the kNN running on a single core of the ARM Cortex-A9 CPU. Furthermore, its energy efficiency improvements are as high as 50.6×.

INDEX TERMS FPGA, SoC, kNN, Parallel architectures, real-time classification, IoT.

I. INTRODUCTION

The k-Nearest Neighbors (kNN) is an important unsupervised Machine Learning (ML) algorithm for classification used in diverse fields, such as data mining, satellite and medical imaging, speech recognition, computer vision, text categorization, data compression, computational genomics, and predictive analysis [1]. It classifies multi-dimensional points in the $\mathbb{R}^M$ plan, called samples, by computing their distances to all samples in a training set, whose classes are known beforehand, and determining the most repeated class among the k training samples closest to the one being classified.

Even though the kNN algorithm is conceptually simple, when implemented in a classical fashion, the required amount of data transfers between the memory and the Central Processing Unit (CPU), as well as the instructions executed by the CPU, scale superlinearly with the size of the dataset. Thus, the use of big datasets leads to extremely long and intensive computational workloads, which require significant processing power [2]. Plus, since kNN is memory-bounded it requires massive data transfers between the CPU and the main memory, which accounts for more than 25% of the energy spent by the system [3].

The associate editor coordinating the review of this manuscript and approving it for publication was Giacomo Verticale.

With the advents of Big Data and Internet of Things (IoT), the necessity of processing in real time large amounts of data in systems that are highly constrained in terms of energy and hardware resources arose. Thus, creating mechanisms capable of efficiently running heavy workloads in low-end platforms became a need [4]–[7]. For this purpose, Systems on Chip (SoCs) containing both hard processors and reconfigurable logic have enabled the development of energy-efficient Hardware/Software (HW/SW) architectures capable of achieving high-performance. However, in the particular context of the kNN algorithm, designing a dedicated accelerator is not trivial due to the constant change of the datasets' parameters, the classifier requirements and how data is fed to the system.

Previous work on kNN accelerators targeting Field-Programmable Gate Arrays (FPGAs) and SoCs often do not scale in terms of performance and hardware and energy requirements and support only a small set of devices [8]–[12]. Furthermore, they require to re-implement the designs whenever changing the parameters of the dataset. Other contributions such as [13] show remarkable performance improvements, but require a significant amount of hardware resources to be implemented, which are hardly available in the context of IoT. Moreover, in general, these proposals focus mainly on accelerating the classification of a single sample at a time, not exploiting the parallelism enabled by classifying several samples in parallel, which is possible in many circumstances outside the laboratory environment. Another limitation of most previous solutions is their inherent data format. While most of the previously designed kNN accelerators can only operate data in fixed-point format (which usually leads to less complex data paths), in various cases the features of the samples may be only representable in floating-point format. For instance, most of bio-signals [14] are acquired in the form of complex time-series and require feature extraction before classification, which often produces real or complex floating-point values.

For all the reasons above, there is a need for creating a generic architecture that implements the kNN classifier independent of the parameters of the dataset, capable of processing data at the rate it is received and capable of operating values in floating-point format efficiently.

This paper presents kNN STreaming Unit For Fpgas (kNN-STUFF), a novel HW/SW floating-point implementation of the kNN classifier that offloads over 99% of the algorithm's workload from the processor to custom hardware accelerators. The architecture of the accelerator used by kNN-STUFF is independent of the parameters of the dataset and processes the data in a streamed manner, being able to consume new inputs every clock cycle. Furthermore, kNN-STUFF enables the implementation of multiple accelerators to perform the classification of several testing samples simultaneously. To achieve this, the training set is broadcasted to all accelerators at once, reducing the memory accesses by the number of instances running in parallel, hence, enhancing both performance and energy efficiency. Furthermore, kNN-STUFF also allows parallelizing the classification of a single testing sample using multiple accelerators. In this case, each accelerator computes the kNN relative to a training subset, allowing to transfer the training subsets to the different accelerators in parallel, thus increasing the bandwidth to the memory by the number of accelerators.

Overall, the contributions of this paper are the following:

1) We present a scalable HW/SW architecture of a system that optimizes the execution of the kNN algorithm targeting SoCs.
2) We provide an optimized, fully parameterized and synthesizable Register Transfer Level (RTL) description of an accelerator for the kNN algorithm suitable for implementation in FPGA.
3) We define a framework to generate and implement clusters of accelerators automatically.
4) We provide a software library to use the accelerators/clusters of accelerators from the software side.
5) We define an optimized software-only implementation of the kNN algorithm targeting processors to use as a baseline for performance comparison.
6) We perform extensive validation and assessment of the novel system, using real datasets and studying the effect of varying the dataset and the classifier parameters on the system's performance and hardware requirements.

The rest of this paper is organized as follows. Section II analyzes previous hardware implementations of the kNN classifier targeting FPGAs and SoCs and Section III provides background on the kNN algorithm. Section IV introduces the challenges associated with designing an accelerator, and the profiling of the algorithm used to identify its computational bottlenecks is presented in Section V. Section VI details the architecture of the custom accelerator, and Section VII explains the overall kNN-STUFF system. Section VIII presents the experimental results, and Section IX summarizes the main remarks of this work.

II. RELATED WORK

Previous hardware implementations of the kNN classifier targeting FPGAs and SoCs have been proposed (e.g., [8], [10], [12], [13], [15]–[18]). However, such proposals tackle particular applications of kNN, adopt custom fixed-point representations, and, in general, need to be re-engineered whenever the parameters of the classifier or the dataset change.

For example, Manolakos E. *et al.* [8] propose a Very Large Scale Integration (VLSI) architecture obtained through linear space-time mapping [19]. Their work devises two architectures depending on the relation between the size of the training set, N, and the number of features per sample, M. One of the architectures, obtained through the horizontal projection of the Dependence Graph (DG) of the kNN algorithm is optimized for datasets where $N \gg M$, while the other, result of the vertical projection, is suitable for datasets where $M \gg N$. They show that their architectures can achieve performance improvements between 1.5× and 3× over the NVIDIA GeForce 8800 GTX General Purpose Graphics Processing Unit (GPGPU) while making efficient usage of FPGA resources. However, their architectures are dependent on the datasets, requiring to be re-implemented whenever the training set changes. Moreover, the hardware requirements grow with the size of the datasets, since the entire training set is permanently stored in Block RAMs (BRAMs). Thus, their architectures are not scalable.

Hussain H. *et al.* [10] used an analogous approach to the previous work, but performed partial reconfiguration to reduce the overhead of re-implementing the whole design. Their system only requires to re-implement the part of the architecture that dependents on the number of nearest neighbors, k. Their base design consists of three types of blocks targeting different stages of the kNN algorithm. They show that their architectures can outperform an Intel Pentium Dual-Core E5300 up to 76×. However, the design still needs to be entirely re-implemented whenever any of the other parameters of the classifier changes.

Mohsin M. *et al.* [12] propose a heterogeneous system formed by a MicroBlaze soft-processor and a custom accelerator. Besides the original flow of the kNN algorithm, they also implement a pre-processing stage to normalize the dataset and prevent high-value attributes from overwhelming the influence of the low-value attributes. Similarly to [10], their architecture features dedicated blocks to perform different stages of the algorithm. Their design is independent of the features of the dataset. However, it uses a fixed considerable amount of hardware resources and can only fit in large scale FPGAs.

Following a different approach, Pu Y. *et al.* [13] designed a kNN accelerator for FPGAs using Open Computing Language (OpenCL). Therefore, the programmer needs only to be aware of the number of work-items for distributing the workload in a manner that optimally uses the available resources of the device. Their architecture outperformed an Intel Core i7-3770K by 148× while having an Energy Efficiency Ratio (EER) of 804×, which is 3× superior to that of an AMD Radeon HD7950 GPGPU. The main disadvantage of their design is the overhead associated with data transfers via the PCI-Express bus, which is identical to that of GPGPUs. Thus, their architecture is only suitable for large workloads. Furthermore, it requires a massive amount of hardware resources, making it unfeasible on smaller devices, such as IoT.

All in all, previous works neglect to provide a reasonable trade-off between flexibility, performance and hardware requirements. In general, they are not parameterized and require major changes to the architecture whenever the requirements of the datasets change. kNN-STUFF presented in this work surpasses previous works in terms of efficiency, is scalable and can be implemented in both small and large scale reconfigurable devices. It has a set of parameters that can be adjusted prior to synthesizing the design to produce different architectures depending on the requirements of the problem and hardware constraints. It can enable high levels of parallelism by instantiating multiple accelerators. Moreover, the accelerators of kNN-STUFF are dataset-agnostic, meaning that they do not require to be re-implemented for different types of datasets.

