# Optimized implementation of an improved KNN classification algorithm using Intel FPGA platform: Covid-19 case study

**Article** *in* Journal of King Saud University - Computer and Information Sciences · April 2022

DOI: 10.1016/j.jksuci.2022.04.006

CITATIONS
54

READS
285

**3 authors**, including:

Abedalmuhdi M Almomany
Gulf University for Science & Technology
**21** PUBLICATIONS **146** CITATIONS

SEE PROFILE

Amin Jarrah
Yarmouk University
**47** PUBLICATIONS **244** CITATIONS

SEE PROFILE

Journal of King Saud University – Computer and Information Sciences xxx (xxxx) xxx

Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

# Optimized implementation of an improved KNN classification algorithm using Intel FPGA platform: Covid-19 case study

Abedalmuhdi Almomany *, Walaa R. Ayyad, Amin Jarrah

*Department of Computer Engineering, Hijjawi Faculty for Engineering Technology, Yarmouk University, Irbid 21163, Jordan*

ARTICLE INFO

*Article history:*
Received 1 January 2022
Revised 9 April 2022
Accepted 9 April 2022
Available online xxxx

*Keywords:*
FPGA
DCT-KNN
HLS
HDL
OPENCL

ABSTRACT

The improved k-nearest neighbor (KNN) algorithm based on class contribution and feature weighting (DCT-KNN) is a highly accurate approach. However, it requires complex computational steps which consumes much time for the classification process. A field programmable gate array (FPGA) can be used to solve this drawback. However, using traditional hardware description language (HDL) to implement FPGA-based accelerators requires a high design time. Fortunately, the open computing language (OpenCL) high level parallel programming tool allows rapid and effective design on FPGA-based hardware accelerators. In this study, OpenCL has been used to examine speeding up the DCT-KNN algorithm on the FPGA parallel computing platform through applying numerous parallelization and optimization techniques. The optimized approach of the improved KNN could be used in various engineering problems that require a high speed of classification process. Classification of the COVID-19 disease is the case study used to examine this work. The experimental results show that implementing the DCT-KNN algorithm on the FPGA platform (Intel De5a-net Arria-10 device was used) gives an extremely high performance when compared to the traditional single-core-CPU based implementation. The execution time for our optimized design on the FPGA accelerator is 44 times faster than the conventional design implemented on the regular CPU-based computational platform.



## 1. Introduction

### 1.1. The traditional KNN classifier

The k-nearest-neighbors (KNN) is a supervised machine learning algorithm that is used in both classification and regression problems (Guo et al., 2003). KNN is adopted in several applications such as in text categorization (Chen, 2018), agriculture (Meivel and Maheswari, 2020), medicine (Xing and Bei, 2019), finance (Uludağ and Gürsoy, 2020), facial recognition (Sugiharti and Putra, 2020), economic forecasting (Kück and Freitag, 2021), and heart disease diagnosis (Pawlovsky, 2018). KNN classifies unlabeled data by calculating the distance between each unlabeled datapoint and all other points in the dataset. Then, assign each unlabeled datapoint to the class of the most identically labeled data by finding patterns in the dataset (Ray, 2019). The straight-line distance (also called the Euclidean distance) formula is the most common method to find the distance in KNN. The distance between each test sample $x$ and the training datapoints $X$ where $x_i = (x_1, x_2, x_3 \cdots, x_n)$ and $X_i = (X_1, X, X_3 \cdots, X_n)$ can be calculated using Eq. (1) (Zhang et al., 2017).

$$d(x,X) = \sqrt{\sum_{i=1}^{n} (x_i - X_i)^2} \tag{1}$$

The classification is done based on the k-nearest neighbors (smallest distances), where k is number of nearest neighbors involved in the majority voting process (Zhang, 2016). The class label assigned to the test sample is classified by the majority votes (the class with maximum members in k) of its k-nearest neighbors according to Eqn. (2) (Du and Li, 2019).

$$C(x_i) = \arg\max_k \sum_{X_j \epsilon KNN} C(X_j, Y_K) \tag{2}$$

* Corresponding author.
*E-mail addresses:* emomani@yu.edu.jo (A. Almomany), amin.jarrah@yu.edu.jo (A. Jarrah).

Almomany, A., Ayyad, W. R., & Jarrah, A. (2022). Optimized implementation of an improved KNN classification algorithm using intel FPGA platform: Covid-19 case study. Journal of King Saud University-Computer and Information Sciences.

Production and hosting by Elsevier

https://doi.org/10.1016/j.jksuci.2022.04.006
1319-1578/

where $x_i$ is a test object, $X_j$ is one of its k-nearest neighbors in the training set, $C(X_j, Y_K)$ indicates whether $X_j$ belongs to class $Y_K$. The pseudo code of the classical k-nearest neighbor algorithm is given in Fig. 1.

The KNN algorithm has many features that make it a strong contender to other classification methods. These include simplicity, comprehensibility, and robustness towards noisy data. Further, it gives a relatively high performance (Wu et al., 2017). In general, the KNN classifier has many drawbacks, such as slow execution time, sensitivity to large datasets, sensitivity to the value of parameter K, and high computation steps (Lamba and Kumar, 2016). Thus, due to these limitations, researchers have implemented modifications to improve the performance of the KNN algorithm. Improved versions of the KNN algorithm that have been proposed in the literature to overcome its constraints are SVM KNN (Zhang et al., 2006), Fuzzy KNN (Keller et al., 1985), KNN with Genetic Algorithm (Suguna and Thanushkodi, 2010), KNN with K-Means (Buana et al., 2012), Weight Adjusted KNN (Han et al., 2001), and others.