III. KNN ALGORITHM REVISITED

The kNN algorithm classifies multidimensional samples according to their distance to known training samples. The classification result is derived from the class of the k closest training samples. The algorithm has three stages: 1) distance computation, 2) kNN finder, and 3) query label finder. Also, the kNN algorithm splits the dataset in two parts: the training set with N pre-classified samples with M features each, and the testing set with N' samples with M features each from unknown classes. k is the number of nearest neighbors contributing to the classification. For each of the N' samples in the testing set, the three stages of the kNN algorithm are executed as follows:

1) First, the distances between the testing sample and all samples in the training set are determined. Different metrics can be applied to compute the distance between the two samples (e.g., Euclidean distance, Manhattan distance, among others [20]). The Euclidean distance is the most widely used distance metric in kNN, although other metrics might be more suitable for particular datasets [21]. This work uses the sum of squared differences,
$$d(A, B) = \sum_{i=0}^{M} (A_i - B_i)^2, \tag{1}$$
which is equivalent to the Euclidean distance
$$d_{Euc}(A, B) = \sqrt{\sum_{i=0}^{M} (A_i - B_i)^2} \tag{2}$$
in the context of the kNN algorithm, since
$$d_{Euc}(A, B) > d_{Euc}(A, C) \Rightarrow d(A, B) > d(A, C). \tag{3}$$
The removal of the square root simplifies the implementation of the distance metric in hardware.
2) Second, the distances calculated in the first stage of the algorithm are sorted, and the k smallest values are extracted. In the particular case of the software-only baseline developed for this work, this phase was implemented using a simple insertion sort. Additionally, since only the k smallest distances are required, the insertion sort can be interrupted after the first k elements of the distance vector are determined, reducing its software complexity from $O(N^2)$ to $O(N \times k)$.
3) Finally, the most common class among the k training samples corresponding to the k smallest distances is determined and assigned to the testing sample.

Accordingly, the pseudo-code of the kNN algorithm highlighting its different stages is described in Algorithm 1 and Figure 1 illustrates its operation.

ALGORITHM 1 Pseudo-Code of the kNN Algorithm

```
for all test_samples do
    for all train_samples do
        distances.append(distance(test_sample,
        train_sample))
    end for
    indexes ← getKLowestDistances(distances, k)
    new_class ← getMostFreqClass(indexes, train_set)
end for
```

Complexity-wise, the execution time of the first two phases of the kNN algorithm scale with $O(N \times k)$, whereas the last phase has complexity $O(k)$. Thus, kNN's complexity is given by $O(N \times k + N \times k + k)$, or simply $O(N \times k)$.

IV. CHALLENGES IMPLEMENTING KNN ON FPGA

FPGAs are reconfigurable silicon devices which are capable of implementing highly optimized and parallel digital architectures. They have configurable Processing Elements (PEs) which are programmed to implement any digital function, memory blocks (BRAMs), fixed-point multiply and add units known as Digital Signal Processors (DSPs), and programmable interconnections to connect the elements. Currently, technology has evolved to SoC devices which can combine hard processors with reconfigurable logic. While

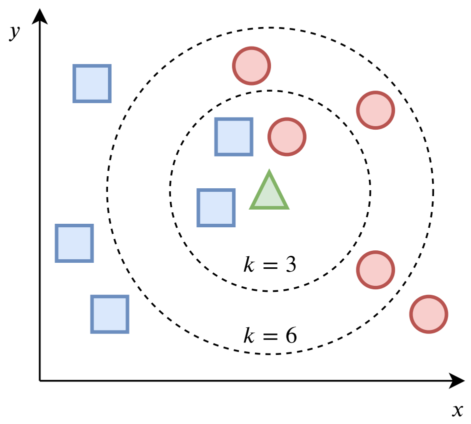


FIGURE 1. Example of the kNN algorithm applied to classify an unknown bi-dimensional sample (represented by the green triangle) using a training set with two classes, blue square and red circle. For $k = 3$, the majority of the neighbors are blue squares, thus the assigned class is also blue square. However, for $k = 6$ most of the nearest neighbors are red circles, which determines that the testing sample should also be classified as a red circle.

the reconfigurable logic offers flexibility to implement optimized custom accelerators for given applications, enhancing their performance, hard processors provide efficient ways of feeding data to the devices and controlling them. Moreover, hard processors easily overcome soft-cores (processors that can be implemented in reconfigurable logic) performance-wise. For example, the dual-core ARM Cortex-A9 included in the Xilinx ZYNQ-7020 SoC can operate up to 650 MHz and is equipped with the ARM NEON Single Instruction Multiple Data (SIMD) engine, capable of performing four 32-bit floating point operations at once. This makes such processors the ideal candidates to execute the software parts of the applications that are not dealt with in the accelerators.

SoC devices alone do not have enough memory capacity to hold all the data necessary to do the computations internally and hence, they are interfaced with Double Data Rate Synchronous Dynamic Random-Access Memories (DDR SDRAMs). Moreover, DDR SDRAMs can be shared with other Input/Output (I/O) sub-systems which facilitate the exchange of data avoiding extra communication channels. The downside is the access overhead introduced by DDR SDRAMs when compared to the internal BRAMs.

Relating the kNN algorithm with the technology, it is expected to spend a significant part of the time on loading data since it requires to constantly evaluate the distance between samples from the training and testing sets.

To avoid the data transfers from the DDR SDRAM to the FPGA become the performance bottleneck, a strategy is to fetch only once each sample from the testing set and compute the distances for each sample from the training set and reduce the number of times that the entire training set is transferred from memory.

A challenge in the implementation of the kNN algorithm is to choose the optimal data format. Even though data from sensors is usually quantified with 8-16 bits, which promotes the use of fixed-point arithmetic, very often the outputs of feature extraction algorithms are single or double-precision floating-points. Even though fixed-point is computed faster than floating-point arithmetic, the effect on the classification of the algorithm may compromise its application.

Moreover, in the kNN algorithm the quality of the results depends on the considered precision. If the distances of a test sample to two samples, one of each class, differ by 2^{-18}, to distinguish between them the implementation would have to support more than 18 bits in the wordlength. However, adopting single-precision floating-point, it supports values as small as 1.17×10^{-38}.

V. KNN ALGORITHM PROFILING

The design of an accelerator is focused on offloading the computation of operations that take the most time to hardware accelerators. Therefore, before establishing the novel architecture it was necessary to profile the execution of a bare implementation of the algorithm in software and evaluate which parts were responsible for the performance bottleneck.

The kNN algorithm was implemented in C language and executed on an Arm Cortex-A9 processor. The application was compiled with the highest optimization level (-O3).

The distance computation phase dominates the execution time of the algorithm, and for increasingly bigger datasets (increasing N and M) the time spent computing the distances between samples tends to 100%. The remaining execution time is mostly spent on the kNN finder phase. In addition, increasing k increases the relative execution time of the kNN finder, due to increasing the insertion sort's execution time ($O(N \times k)$). The time spent during the query label finder is negligible. Considering the above, the devised hardware accelerator only implements the distance computation and kNN finder phases. The query label finder continues to be executed in the processor.

VI. PROPOSED KNN ACCELERATOR ARCHITECTURE

The architecture of the accelerator that executes the first two stages of the kNN algorithm is divided into two logical blocks, the Distance Computing (DistComp), and the Distance Sorting (DistSort), as shown in Figure 2 and detailed in the following subsections. The hardware accelerator computes and sorts the distances using single-precision floating-point arithmetic. Since the accuracy of the kNN is defined only by the correct computation and sorting of the distances between samples, the accuracy of the results computed by the hardware accelerator is the same as a software-only implementation using the Euclidean distance and single-precision floating-point.

A. DISTANCE COMPUTING BLOCK

The DistComp block computes the distance between one testing sample and all the samples from the training set using three single-precision floating-point arithmetic sub-blocks: a subtracter, a multiplier, and an accumulator. These sub-blocks are fully-pipelined such that it consumes a 32-bit word

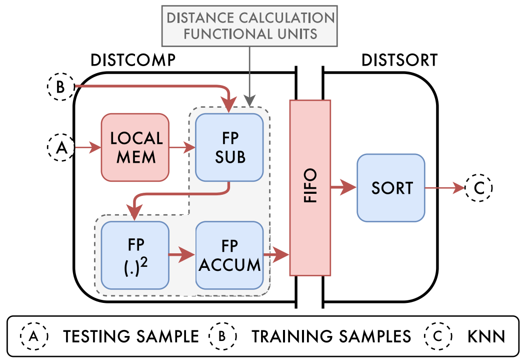


FIGURE 2. Simplified schematics of the devised kNN accelerator. The DistComp block, to the left of the FIFO, is responsible for computing the sum of square differences between one testing sample and all the training samples. The DistSort block, to the right of the FIFO, selects the indexes of the lowest values computed by the input block. Input *A* represents the testing sample, which is stored in the local memory in the DistComp block, while input *B* represents the stream of training samples. The result of the classification is represented by *C*, and consists of the indexes of the training samples which hold the smallest distances to the testing sample. Both the inputs and output of the circuit receive and transmit data sequentially through streaming buses.

per clock cycle. Additionally, this block has a local memory for storing the features of a single testing sample during the computation. The sub-blocks that implement floating-point arithmetic operations receive the operands in a streamed manner and pass the sub-results to the next sub-block in an equally streamed way. A signal that indicates the end of the stream is also propagated through the pipeline. That signal is used to reset the accumulator and enable the write of a new result into the FIFO. The arithmetic sub-blocks are also pipelined to optimize the throughput of the overall system. In total, the DistComp block has a throughput of one 32-bit word every M cycles, and has sixteen pipeline stages. The results produced by the arithmetic sub-blocks are stored in the FIFO, whose write enable is activated whenever the signal that indicates the end of the stream is detected. Overall, the operation of the DistComp block is described in the following paragraph.