### 1.2. The improved KNN based on class contribution and feature weighting (DCT-KNN)

The weighted KNN based on the class contribution algorithm (DCT-KNN) is an improved version of traditional KNN (Huang et al., 2018). The DCT-KNN improves the classification accuracy by giving weight for each feature in the examined dataset. To measure the importance of each feature on the accuracy of classification Eq. (3) can be used (Huang et al., 2018).

$$Disc_i = 1 - (Pre_i - pre_t) \tag{3}$$

where $pre_t$ is the average of the accuracy summation of the traditional kNN algorithm when the value of parameter $k$ is 3,5 and 7 on the condition of 5-fold cross validation, and the $Pre_i$ is the same sum of the traditional KNN algorithm on the data but without the *i-th* feature. According to the DCT-KNN, the degree of importance relies on the difference in accuracy, where the smaller the difference is, the more important the feature is.

Next, the corresponding weight ($w_i$) is acquired by normalizing the i-dimensional feature based on Eq. (4) (Huang et al., 2018), where $n$ is the number of features.

$$w_i = \frac{Disc_i}{\sum_{i=1}^{n} Disc_i} \tag{4}$$

Further, the weight of each feature from Eq. (4) is used to find the Weighted Euclidean distance based on Eq. (5) (Song et al., 2019).

$$dist(X1, X2) = \sqrt{\sum_{i=1}^{n} w_i (x_{1i} - x_{2i})^2} \tag{5}$$

The value of the sample number and the distance of the sample to the k-nearest neighbors are factors considered to find the class contribution (CT), according to Eq. (6) (Huang et al., 2018).

$$CT_J = \frac{k}{N_J} + \frac{1}{N_J} \sum d(X, Y_J) \tag{6}$$

where $k$ is the nearest neighbors, $N_J$ is the number of samples of the j-th class in the $k$ neighborhoods, and $\sum(X, Y_J)$ is the sum of the distances among all the samples in the $k$ neighborhoods. The smallest value of $CT_J$ is selected as the final category of the test sample. The discriminant is given in Eqn. (7) (Huang et al., 2018). Fig. 2 shows the detailed procedure of the improved DCT-KNN algorithm.

$$C_x = indesof(\min(CT_j)) \tag{7}$$

According to the DCT-KNN algorithm, the distance between each test point and all other points in the dataset is calculated by considering the weight for each feature. Further, the summation of samples for each class in the k-neighborhood is used to find the CT for each class. Finally, the index of the minimum value of CTs will be the corresponding class labeled (category). It has been noticed that the DCT-KN requires complex computational steps. Therefore, they consume more time, especially in the case of classifying big data.

### 1.3. FPGA platform

High computational processing platforms, such as FPGA, graphics processing unit (GPU), and multi-core, can be utilized to reduce the latency and improve the overall performance (Mittal and Vetter, 2015). FPGAs are high speed computation platforms that can be customized for a proposed application (Kuon et al., 2007). FPGA contains logic blocks that are connected through different configurations. These blocks provide a physical array of logic gates that can be utilized to carry out different processes. Logic blocks include simple gates such as AND, OR, NAND, NOR XOR, and XNOR (Trimberger, 2012). FPGA offers energy efficient programmable hardware resources compared to multi-core-CPUs and GPUs (Abedalmuhdi et al., 2017). Also, FPGAs have a distinctive structure that takes advantage of CPU's complex computation capability and GPU's high speed parallel computing capability (Almomany et al., 2020).

In Intel FPGA devices, the structure usually incorporates adaptive logic modules (ALMs), RAM blocks, and extensive digital signal processing (DSP) blocks. FPGAs can also carry other kinds of blocks, such as phase-lock loops (PLLs), which can adjust the internal clock frequency. ALMs contain at least one lookup table (LUT), each of which is made of one or more flip-flops (FFs). These ALMs are diffused throughout the FPGA fabric, making the FPGAs very amenable for temporally parallel (systolic or pipelined) computations that can be applied to monopolize loop-level concurrency in sev-

**The pseudocode of classical KNN**

**Input:** X: training data, Y: class labels of X, K: number of nearest neighbors.
**Output:** Class of a test sample x.

***Start***
Classify (X,Y,x)
1. ***for*** each sample x ***do***
   Calculate the distance: $d(x, X) = \sqrt{\sum_{i=1}^{n} (x_i - X_i)^2}$
   ***end for***
2. Classify x in the majority class: $C(x_i) = argmax_k \sum_{X_j \in KNN} C(X_j, Y_K)$

***End***

**Fig. 1.** The pseudocode of the classical k-nearest neighbor.

The pseudocode of the improved DCT-KNN

**Input:** i-dimensional dataset A, parameter k, classes.
**Output:** Class of a test sample x.

1- Find discriminate for each feature: $Disc_i = 1 - (Pre_i - pre_t)$
2- Find the weight of each feature: $w_i = \frac{Disc_i}{\sum_{i=1}^{n} Disc_i}$
3- Compute the weighted distance: $dist\ (X1, X2) = \sqrt{\sum_{i=1}^{n} w_i (x_{1_i} - x_{2_i})^2}$
4- Find the average distance of each class in k: $\frac{1}{N_J} \sum d(X, Y_J)$
5- Find the class contribution: $CT_J = \frac{k}{N_J} + \frac{1}{N_J} \sum d(X, Y_J)$
6- Assign the class of the test sample: $C_x = indesof\ (\min\ (CT_i))$

**Fig. 2.** Improved DCT-KNN Algorithm procedure.

eral applications. The device used in this research study—namely, Intel De5a-net Arria-10, this device has adequate resources that can be utilized effectively to synthesize the user's code in various complex applications. The Target FPGA has 427,200 ALMs, which are used to implement several hardware circuits functions, 1518 DSP blocks to ensure the efficient implementation of several floating points operators, and 2713 RAM blocks to store data for the synthesized design.

The traditional approach to program FPGAs is to use hardware description language (HDL) such as VHDL and Verilog. However, HDL is a low-level language that requires the programmer to have a deep knowledge of the underlying hardware architecture (Grewal, 2018). Thus, the introduced high-level synthesis (HLS), such as Vivado (Jarrah et al., 2021) and OpenCL, become more common for the implementation of a proposed design on FPGAs, as it generally abstracts many hardware details (Nane, 2015). Using HLS simplifies the HW implementation and speeds up the design process through reducing and simplifying the overall required steps (Alqudah and Jarrah, 2020).

### *1.4. The open computing language (OpenCL)*

The open computing language (OpenCL) high level parallel programming tool allows rapid and effective design on heterogeneous parallel devices (e.g., CPUs, GPUs, and FPGAs) (Howes and Munshi, 2015). OpenCL decreases the difficulty of programming, execution time, and the hardware cost of parallel computing on heterogeneous devices (xxxx). However, programmers do not require a deep knowledge of hardware design language, such as VHDL, when using OpenCL because of the ability of OpenCL to synthesize high level language specification to produce optimized hardware accelerators (Oninda et al., 2019). Furthermore, OpenCL provides functional portability. Therefore, allowing code execution on several supported devices needs only a few modifications to the host code (Munshi, 7/1/2021.). The design flow for OpenCL, as depicted in Fig. 3, includes three essential stages. These include the emulation stage, performance tuning, and the execution stage (Waidyasooriya et al., 2018). In the emulation phase, the code is executed on a CPU for code verification. Next is the performance tuning phase where the compilation reports and profile information are analyzed to improve the performance by removing the performance bottlenecks. Finally, the real performance is evaluated in the execution phase where the OpenCL kernel code is executed on the FPGA device (Owaida et al., 2011). In general, there is an overhead associated with using the OpenCL tool to program the FPGAs over using hardware description languages such as VHDL in terms of the circuit size. However, the overall performance is close to each other (Bispo and Cardoso, 2017).

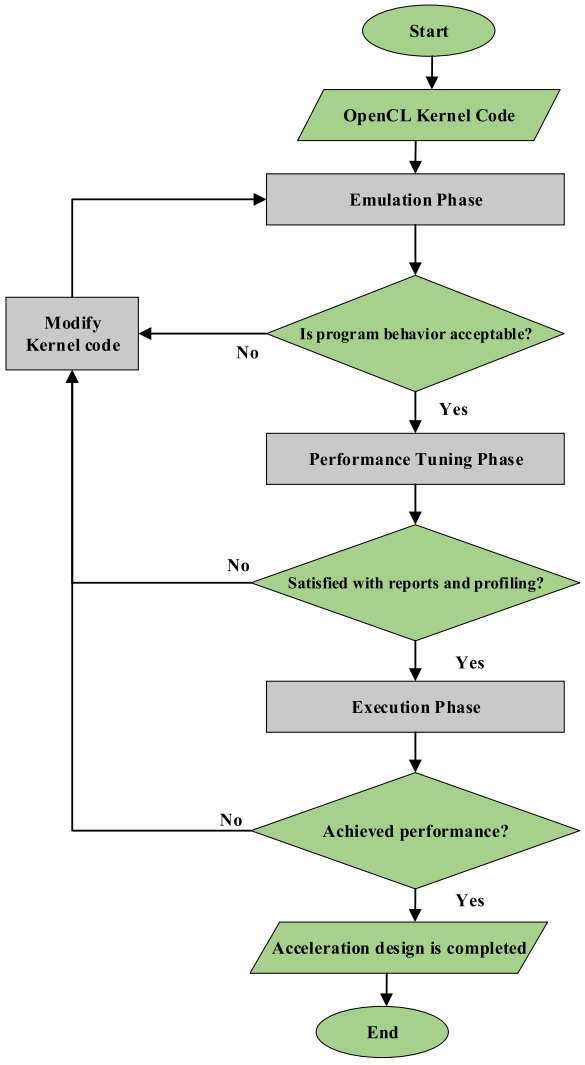



**Fig. 3.** The design flow for OpenCL.

In this study, we designed an efficient parallel implementation of DCT-KNN on the Intel FPGA device using OpenCL parallel programming language to accelerate the DCT-KNN algorithm. This optimized FPGA-based classification implementation could be used in many classification problems that require a high speed of classification. The case study of the Coronavirus disease

classification is utilized to evaluate the effectiveness of the optimized implementation.

Many studies has discussed the process of improving the DCT-KNN Algorithm or similar approaches (Huang et al., 2018; Wang et al., 2021). However, most of these studies focus on improving the accuracy of the classification process. This paper focuses not only on the accuracy of the classification process on a pre-filtered dataset of the COVID-19 disease but also discusses the process of performance tuning on a high-speed FPGA architecture to create an efficient hardware design. The contributions of the proposed paper can be summarized as follow: 1) this study introduces the ability of a parallel high-level computing language (OpenCL) to tune the performance of a common heavy computation classification algorithm to run on high-speed Intel FPGA technology. The efficiency of the created hardware design is evaluated in terms of speed and accuracy; the experimental results show that implementing the DCT-KNN algorithm on the FPGA platform gives an extremely high speedup performance with a high degree of accuracy. 2) Introducing a hybrid sorting approach (MK-sorting) that benefits from the FPGA computing architecture to improve the process of achieving the closest K objects to a reference query object. 3) Evaluating the performance of using high-capability FPGA devices using the OpenCL HLS tool and provide a suggestion on which kind of resources may be needed to increase to improve the overall performance for these kinds of applications.