First, this block receives a stream containing the features of a testing sample and stores it in its local memory. After receiving all the features of the testing sample, it calculates and stores the number of features per sample, since this value is not known beforehand. Then, in parallel with receiving a stream of all the training samples' features, it calculates the distance from the stored testing sample to each training sample being received. Whenever the number of received features equals the number of features per sample, the computation of the distance between the testing sample and the training sample being received is complete. Thus, the address counter of the local memory is reset, and a signal is propagated to the accumulator to insert a new value in the FIFO and also to reset the accumulator. Figure 3 illustrates the order of operations executed by the DistComp block.

The distance metric used by the kNN algorithm is entirely defined inside the DistComp. Changing it requires only to replace the arithmetic sub-blocks by others that implement a different function. For example, to use the Manhattan distance, the block that calculates the square has to be replaced by one that calculates the absolute value. Modifying the circuit that calculates the distance may only possibly impact the latency of the DistComp block and not its throughput, as long as it is kept fully-pipelined. Thus, changing the metric used for calculating distances does not affect the overall performance of the accelerator.

In addition, the format of the data is solely determined by the sub-blocks that implement the distance calculation. Thus, the entire data format of the accelerator can be easily changed by replacing these sub-blocks by, for instance, fixed-point ones.

B. DISTANCE SORTING BLOCK

The DistSort block receives a sequence of values and holds the k lowest values and respective indexes. After consuming the entire input stream of size N, it produces a sorted sequence of the indexes corresponding to the k smallest values.

This block is a parallel implementation of the insertion sort algorithm, where each incoming value is compared simultaneously with all the current k lowest values and is inserted in the correct place within a single cycle.

The internal architecture of the DistSort block is a cascade of k Sorting Nodes (SortNodes), as shown in Figure 4a. The architecture of a single SortNode is depicted in Figure 4b. Each SortNode evaluates the incoming value, the value of the previous SortNode and its own and decides on what action to perform. If the incoming value is lower than the present value and lower that the value stored in the previous SortNode, the current SortNode forwards its present value and respective index to the next SortNode and receives the value and index that were stored in the previous SortNode. If the incoming value is lower than the present value but higher than or equal to the value stored in the previous SortNode, the SortNode forwards its current value and index to the next SortNode and stores the incoming value and respective index. Otherwise, the incoming value is higher than the current value of the SortNode and it maintains its current value. When all the k SortNodes store a value and an incoming value is read from the FIFO, the highest value among them is discarded. After consuming all the distances computed by the DistComp block, the DistSort block will hold the indexes of the k training samples whose distances to the testing sample are the smallest.

There are three further special cases to be considered which are controlled by the "force load" and "sequence" signals:

1) Whenever a SortNode loads a value for the first time the *force load* signal associated with that SortNode is activated, causing it to load the highest value between the incoming value and the value stored in the previous SortNode and its respective index.
2) After consuming all distances computed by the DistComp block, the signal *sequence* is activated, and all

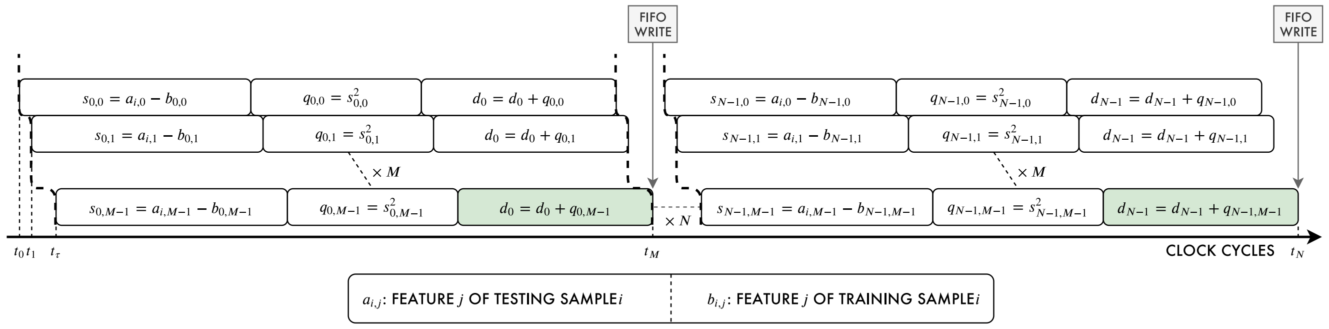

FIFO WRITE
FIFO WRITE
$s_{0,0} = a_{i,0} - b_{0,0}$
$q_{0,0} = s_{0,0}^2$
$d_0 = d_0 + q_{0,0}$
$s_{N-1,M-1} = a_{i,M-1} - b_{N-1,M-1}$
$d_{N-1} = d_{N-1} + q_{N-1,M-1}$
× M
× N
CLOCK CYCLES
t_0 t_1 t_τ t_M t_N
$a_{i,j}$: FEATURE j OF TESTING SAMPLE i
$b_{i,j}$: FEATURE j OF TRAINING SAMPLE i

FIGURE 3. **Order of operations executed by the DistComp block. First, the block receives the features of a testing sample, in stream, and stores them in a local memory (not represented in the diagram). Then, it receives a stream with all the features of the training samples concatenated and calculates, in a pipelined manner, the distance between the testing sample to each of the training samples. From t_0 to t_τ the DistComp block receives the M features of the first training sample. Whenever the number of received features from the training set equals the number of features per sample, the computation of the distance between the testing sample and the training sample being received is complete (t_M). Therefore, the address counter of the local memory is reset, and a signal is propagated to the accumulator to insert a new value in the FIFO. This process repeats until all the training samples are received and processed (t_N). Note that to make it easier to represent, this diagram does not provide an accurate scale.**

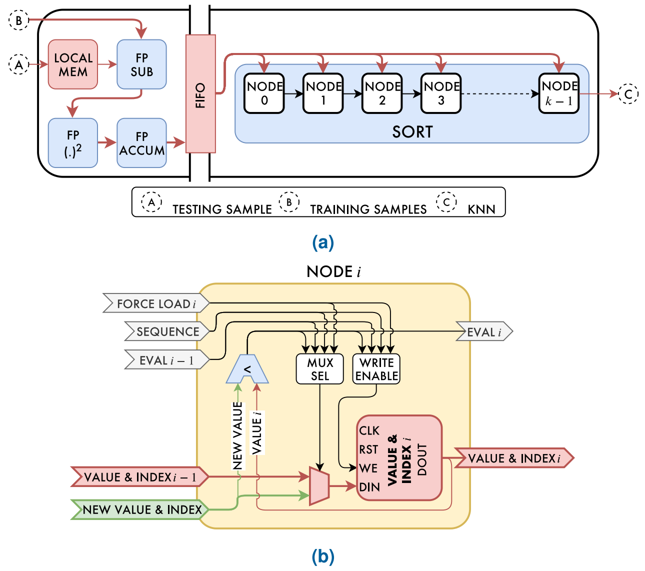


FIGURE 4. **Figure 4a shows the simplified data path of the DistSort block composed by several SortNodes. Figure 4b depicts the control path of a single SortNode.**

the SortNodes shift their current values and indexes to the next SortNode, one per cycle, producing an ordered stream of the indexes corresponding to the k closest training samples to the testing sample.

3) Since the distance between the testing sample and a training sample is calculated iteratively, which takes several cycles, the DistSort block is not fed one distance per cycle, thus it has to stall during the cycles when there is no new distance being produced. For this purpose, both the signals "force load" and "sequence" are activated, disabling the write enable of the registers inside each SortNode.

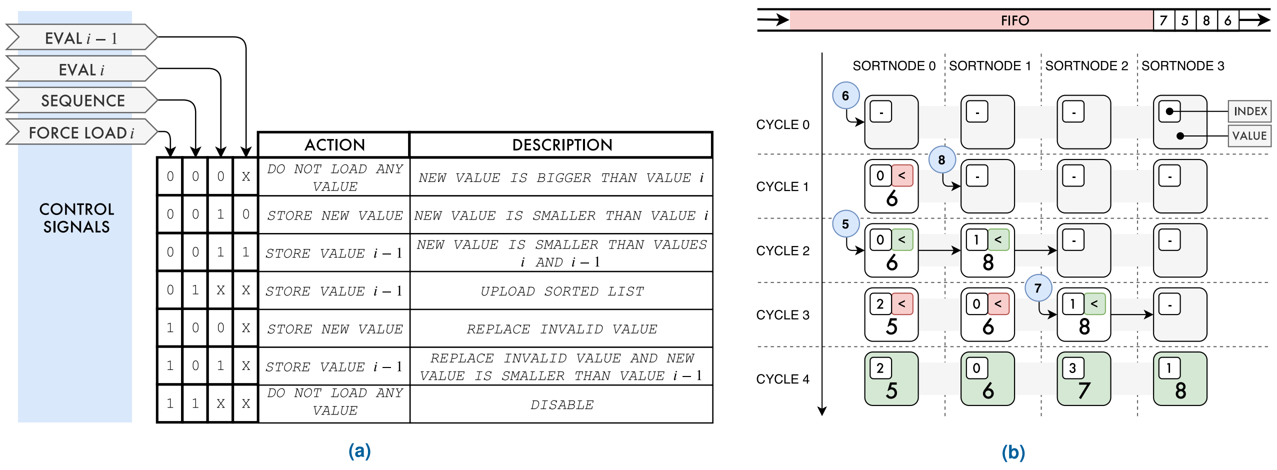
The control of each SortNode is independent of the remaining SortNodes, and thus independent on the number of SortNodes within the DistSort block. Increasing the number of SortNodes only increases the fan-in of the DistSort block due to the need of feeding the incoming value to all SortNodes at once, which may slightly affect the maximum operating frequency of the block.