The rest of the paper is organized as follows: Section 2 discusses numerous state-of-the-art related to accelerating the KNN algorithm. Next, the methodology is provided in Section 3. Then, the experimental results are discussed in Section 4. Finally, Section 5 presents the conclusions.

## 2. Related work

A significant number of researches work on machine learning algorithms for classification (such as the KNN classifier) have been discussed substantially over the last few decades. However, most research studies focused on accelerating the KNN algorithm in heterogeneous computing systems, as well as the reduction of power consumption and the cost of hardware resources.

This literature review section shows numerous studies related to the FPGA-based KNN algorithm that improves the acceleration performance. A brief description of the methodology and the results of each study are presented. Our study has been motivated by the topics covered in this section.

In (Pu et al., 2015), the authors implemented KNN kernel on an FPGA platform using HLS to minimize the influence of external memory access. Low-precision data representation and principal component analysis based filtering (PCAF) are the two data access reduction methods used. The optimized approach, called MPCAF-KNN (Memory-efficient PCAF KNN), employed the features of FPGA. However, the MPCAF-KNN method is tested with several settings and the results showed that this method surpassed the traditional classification methods in terms of execution time and power efficiency.

Since the KNN algorithm is one of the most important classification techniques, the authors in (Peng et al., 2016) presented an efficient method to accelerate the KNN when implemented on an FPGA platform using OpenCL. The FPGA platform has many useful features, such as its parallel pipelined structure. This feature helps optimize the KNN algorithm through designing a particular bubble sort approach. The results concluded that this implementation exceeds the speed of an Intel Core i7-3770k CPU by 148 times when using a bubble sort algorithm-based KNN. In contrast, the speed of FPGA in (Tang, 2016) was 142 times faster than an Intel Core i7-3770k CPU when an odd–even sort algorithm based KNN was implemented using the OpenCL parallel programming tool.

Further, another implementation for KNN on FPGA was presented in (Vestias and Neto, 2014). The experimental results showed that FPGA was 15 times faster than a Xeon E5-2637V3 CPU. However, some parallel platforms, such as GPUs, are appropriate for executing machine learning algorithms. GPUs consume much power because of the costly cooling systems required to deal with heat wastage. In addition, it is hard to parallelize the structure of parts of the algorithms when using GPU. Also, the time consumed by data transmission among CPU and GPU causes high overhead costs (Stamoulias and Manolakos, 2013). In (Hussain et al., 2012), the implementation of the KNN classifier of big data on Xilinx Vertex 5 FPGA was 10 times slower than that on the earlier (2008) GeForce 8800GTX GPU. On the other hand, the FPGA implementation was more energy efficient than the GPU implementation, which surpassed the CPU speed by 100 times.

Other researchers receive benefits from the parallelization features provided in the FPGA to speed up the KNN algorithm. The k-nearest neighbor classifier is sensitive to the size of database and to the parameter K, which is the number of closest neighbors. The algorithm works slowly when the dataset is large. Also, the value of parameter K influences the performance. Therefore, the researchers in (Liu and Khalid, 2018) try to solve these two issues. So, two FPGA implementations of KNN are proposed and compared with similar architecture running on general purpose processor (GPP). The first architecture (A1) achieved a speed up to 76 times over the GPP architecture while the second implementation (A2) attained 68 times speed over the GPP architecture.

In (Muslim et al., 2017), the Intel FPGA SDK for OpenCL (IFSO) is used to examine a speeding up of the KNN algorithm when implemented on an FPGA device. To improve the performance of KNN, a bitonic sorting algorithm was utilized. The performance of an old CPU implementation was compared with the optimized approach. Two FPGA devices were used to implement the optimized design. These included the Intel Stratix A7 and Intel Arria 10 GX. The results showed that the execution time of both FPGA-based accelerators was improved by up to 80 times and the power consumption was reduced by 83%.

Different algorithms have been implemented using HLS-based FPGA in (Theerthagiri et al., 2021). The algorithms were implemented on both GPU and FPGA platforms using the OpenCL tool. A comparison of different performance metrics was experimented for both GPU and FPGA implementations. These included the power consumption and execution time. The experimental results showed that FPGA surpasses GPU in terms of speed and power consumption. This is because FPGA benefits from an OpenCL programming style and use different HLS directives.

Several studies also discussed the opportunities of accelerating the KNN algorithm on high-speed computing FPGAs. Among these studies is one that focused on using an equivalent acceleration device (Terasic DE5) and presented an optimized KNN FPGA-based implementation using the OpenCL framework tool (Tang, 2016). The speedup factor was up to 142; however, this speedup was achieved when compared with a reference software written in MATLAB which may be ten times slower than C/C++ compilers (Andrews, 2012; Liu and Khalid, 2018). It also consumed more resources with a lower or half-clock frequency. Liyuan and Khalid also discussed the acceleration of KNN on comparable intel FPGA devices using the Intel OpenCL framework (Sinhashthita and Jearanaitanakij, 2020), the speedup factor was between 50 and 80 times. However, the reference CPU (Xeon E5-2620) used is slower than the one used in this proposed study, and it is not clear what the software is used as a reference to run on the CPU platform.

Over the last two years, the COVID-19 disease has become a popular research trend due to its negative influence on human beings, where millions of people have died due to the infection of the virus. In (Xiaoming, 2014), several classification machine learning algorithms have been studied with the goal of predicting a possible infection. Different performance metrics were utilized to examine the effectiveness of predictions of different machine learning algorithms. The results concluded that for the COVID-19 dataset used, the KNN algorithm is superior compared to other algorithms in terms of predication accuracy, precision, recall, mean square error, confusion matrix, and kappa score. When reviewing the state of art for several studies related to our work, it was concluded that researchers focused on improving the KNN classification algorithm to overcome its constraints. This study proposes an efficient, optimized implementation of the DCT-KNN algorithm. The proposed design was implemented on an FPGA device by using the OpenCL tool to accelerate the execution time through exploiting the parallelization and optimization features of the FPGA platform. A COVID-19 dataset is used to investigate the performance of our optimized implementation.

## 3. Proposed work and methodology

In the literature there are many modified KNN methods that improve the accuracy of classification, such as in (Xiao and Duan, 2013; xxxx). However, in (Huang et al., 2018), the improved KNN algorithm based on class contribution and feature weighting (DCT-KNN) achieved high accuracy compared to the methods in (Xiao and Duan, 2013; xxxx). It has been noticed that the DCT-KNN algorithm needs complex arithmetic operations, and therefore consumes much time during the classification process. In this work we accelerated the speed of execution of the improved DCT-KNN by adopting several parallelization and optimization techniques of the FPGA computing platform using the OpenCL tool. A flowchart of the methodology is depicted in Fig. 4. For the purpose of performance validation of our optimized implementation, a pre-filtered dataset of the COVID-19 disease has been used. The COVID-19 dataset is taken from the famous website, Kaggle, which offers many public datasets for the aim of data science and machine learning studies. The considered dataset was created synthetically, based on the symptom's guidelines available on the World Health Organization in March 2020 (Booshehri et al., 2013). The dataset includes common symptoms and possible causes of COVID-19. The resulting outcome determined whether an individual was infected with COVID-19. The studied dataset was cleaned using data pre-processing and data cleaning methodologies, so it could be effectively used in our classification algorithm. However, the DCT-KNN algorithm is used to classify the patient into healthy or diseased based on the symptoms. The proposed dataset has a size of 5430x21, where the number of samples in the dataset is 5430 with 21 features. All the data are numbers (0′s and 1′s). A high-speed interface connection or a peripheral component interconnect express (PCIe) connects the target device and the host CPU, providing the opportunity of transferring data very quickly between the computation units.

For performance evaluation, the algorithm was implemented on both single-core-CPU and Intel FPGA devices. Then, different optimization techniques were applied on the FPGA-based implementation to achieve an acceptable speed of classification.

The code for the CPU-based implementation is written using C++ language, while the OpenCL parallel programming tool is used for the FPGA-based implementation. However, the model of parallelism used for execution by OpenCL is a single-work-item which is also called task-parallel model. A single-work-item kernel can process only one work-item during execution. However, the execution of the loop iterations is overlapped. In an ideal case, each iteration of the loop is executed in merely one clock cycle through the creation of an affective pipelined circuit. In the loop pipelining, the computational steps are divided into stages where multiple operations are overlapped (parallel executing), as shown in Fig. 5. In this way, the number of clock cycles is reduced by factor two. However, in the case of high data dependencies, the offline compiler may not be able to create a pipelined data-path. Therefore, it raises the initiation interval (II). This implements the number of cycles (lunch delay) between two successive loop-iterations (where, ideally, the II should be equal to 1) or decreases the clock frequency. Consequently, the performance of the system will be decreased.

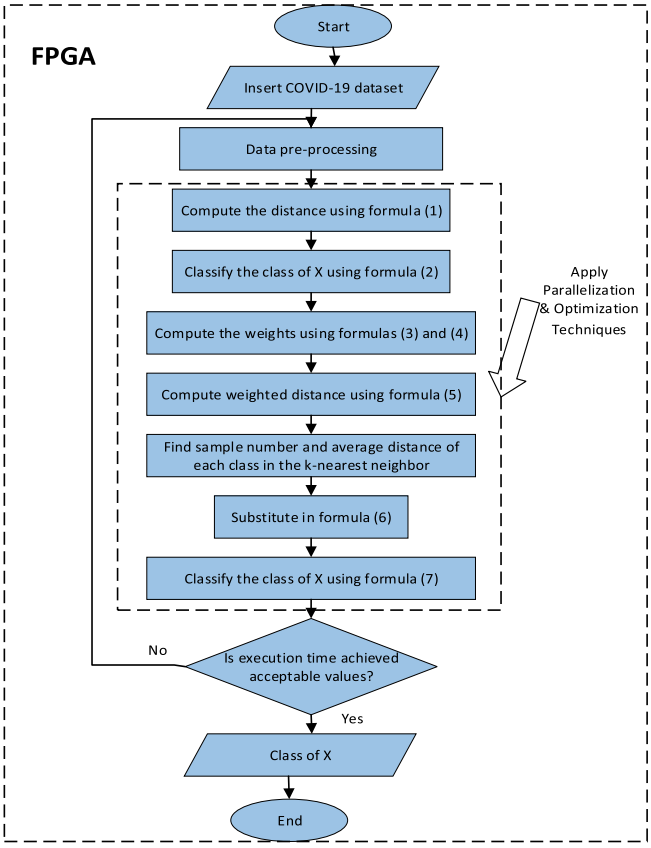



**Fig. 4.** Flowchart of the methodology.

During the compilation of the kernel code (e.g., code.cl), the Intel FPGA compiler generates multiple files. The optimization report file is among these files and contains very useful information such as initiation interval (II), unrolling information, pipelined information, and bottlenecks. By analyzing this information, we can modify the created design to create a more affective pipelined design to get a better performance. In summary, the overall aim of examining the optimization report is to:

- Generate an efficient pipelined data-path for all loops.
- Achieve an initiation interval (II) for each loop in the developed kernel that is equal or close to 1.