Figure 5a depicts the control flow of the DistSort block. To exemplify how the DistSort block operates, Figure 5b illustrates a sequence of four values being sorted by the DistSort block, {6, 8, 5, 7}. The first received value, '6' (index '0'), is forced into the first SortNode as all the SortNodes are still empty. The second received value is '8' (index '1'). Since '8' is higher than '6', the first SortNode does not change its value and '8' is forced into the second SortNode. Afterward, '5' (index '2') is received. Since '5' is lower than '6', the first SortNode forwards its value, which causes all the SortNodes to shift right and '5' to be stored into the first SortNode. Finally, '7' (index '3'), is received. Since '7' is higher than both '5' and '6', the first and second SortNodes do not change their values. However, '7' is lower than '8' which causes the third SortNode to forward its value to the fourth SortNode and store '7'. After sorting all the values, the DistSort block exports the indexes of the values in the arrival order of the elements in the sorted sequence, {2, 0, 3, 1}.

VII. KNN SYSTEM

The devised system is a fully autonomous implementation of the kNN algorithm running on the ARM Cortex-A9 processor coupled with a custom accelerator, described previously, in the Programmable Logic (PL). The transfer of data between the DDR SDRAM and the kNN accelerator(s) supported through Direct Memory Access (DMA) engine(s) is shown in Figure 6.

Before the execution starts, both the testing set and the training set are stored in the DDR SDRAM. Then, kNN-STUFF proceeds to classify each testing sample. The processor instructs the DMA engine to read the coordinates of one testing sample and waits for the transaction to complete. Then, the processor orders the DMA engine to read the entire training set from the main memory and waits for it to transfer



				ACTION	DESCRIPTION
0	0	0	X	DO NOT LOAD ANY VALUE	NEW VALUE IS BIGGER THAN VALUE i
0	0	1	0	STORE NEW VALUE	NEW VALUE IS SMALLER THAN VALUE i
0	0	1	1	STORE VALUE $i-1$	NEW VALUE IS SMALLER THAN VALUES i AND $i-1$
0	1	X	X	STORE VALUE $i-1$	UPLOAD SORTED LIST
1	0	0	X	STORE NEW VALUE	REPLACE INVALID VALUE
1	0	1	X	STORE VALUE $i-1$	REPLACE INVALID VALUE AND NEW VALUE IS SMALLER THAN VALUE $i-1$
1	1	X	X	DO NOT LOAD ANY VALUE	DISABLE

FIGURE 5. Description of the operation of the DistSort block. Figure 5a demonstrates the control flow of the DistSort block taking into account the control signals, whereas Figure 5b shows an example of four values, initially in the FIFO, being sorted.

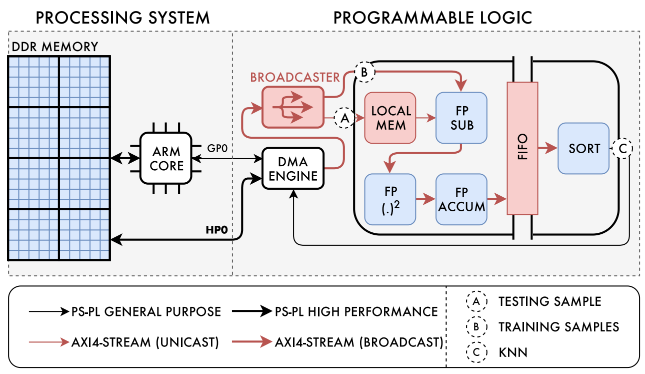


FIGURE 6. Simplified diagram of kNN-STUFF architecture featuring an ARM CPU core, a DDR SDRAM main memory, a DMA engine, a broadcaster and one instance of the devised kNN accelerator.

all the training samples to the accelerator. While the training set is being received by the accelerator, the distances between the stored testing sample and all the training samples are calculated, and the indexes corresponding to the k shortest distances are determined. Then, the processor programs the DMA to retrieve the k indexes corresponding to the k shortest distances. Finally, the processor calculates the most frequent class among the selected training samples and assigns it to the testing sample. The pseudo-code of the software part that controls the DMA engine and executes the non-accelerated part of the kNN algorithm is shown in Algorithm 2.

The limitations in the dataset size are imposed by (1) the size of the internal memory that temporarily stores the testing sample (is mitigated by re-implementing the design using a bigger memory); (2) the maximum stream length supported by the Xilinx DMA Intellectual Property (IP) limits the training set to a maximum size of $N \times M \leq 2^{26} \times 8$ bytes (is mitigated by re-engineering the accelerator to receive the training set in multiple streams). Additionally, since there are only four High Performance (HP) ports available in the Processing System (PS), only four DMAs are worthwhile to be implemented, since using more than four DMAs does not increase the bandwidth that can be achieved.

A. USING MULTIPLE ACCELERATORS

Since the operations in the kNN algorithm are independent, multiple accelerators can be instantiated to explore massive parallelism. The number of accelerators per cluster, the number of nearest neighbors (k) and the size of the accelerators' internal memories are parametrizable in the RTL specification to easily produce different architectures targeting distinct classifier configurations and hardware constraints, without reengineering the entire design.

Implementing multiple accelerators in kNN-STUFF can be done using two distinct methods: connecting the PS to each accelerator through an independent DMA engine, as shown in Figure 7; or grouping the accelerators in clusters that are connected to the processor through a single DMA engine. Additionally, a combination of both is possible where several clusters are connected to the PS using one dedicated DMA engine per cluster, and the accelerators within a cluster share the same connection, as illustrated in Figure 8. This approach allows the best trade-off between hardware requirements and bandwidth utilization since only a limited number of DMAs is implemented while still being possible to parallelize the communication between clusters. Inside a cluster, the training set

ALGORITHM 2 Pseudo-Code That Describes the Program Running in the Processor, Responsible for Controlling the DMA Engine and Executing the Last Phase of the kNN Algorithm

```
for int $i = 0$; $i < N'$; $i = i + 1$ do
    dmaUpload(test_sample[$i$])
    dmaUpload(train_set)
    dmaDownload(indexes)
    new_class[$i$] $\leftarrow$ getMostFreqClass(indexes, train_set)
end for
```

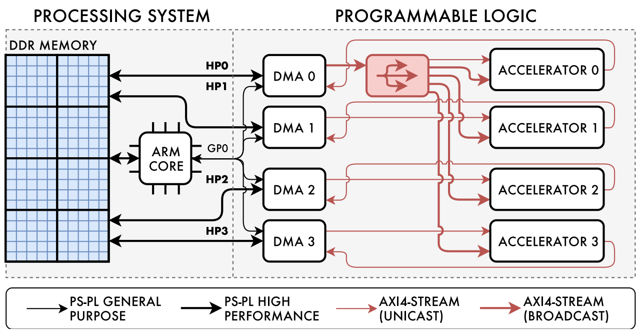


FIGURE 7. Overview of a system that implements four instances of the kNN accelerator, each one connected to the PS by a DMA engine.

is broadcasted to all the accelerators. Hence, the performance and bandwidth utilization is the same as having a dedicated DMA per accelerator.

Regardless of how the multiple accelerators are connected to the PS, kNN-STUFF can be configured to explore various levels of parallelism that are intrinsic to the kNN algorithm. The classification of different testing samples is independent, thus multiple testing samples can be classified simultaneously using various accelerators. There are no dependencies in computing the distance between two samples, therefore multiple accelerators can be used to calculate the distance from one testing sample to several training samples at the same time.

When using multiple accelerators for classifying different test samples, the accelerators are synchronized so that the training set is broadcasted only once to all of them. By doing this, the number of times that the training set is read from memory is reduced by the number of instances running in parallel. Since the first two phases of kNN dominate the execution time, when using two accelerators, classifying two testing samples takes nearly the same time as classifying one. Figure 9 shows the communication diagram for using multiple accelerators to perform several classifications in parallel, while Algorithm 3 illustrates the program running in the processor.