In our study, the execution time for the serial code in single-core-CPU was around 3.31 min (198.71 s). In the first step, the design was modified to synthesize on the Intel FPGA device (Intel De5a-net Arria-10) before applying any optimization method (conventional design). This took around 108.249 s, which is a long time for classification. According to the first loop analysis report (before optimization), as shown in Table 1, the II was close to 10 for some

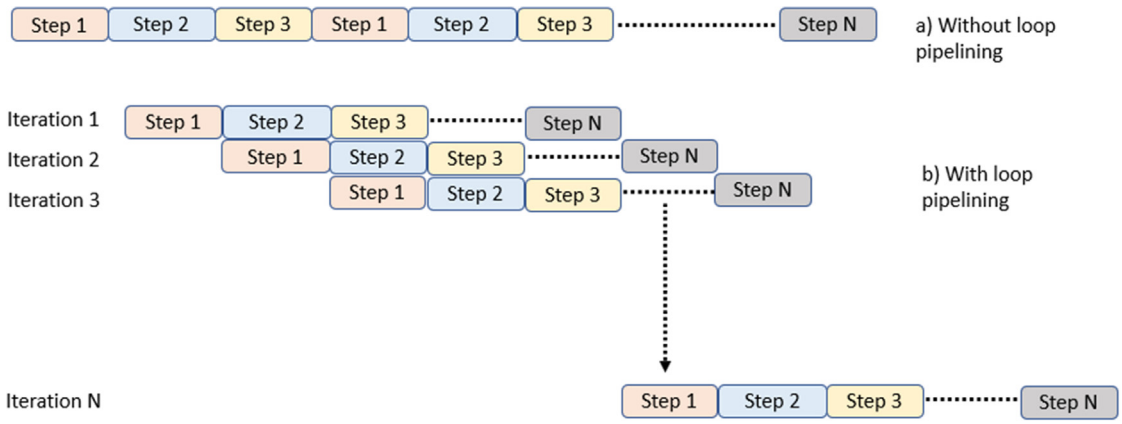



**Fig. 5.** The pipelined data-path generated by FPGA compiler.

steps because of data dependency between loop iterations. Also, the Intel FPGA compiler could not generate a pipelined path for some loops because of an out of order loop-iteration. Therefore, different optimization techniques have been applied for performance tuning.

The following lists the performance improvement techniques that we mainly utilized in our single-work-item kernel for the goal of performance tuning:

- Avoiding serial executions due to data dependency.
- Ignoring loop-carried dependencies due to read-modify-write operations.
- Reducing initiation interval (II).
- Using shift register concept.
- Using loop unrolling technique.
- Using loop merging technique.
- Using loop inlining technique.

### 3.1. Loop unrolling technique

The report in Table 1 indicates a problem of the loop not being pipelined by the compiler. We tried to resolve this problem by utilizing the loop unrolling technique. Loop unrolling can be applied when there is no data dependency between loop computations to reduce the number of clock cycles. In a loop unrolling technique, the HW resources are duplicated based on the loop unrolling factor N (Davidson and Jinturkar, 1995). However, loop unrolling increases the amount of work per clock cycle at the expense of HW resources (Kuznetsov et al., 2012). In our code, loop unrolling was used several times with different unrolling factors. For example, when calculating the average distance in step 4, as shown in Fig. 2, the loop iterations were reduced to half using the unrolling factor 2, as shown in Fig. 6.

Since loop unrolling increase the resource utilization and memory bandwidth, the unroll factor *N* needs to be considered carefully by testing. For instance, if the unroll factor is large, the clock frequency may reduce due to large resource utilization. Thus, we need to reduce the unroll factor *N* in case of low clock frequency. Further, loop-unrolling cannot be implemented in case the kernel does not fit into the FPGA due to a large resource utilization.

### 3.2. Loop merging technique

This occurs when there are multiple sequential loops that have the same loop bounds. Executing these loops consumes extra unnecessary clock cycles and increases the overhead, which negatively impacts the performance (Suganuma et al., 2002). Therefore, a loop merging technique can be applied in this occurrence. Loop merging technique combines the bodies of adjacent loops that have the same bounds to form a single loop. In our code, we rearranged some loop structures that satisfied the requirements of loop merging, into a single large loop. In this way, the merged loops were executed concurrently. Thus, the number of loop iterations were reduced, as were the number of clock cycles. Fig. 7 demonstrates the concept of the loop merging technique. Fig. 7 (a) shows that each loop iteration requires one clock cycle. This includes one clock cycle to enter loop A, four clock cycles to execute loop A, one clock cycle to exit loop A and enter loop B, four clock cycles to execute loop B, and, finally, one clock cycle to exit loop B. As a result, it takes a total of 11 clock cycles to execute both loops (loop A and loop B). Fig. 7(b) shows how the number of clock cycles is reduced after merging loop A and B into one large loop.

**Table 1**
Loop analysis report for steps 3, 4, and 5 in Fig. 2 before optimization.

Bottleneck

| Loop analysis report | | | |
|---|---|---|---|
| KNN computational step | Pipelined | Initiation interval (II) | Details |
| Weighted distance (Step 3 in figure 2) | Yes | ~7 | Data dependency |
| Avg distance in k (Step 4 in figure 2) | No | n/a | Out-of-order inner loop |
| Class contribution (Step 5 in figure 2) | Yes | ~10 | Data dependency |

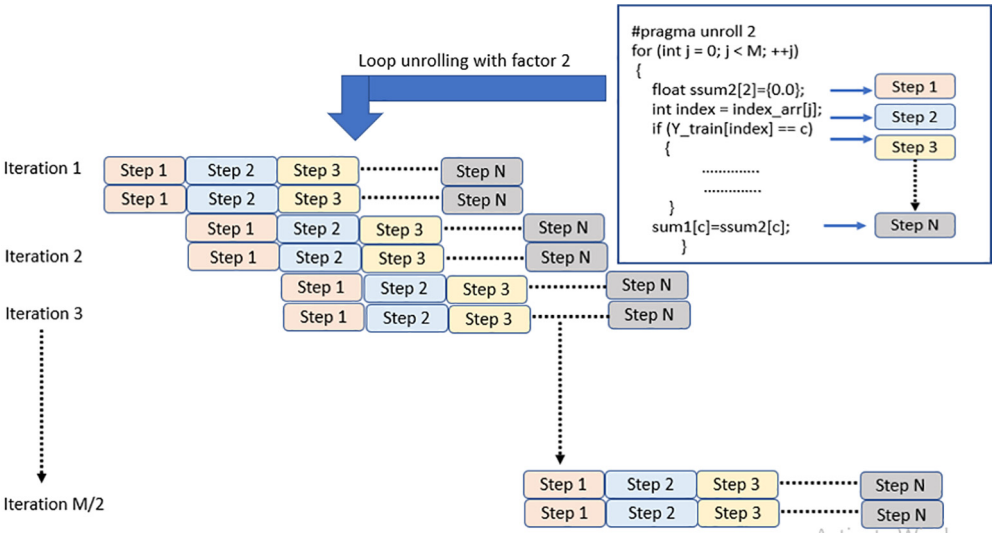



**Fig. 6.** Reducing number of clock cycles using loop unrolling technique for step 4 in Fig. 2.

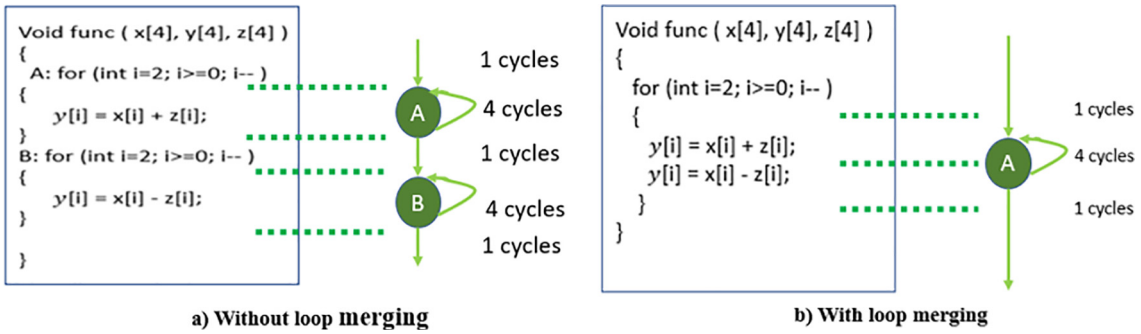



**Fig. 7.** Loop merging technique.

### 3.3. Function inlining technique

Inlining technique is a method to optimize a compiled source code at runtime by substituting the calling functions with its body code. This helps to reduce the overhead of function call (Qian and Hendren, 2005). When the inline function is called, the whole code of the inline function is inserted or replaced at the point of inline function call. This insertion is implemented by the compiler at compile time (Hill et al., 2015). In our code, the Weighted-Euclidean-Distance function (step 3 in Fig. 2) was replaced with a duplicate version of its body when called during compilation, as shown in Fig. 8. As a result, the calling overhead of the function was reduced.

### 3.4. Shift register technique

Creating an effective pipelined circuit with II that is close or equal to 1 is a challenge. However, the concept of a shift register could be used to create a perfect pipelined structure where the programmer tries to remove all data dependencies. In this way, the offline compiler could create a deep pipeline with II close to 1. However, the shift register array is created by unrolling the loop so that the data of the register array can be accessed in parallel. Consequently, the performance of the overall system will improve.

According to the previous loop analysis report shown in Table 1, the II for step 3 in Fig. 2 (calculating the Weighted distance) was close to 7. Thus, the next loop iteration needs to wait 7 clock cycles before the previous iteration finishes its execution, as shown in Fig. 9. This is because of the data dependency of variable s, as depicted in Fig. 10(a). Thus, the shift register concept is used to solve this problem, as shown in Fig. 10(b). We create a shift register array (s (Huang et al., 2018) where the data are accumulated in local variable. Then, the summation of all the values is stored in different external variables (sm) using a single clock cycle by fully unrolling the loop. In this way, the offline compiler can generate a deep pipeline data-path with II equal to 1.

### 3.5. Ignore loop-carried dependencies due to read-modify-write operations

When the updated data of the previous loop-iteration is not required by the next loop-iteration to be lunched, then the next loop-iteration does not need to wait for the previous iteration to write its data to the global memory (no data dependency). In this case, the #pragma ivdep directive can be used to instruct the compiler to ignore data dependencies due to global memory updates. Then, the offline compiler can optimize the loop. In sum, by using the optimization techniques, the execution time for the DCT-KNN algorithm was accelerated successfully. The experimental results are discussed in the next chapter.