Alternatively, multiple accelerators can be used to classify one testing sample by splitting the training set into subsets that are processed by different accelerators concurrently. This allows the subsets of the training set to be transferred simultaneously to the accelerators, leveraging the bandwidth enabled by using multiple DMA engines. The timing diagram for this scenario is shown in Figure 10, and its software counterpart is described in Algorithm 4.

Both alternatives for exploiting parallelism have the same performance when classifying multiple testing samples. In the first case, D testing samples are evaluated in parallel by the D accelerators and a single training sample is broadcasted to all every clock cycle, corresponding to a bandwidth utilization of one 32-bit word per cycle. In the other case, all accelerators evaluate a single testing sample and D training samples are transferred in parallel every clock cycle, leading to a bandwidth of D 32-bit words per cycle. Hence, classifying multiple testing samples takes the same time in both cases.

When using multiple accelerators for classifying a single testing sample, an additional phase is required in the software part to merge the $C \times k$ results into the final kNN, where C represents the number of clusters and k represents the kNN produced by each cluster for one testing sample. The merge of these results is done with the same method used for the software version of the kNN finder. The main differences are: since there are only $C \times k$ elements in the vector to be sorted, the complexity of the merge is $O(k \times (C \times k))$ instead of $O(N \times k)$; as the output of the accelerators are the indexes of the closest training samples, the distances from the testing sample to the $C \times k$ training samples determined by the accelerators have to be recalculated by the software. Therefore, the introduction of the merge phase adds a small overhead, when compared with the configuration that uses different accelerators to classify different testing samples. This is more evident for small datasets, as shown in Section VIII.

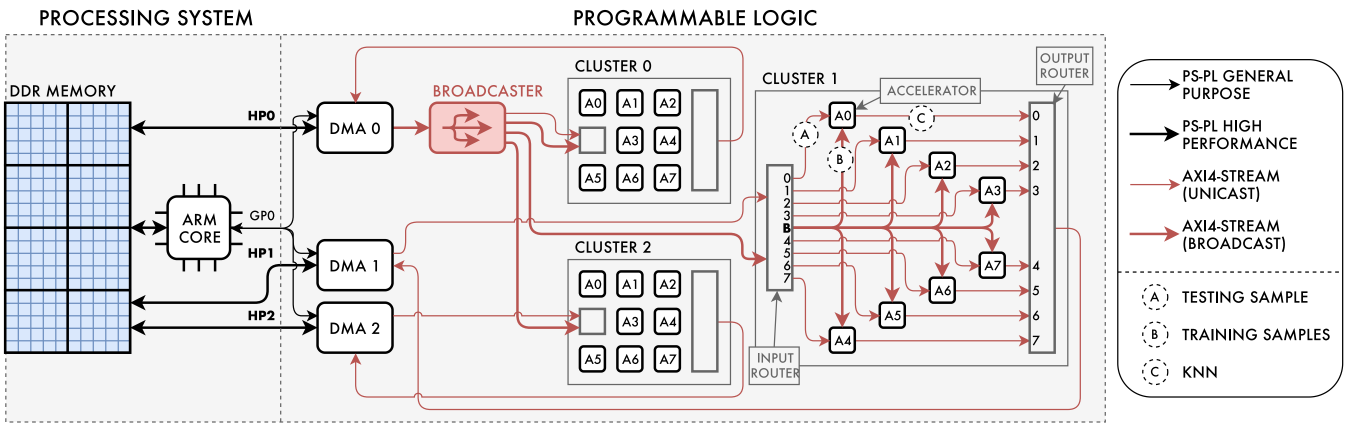


FIGURE 8. Implementation of kNN-STUFF with three clusters with eight accelerators each connected to the PS by independent DMA engines. The eight accelerators within a cluster share the same connection to the PS.

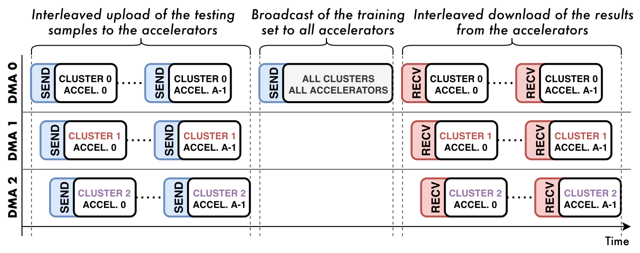

Interleaved upload of the testing samples to the accelerators
Broadcast of the training set to all accelerators
Interleaved download of the results from the accelerators
DMA 0
DMA 1
DMA 2
SEND
RECV
CLUSTER 0 ACCEL. 0
CLUSTER 0 ACCEL. A-1
ALL CLUSTERS ALL ACCELERATORS
CLUSTER 1 ACCEL. 0
CLUSTER 1 ACCEL. A-1
CLUSTER 2 ACCEL. 0
CLUSTER 2 ACCEL. A-1
Time

FIGURE 9. Time diagram illustrating how the DMA engines are coordinated to send the testing samples and broadcasting the training set to the accelerators. Since the training set is broadcasted only once, the overhead associated with sending it is reduced by the number of accelerators.

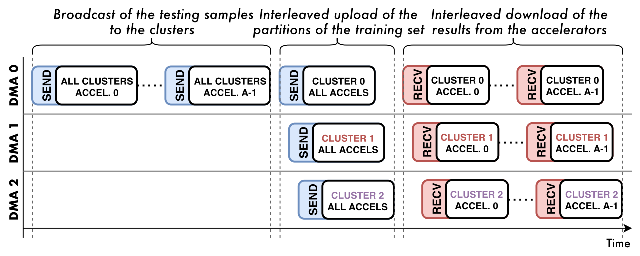


FIGURE 10. Time diagram illustrating how the DMA engines are coordinated to send the testing sample and transfer the training subsets to the accelerators. The communication of the training subsets is done in an interleaved way. Thus, in practise, the training subsets are transferred simultaneously.

ALGORITHM 3 Routine Being Executed by the Processor When Using Multiple Accelerators for Classifying Different Testing Samples. The Number of Clusters Is Represented by C, and A Represents the Number of Accelerators per Cluster

```
for int $i = 0; i < N'; i = i + (C \times A)$ do
    for int $j = 0; j < A; j = j + 1$ do
        for int $k = 0; k < C; k = k + 1$ do
            dmaUpload(dma[$k$], test_sample[$i + k \times A + j$])
        end for
    end for
    dmaBroadcast(train_set)
    for int $j = 0; j < A; j = j + 1$ do
        for int $k = 0; k < C; k = k + 1$ do
            dmaDownload(dma[$k$], indexes[$i + k \times A + j$])
            new_class[$i + k \times A + j$] ←
                getMostFreqClass(indexes[$i + k \times A + j$],
            train_set)
        end for
    end for
end for
```

ALGORITHM 4 Routine Being Executed by the Processor When Using Multiple Accelerators for Classifying One Testing Sample. The Number of Clusters Is Represented by C, and A Represents the Number of Accelerators per Cluster

```
for int $i = 0; i < N'; i = i + A$ do
    for int $j = 0; j < A; j = j + 1$ do
        dmaBroadcast(test_sample[$i + j$])
    end for
    for int $j = 0; j < C; j = j + 1$ do
        dmaUpload(dma[$j$], train_subset[$j$])
    end for
    for int $j = 0; j < A; j = j + 1$ do
        for int $k = 0; k < C; k = k + 1$ do
            dmaDownload(dma[$k$], aux[$k$])
        end for
        indexes[$i + j$]← mergeResults(aux)
        new_class[$i + j$] ←
            getMostFreqClass(indexes[$i + j$], train_set)
    end for
end for
```

When grouping the different accelerators using clusters, it is also possible to classify multiple testing samples while using more than one accelerator to classify a single test sample. In that case, the different testing samples are broadcasted, in sequence, to all clusters. Then, a training subset is transferred to each cluster. Although the accelerators within the same cluster are responsible for classifying different testing samples, the classifying testing samples are the same for all clusters.

VIII. EXPERIMENTAL RESULTS

The kNN-STUFF was evaluated for correctness, by comparing the outputs produced against the expected result from the software baseline implementation for several real and synthetic datasets.

kNN-STUFF was also evaluated in terms of hardware resources, performance and energy efficiency for four scenarios:

Scenario 1 Three clusters featuring eight accelerators each were implemented, and seven real datasets (Iris[1], Wine[2], Breast Cancer Wisconsin[3], Car Evaluation[4], Abalone[5], Bank Marketing[6], and Poker Hand[7]) from [22], [23] were used, considering $k = 4$. The clusters were connected to the PS in such a way that each accelerator classifies a different testing set.

Scenario 2 Four accelerators were implemented and connected to the PS through independent DMA engines. The same seven real datasets from [22], [23] were used, considering $k = 4$. The connections between the accelerators and the PS allow to have a testing sample to be broadcasted to all accelerators, and a subset of the training set is

[1]https://archive.ics.uci.edu/ml/datasets/iris
[2]https://archive.ics.uci.edu/ml/datasets/wine
[3]https://archive.ics.uci.edu/ml/datasets//Breast+Cancer+Wisconsin+(Diagnostic)
[4]https://archive.ics.uci.edu/ml/datasets/car+evaluation
[5]https://archive.ics.uci.edu/ml/datasets/abalone
[6]https://archive.ics.uci.edu/ml/datasets/bank+marketing
[7]https://archive.ics.uci.edu/ml/datasets/Poker+Hand

transferred to each accelerator using an independent DMA engine.

Scenario 3 A single accelerator was implemented, and synthetic random datasets were generated to evaluate the impact of varying the number of training samples and features in the performance of kNN-STUFF. The fixed parameters used for this scenario were $k = 4$, $N' = 1$, and 4 possible classes. Depending on the varying parameter, N or M, the other was fixed to $M = 64$ or $N = 10,000$, respectively.

Scenario 4 A pseudo-random dataset was generated with fixed parameters $N = 10,000$, $N' = 1$, $M = 64$, and 4 possible classes to evaluate the impact of changing k in the hardware requirements and performance of a single accelerator.

Choosing a suitable value for k in the context of the kNN algorithm is a non-trivial task often tightly related to the dataset and the nature of the data being classified. Although such analysis is out of scope of this work, tests showed that $k = 4$ leads to a classification accuracy above 75% of the best achievable for $k \in [1, 21]$ and the used distance metric (sum of squared differences). Hence, in Scenario 1, Scenario 2 and Scenario 3 use $k = 4$.

All scenarios were evaluated on a ZED board featuring a Xilinx ZYNQ-7000 XC7Z020-CLG484 SoC, and Xilinx Vivado 2018.3 was used to synthesize and implement the designs. The PS of kNN-STUFF uses a single core of the ARM Cortex-A9, operating at 650 MHz, and the components in the PL were implemented to operate at 100 MHz.

The baseline implementation corresponds to the single core of the ARM Cortex-A9 inside the ZYNQ device running at 650 MHz. The software part of the kNN-STUFF was compiled with `-O0`, while the baseline was compiled with -O3, enabling all the optimizations including vectorization with the ARM NEON. The `-O0` flag was used because compiler optimizations may violate the control flow of the hardware accelerators, which may lead to incorrect results or deadlocks. Nevertheless, even without software optimizations, kNN-STUFF still significantly outperforms its software-only counterpart, as shown in Section VIII-B. Moreover, since no optimizations are required to achieve such performance results, it is likely that a less powerful processor, or even a soft-processor, could be used instead of the ARM Cortex-A9 in kNN-STUFF, facilitating the adoption of a different device family (e.g., a low-cost FPGA).

A. HARDWARE RESOURCES

The hardware resources, after synthesis, for a system with a single accelerator, considering $k = 4$, in Scenario 1 and Scenario 2 are shown in Table 1. The hardware accelerator is rather lightweight, requiring only 2.4% of the LookUp Tables (LUTs) available in the targeted device. Furthermore, only 9 DSPs are required, which allows instantiating up to 24 accelerators in the XC7Z7020 device.

Moreover, varying k only affects the number of LUTs required to implement the accelerator. Figure 11 shows the variation of LUTs for values of k within the range [1, 128] for Scenario 4. As expected, for larger values of k, the relation between k and the required LUTs to implement a single accelerator is approximately linear. However, for $k \leq 8$, the variation of hardware resources is almost negligible.

TABLE 1. Hardware resources required by a single accelerator with $k = 4$ and the systems corresponding to Scenario 1 and Scenario 2.

	Resource	Usage	Available	Usage [%]
Single Accelerator	LUT	1,263	53,200	2.4
	LUTRAM	15	17,400	0.1
	FF	1,180	106,400	1.1
	BRAM	1	140	0.7
	DSP	9	220	4.1
Scenario 1	LUT	33,376	53,200	62.7
	LUTRAM	734	17,400	4.2
	FF	35,969	106,400	33.8
	BRAM	30	140	21.4
	DSP	216	220	98.2
Scenario 2	LUT	13,099	53,200	24.6
	LUTRAM	594	17,400	3.4
	FF	15,960	106,400	14.8
	BRAM	12	140	8.6
	DSP	36	220	16.4

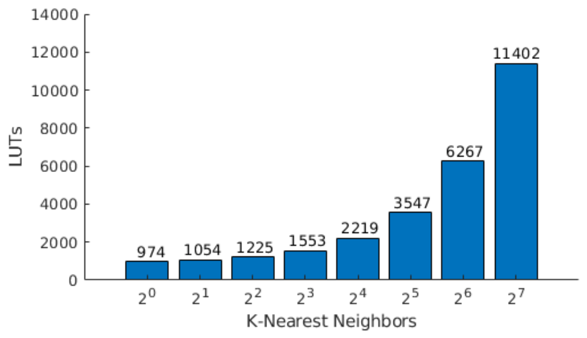


FIGURE 11. Number of LUTs required to implement an accelerator depending on the value of k.

B. PERFORMANCE

The results regarding execution time and attained speedup for the seven real datasets are illustrated in Figure 12. The parameters of each dataset are listed in Table 2. As shown in Figure 12b and Figure 12d, the attained speedup increases with the size of the dataset. However, for small datasets, the synchronization overhead of both scenarios and the merge phase of Scenario 2 overcome the gain provided by implementing more accelerators, as demonstrated by Figure 13a and Figure 13c. Nevertheless, for medium and big datasets the relation between the speedup and the number of accelerators is linear, as shown in Figure 13b and Figure 13d. As expected, duplicating the number of accelerators drops the execution time to half, duplicating the speedup. This is only possible because in Scenario 1 the training set is broadcasted to all the

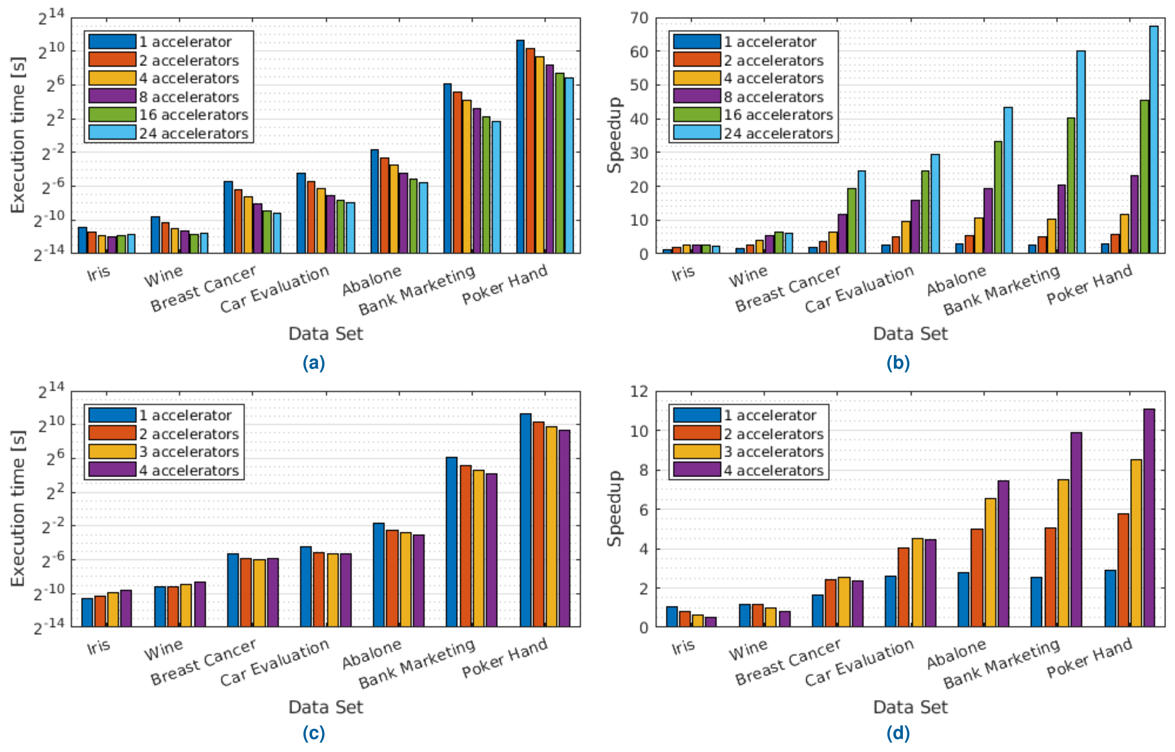


FIGURE 12. **Execution time and speedup attained by kNN-STUFF for seven real datasets using different numbers of accelerators. Figure 12a–Execution time for Scenario 1; Figure 12b–Speedup for Scenario 1; Figure 12c–Execution time for Scenario 2; Figure 12d–Speedup for Scenario 2.**

TABLE 2. **Parameters of the seven real datasets from the *UCI Machine Learning Repository* used in the evaluation of this work.**

Dataset	Training	Testing	Features	Classes
Iris	100	50	4	3
Wine	118	60	13	3
Breast Cancer	379	190	30	2
Car Evaluation	1,152	576	6	4
Abalone	2,784	1,393	8	29
Bank Marketing	30,140	15,071	16	2
Poker Hand	25,010	1,000,000	10	10

accelerators running in parallel and in Scenario 2 the training subsets are transferred simultaneously. Thus, in the former, the communication overhead is reduced by the number of accelerators, and in the latter the bandwidth enabled by the four DMA engines is leveraged.

Overall, the maximum achieved speedup was 67.4×, for the *Poker Hand* dataset using all the 24 accelerators under Scenario 1.

The speedup increases with k, as demonstrated in Figure 14a. This is because the hardware implementation of the kNN finder phase is based on a parallel version of the insertion sort where each incoming value is compared with the previously k sorted values in just one cycle. On the other hand, the software counterpart only compares a pair of values at once, requiring $N \times k$ iterations.

As the incoming value is an input of all k SortNodes, the fan-in of the DistSort block scales linearly with k, which introduces a small timing overhead. Experimental results show that this overhead is negligible as the target operation frequency is still achieved for k as high as 128.

Increasing the size of the training set only affects the attained performance improvements to the point where communicating the entire training set to the accelerator incurs in an overhead proportional to that of the processor reading it from memory, as shown in Figure 14b.

Increasing the number of features per sample decreases the attained speedup, as illustrated in Figure 14c. A possible explanation for this effect involves the use of ARM NEON, loop unrolling and caching effects. For $M \in \{1, 2\}$, the calculation of the sum of square differences involves only one or two subtractions followed by multiplication and the sum of the results. In these scenarios, the NEON engine is not fully exploited, since the operands are insufficient to use all the capabilities of the unit. When $M \geq 4$ the engine is fully used, and all the benefits provided by the NEON are reflected in the algorithm's execution time. For $M \in [4, 64]$, the speedup attained by using kNN-STUFF continues decreasing, but at a much lower rate. This may be associated with minor compiler optimizations allowed by techniques such as loop unrolling as well as caching effects. Eventually, for $M \geq 64$, the speedup becomes nearly constant, which means that the execution time of the kNN algorithm using the ARM Cortex-A9 processor and kNN-STUFF are proportional with the size of the dataset.

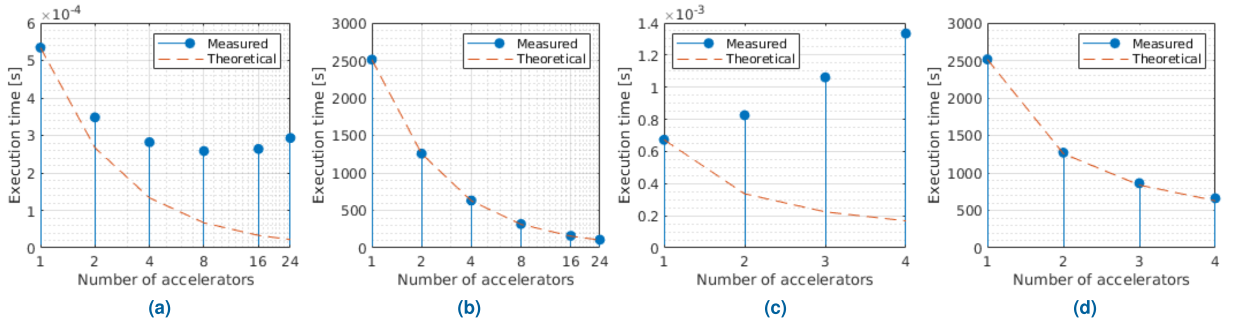


FIGURE 13. Relation between speedup and number of accelerators for two real datasets: the *Iris* dataset (as an example of a small dataset) and the *Poker Hand* dataset (as an example of a large dataset). Figure 13a–*Iris dataset* under Scenario 1; Figure 13b–*Poker Hand* dataset under Scenario 1; Figure 13c–*Iris dataset* under Scenario 2; Figure 13d–*Poker Hand* dataset under Scenario 2.

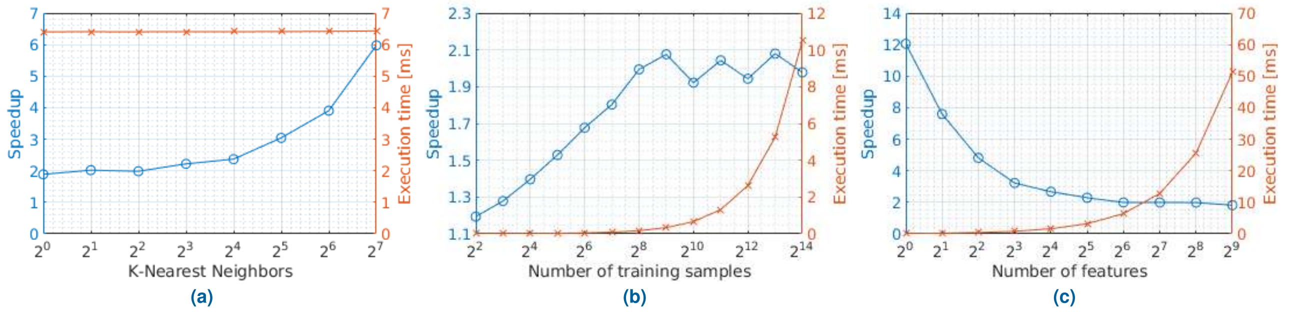


FIGURE 14. Effect of varying the dataset and the parameters of the classifier on the speedup attained by kNN-STUFF. Figure 14a–Effect of varying the *k* nearest neighbors; Figure 14b–Effect of varying the number of training samples (*N*); Figure 14c–Effect of varying the number of features per sample (*M*).

TABLE 3. Static and and dynamic power of the PS and the PL of the device for the configuration regarded in Scenario 1 and Scenario 2.

	Component	Power [mW]
Scenario 1	Device static	159
	PS dynamic	1,533
	PL dynamic	561
Scenario 2	Device static	144
	PS dynamic	1,535
	PL dynamic	165

As an example, the *Poker Hand* has the highest speedup because it has one of the largest training sets (over 25,000 training samples) and has only 10 features per sample. The *Bank Marketing* dataset has a training set of the same order of magnitude (slightly larger), but has a marginally lower speedup because it has 60% more features than the *Poker Hand* dataset.

C. ENERGY EFFICIENCY IMPROVEMENTS

The energy efficiency improvements, calculated based on Xilinx Vivado's power reports for the implemented system, are illustrated in Table 3. To estimate the total energy spent by kNN-STUFF, it was considered that the average instant power of the system is given by the sum of the static power and the dynamic power of both the PS and the PL. For the baseline, the average instant power demand of the processor was approximated by the sum of the static power of the system plus the dynamic power of the ARM Cortex-A9 processor. Since kNN is computationally intensive, this approximation is valid, and it can be assumed that, while executing the algorithm, the processor's average instant power demand is approximately equal to its maximum dynamic power. However, these results do not include the energy spent by the DDR SDRAM. Nevertheless, given the fact that the number of memory accesses when using kNN-STUFF is lower than that of the baseline (or at most the same), the energy spent by the DDR SDRAM when using kNN-STUFF is expected to be lower than the baseline. Therefore, the used approach is even conservative, and the actual energy savings are higher than those corresponding to the determined energy efficiency improvements.

The impact of the data format in the energy efficiency was also assessed. Xilinx Vivado reports that the floating-point arithmetic units responsible for computing the distances account for 12% of the in-chip energy consumption. Furthermore, the memory accesses dominate the energy consumption of the entire system, which can be as high as 90% of the total energy consumption [24]. Consequently, the units responsible for computing the distance account for only 1% of the total energy consumption. On conclusion, changing the representation of the operands from floating-point to another data format does not impact the energy requirements of the system.

Figure 15 illustrates the total energy consumption for each considered dataset and corresponding energy efficiency

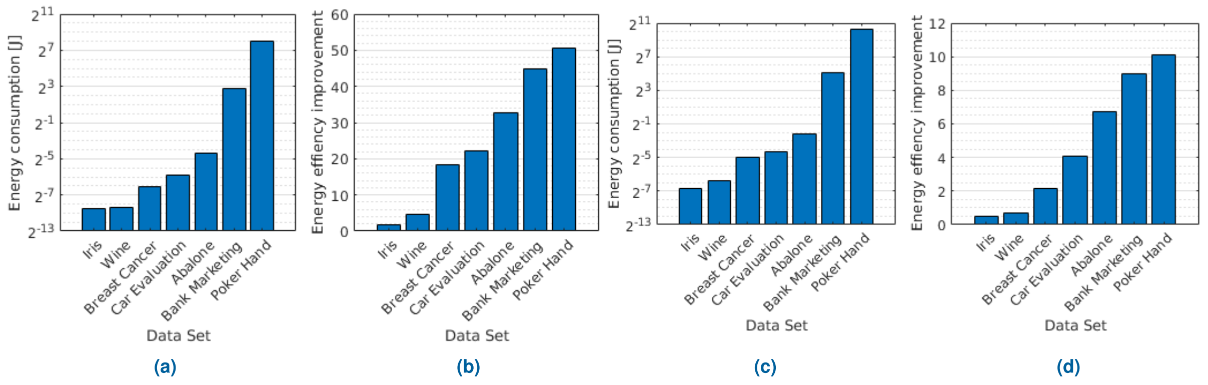


FIGURE 15. Energy consumption and efficiency improvements allowed by kNN-STUFF for seven real datasets. Figure 15a–Energy consumed by kNN-STUFF for seven real datasets under Scenario 1 using all the 24 accelerators. Figure 15b–Energy efficiency improvements allowed by kNN-STUFF for seven real datasets under Scenario 1 using all the 24 accelerators. Figure 15c–Energy consumed by kNN-STUFF for seven real datasets under Scenario 2 using all the four accelerators. Figure 15d–Energy efficiency improvements allowed by kNN-STUFF for seven real datasets under Scenario 2 using all the four accelerators.

improvements. The highest energy efficiency improvement was 50.6×, corresponding to the *Poker Hand* dataset using all the 24 accelerators under Scenario 1.

IX. CONCLUSION

This paper presented a novel hardware implementation of the kNN classifier targeting reconfigurable devices. In contrast with previous proposals, kNN-STUFF is scalable, and allows to automatically scale the system depending on the hardware and power constraints of the targeted devices. When implemented in small reconfigurable devices, each accelerator inside kNN-STUFF requires only 1,263 LUTs. If targeting large scale devices, multiple accelerators or clusters of accelerators can be instantiated.

kNN-STUFF exploits various levels of parallelism enabled by the kNN algorithm, being able to classify multiple testing samples at once using several accelerators or dividing the classification of a single testing sample by various accelerators. While the former allows broadcasting the entire training set to all accelerators, reducing the number of memory accesses by the number of accelerators, the latter allows to simultaneously transfer subsets of the training set to multiple accelerators, leveraging the bandwidth enabled by using various DMA channels.

Results show that using kNN-STUFF allows achieving speedups as high as 67.4× while reducing energy consumption up to 50.6× when compared with a fully optimized software-only implementation of the kNN algorithm running on a single core of an ARM Cortex-A9 CPU. Moreover, it was demonstrated that the speedup and the energy efficiency improvements allowed by kNN-STUFF increase with the number of accelerators, k, and the number of training samples.

A. OPEN SOURCE FRAMEWORK

A framework was created to facilitate the adoption of kNN-STUFF. The framework includes tools that generate both the software and the hardware components automatically, based on the kNN classification parameters.

The kNN-STUFF framework, which allows to replicate the results shown in this paper and includes the RTL description and a comprehensive tutorial showing how to deploy the system using a Xilinx ZYNQ-7000 device, are available at https://github.com/joaomiguelvieira/kNN-STUFF/. The source code for the optimized version of the software-only implementation, regarded as the baseline, alongside its extensive documentation, are available at https://github.com/joaomiguelvieira/kNNSim/.

REFERENCES

[1] Y. Peng, G. Kou, Y. Shi, and Z. Chen, "A descriptive framework for the field of data mining and knowledge discovery," *Int. J. Inf. Technol. Decis. Making*, vol. 7, no. 4, pp. 639–682, Dec. 2008.

[2] J. Saikia, S. Yin, Z. Jiang, M. Seok, and J.-S. Seo, "K-nearest neighbor hardware accelerator using in-memory computing SRAM," in *Proc. ISLPED*, Jul. 2019, pp. 1–6.

[3] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 481–492.

[4] N. Kehtarnavaz, *Smartphone-Based Real-Time Digital Signal Processing, Second Edition* (Synthesis Lectures on Signal Processing). San Rafael, CA, USA: Morgan Claypool, 2018.

[5] E. Onat, "FPGA implementation of real time video signal processing using sobel, robert, prewitt and laplacian filters," in *Proc. SIU*, May 2017, pp. 1–4.

[6] L. Ming, W. Yan, S.-J. Wu, and Y. Wei-Ming, "Implementation of a parallel signal processing system for all-purpose radar," in *Proc. 6th Int. Conf. Signal Process.*, vol. 2, Aug. 2002, pp. 1465–1468.

[7] K. Li, C. Ji, C. Zhong, F. Zheng, and J. Shao, "Application research of energy data acquisition and analysis based on real-time stream processing platform," in *Proc. 6th Int. Conf. Comput. Sci. Netw. Technol. (ICCSNT)*, Oct. 2017, pp. 175–178.

[8] E. S. Manolakos and I. Stamoulias, "IP-cores design for the kNN classifier," in *Proc. ISCAS*, May/Jun. 2010, pp. 4133–4136.

[9] I. Stamoulias and E. S. Manolakos, "Parallel architectures for the kNN classifier—Design of soft IP cores and FPGA implementations," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 2, Sep. 2013, Art. no. 22.

[10] H. M. Hussain, K. Benkrid, and H. Seker, "An adaptive implementation of a dynamically reconfigurable K-nearest neighbour classifier on FPGA," in *Proc. AHS*, Jun. 2012, pp. 205–212.

[11] H. M. 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 *Proc. FPL*, Aug. 2012, pp. 627–630.

[12] M. A. Mohsin and D. G. Perera, "An FPGA-based hardware accelerator for K-nearest neighbor classification for machine learning on mobile devices," in *Proc. HEART*, Jun. 2018, Art. no. 16.

[13] Y. Pu, J. Peng, L. Huang, and J. Chen, "An efficient KNN algorithm implemented on FPGA based heterogeneous computing system using OpenCL," in *Proc. FCCM*, May 2015, pp. 167–170.

[14] F. Canento, A. Lourenço, H. Silva, and A. Fred, "Review and comparison of real time electrocardiogram segmentation algorithms for biometric applications," in *Proc. 6th Int. Conf. Health Inform.*, 2012, pp. 1–9.

[15] H. M. Hussain, K. Benkrid, and H. Seker, "Dynamic partial reconfiguration implementation of the SVM/KNN multi-classifier on FPGA for bioinformatics application," in *Proc. EMBC*, Aug. 2015, pp. 7667–7670.

[16] H. Peng, L. Huang, and J. Chen, "An efficient FPGA implementation for odd-even sort based KNN algorithm using OpenCL," in *Proc. ISOCC*, Oct. 2016, pp. 207–208.

[17] M. Tian, X. Wang, X. Zhang, Z. Yang, and J. Huang, "The implementation of a KNN classifier on FPGA with a parallel and pipelined architecture based on Predetermined Range Search," in *Proc. 13th IEEE Int. Conf. Solid-State Integr. Circuit Technol. (ICSICT)*, Oct. 2016, pp. 1491–1493.

[18] A. Al-Zoubi, K. Tatas, and C. Kyriacou, "Design space exploration of the KNN imputation on FPGA," in *Proc. MOCAST*, May 2018, pp. 1–4.

[19] S. Y. Kung *VLSI Array Processors*, vol. 685. Englewood Cliffs, NJ, USA: Prentice-Hall, 1988,

[20] S. Pandit, "A comparative study on distance measuring approaches for clustering," *Int. J. Res. Comput. Sci.*, vol. 2, no. 1, pp. 29–31, 2011.

[21] L.-Y. Hu, M.-W. Huang, S.-W. Ke, and C.-F. Tsai, "The distance function effect on k-nearest neighbor classification for medical datasets," *SpringerPlus*, vol. 5, no. 1, p. 1304, 2016.

[22] D. Dua and C. Graff. (2017). *UCI Machine Learning Repository*. [Online]. Available: http://archive.ics.uci.edu/ml

[23] S. Moro, P. Cortez, and P. Rita, "A data-driven approach to predict the success of bank telemarketing," *Decis. Support Syst.*, vol. 62, no. 1, pp. 22–31, Jun. 2014.

[24] J. Vieira, "A product engine for energy-efficient execution of binary neural networks using resistive memories," in *Proc. VLSI-SoC*, to be published. [Online]. Available: https://infoscience.epfl.ch/record/267670

JOÃO VIEIRA received the M.Sc. degree in electrical and computer engineering from the Instituto Superior Técnico, Lisbon, Portugal, in 2018, where he is currently pursuing the Ph.D. degree, while doing research at the Signal Processing Systems Research Group, Instituto de Engenharia de Sistemas e Computadores-Investigação e Desenvolvimento. In 2018, he performed a Research Internship with the Processor Architecture Laboratory, EPFL, Switzerland, and the Laboratory for NanoIntegrated Systems, The University of Utah, USA, from January to August 2019. His research interests include high performance computer architectures, near-data processing, hardware/software co-design, and computer architectures applied to quantum computing.

RUI P. DUARTE received the Ph.D. degree from Imperial College London, U.K., in 2014. He is currently a Research Associate with the Electronic Systems Design and Automation (ESDA) Research Group, INESC-ID, Lisbon, Portugal. His research interests include reconfigurable computing, fault-tolerant, and low-power architectures.

HORÁCIO C. NETO received the Ph.D. degree in electrical and computer engineering from the Technical University of Lisbon. He is currently an Associate Professor with the Department of Electrical and Computer Engineering (DEEC), School of Engineering (IST), University of Lisbon. He is also responsible for the Electronic Systems Design and Automation (ESDA) Research Group, INESC-ID, a Research Institute associated with IST, Engineering University. His main research interests are digital systems design and computer architecture, with emphasis in reconfigurable computing.