### 3.6. Distance vector and sorting technique

Distance computations are fully parallelized since there are no dependencies introduced upon performing distance calculation between a query entity and a reference entity. Several distance

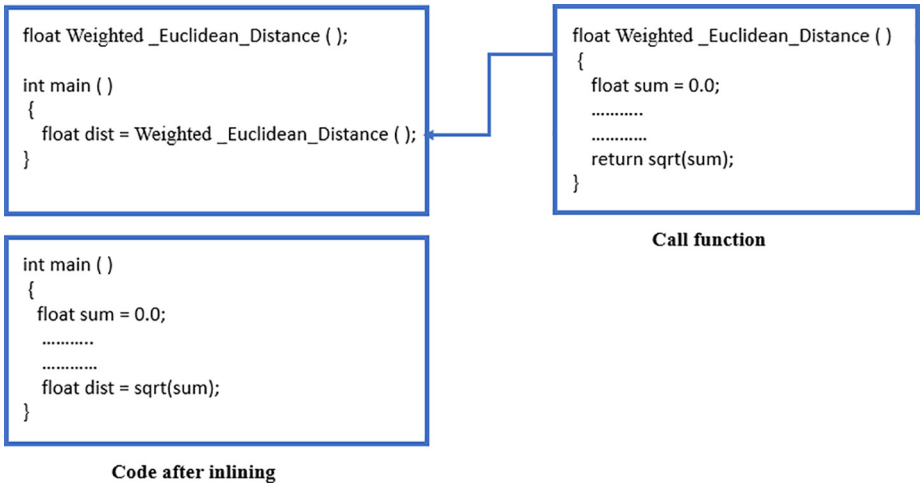



**Fig. 8.** Inlining technique for step 3 in Fig. 2.

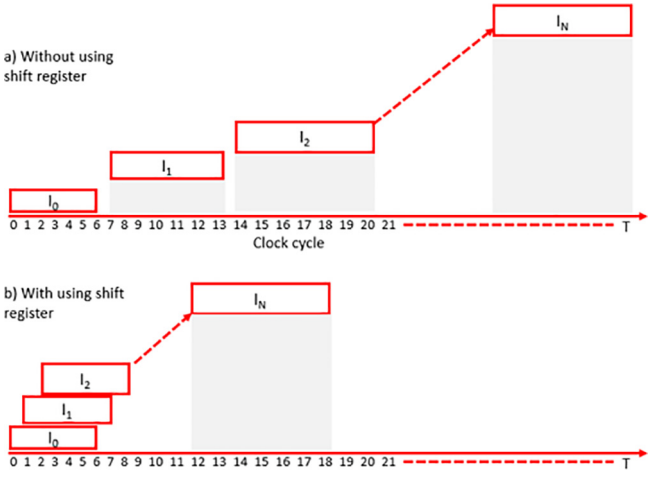



**Fig. 9.** Reducing the II by using the concept of shift register.

values are computed per each clock cycle as the number of function units are duplicated several times through using the loop-unrolling technique, also the shift-registers technique is used to solve dependencies in the created pipelined circuit and increase the operating clock frequency. After gaining the distance vector, the next stage is to sort these distances in the distances vector to locate all the closest neighbours around a new unlabelled data point to predict what class it belongs to. To improve the performance and reduce the memory access time the whole distance vector is stored in a fast access local memory. As the cost of traditional bubble sort degrades the performance significantly and even got worse than execution time on traditional general-purpose computation unit, the sort-function is modified according to the description shown in the Fig. 11 below. The newly adopted sorting algorithm is called MK-sorting, where the distance vectors ranges is divided into *M* sub-vectors. All sub-vectors are sorted in parallel using an odd–even sorting algorithm, where each sub-vector is handled by a separate thread; it is possible to create an efficient pipelined structure with an initiation interval of 1 upon using an odd–even sort technique as proofed in a related study (Tang, 2016). The next stage of sorting is to select the smallest *K* element from each sub-vector and combine them to generate a new small array with a size of *MxK* elements. Finally, as we need the K smallest distances, the small new array is sorted and then the smallest *K* distances are fetched.

We also need to mention that all discussed techniques are working together to create the proposed effective design to run the DCT_KNN algorithm. In summary, we can highlight these facts:

i. Modified the slowest bubble sort to new parallel MK-sorting techniques improve the speedup with a significant factor (more than 50 times).

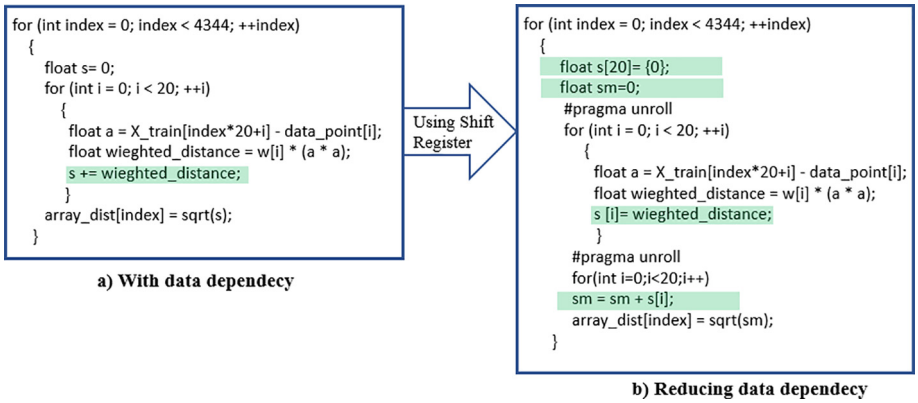



**Fig. 10.** The use of shift register concept in step 3 in Fig. 2.

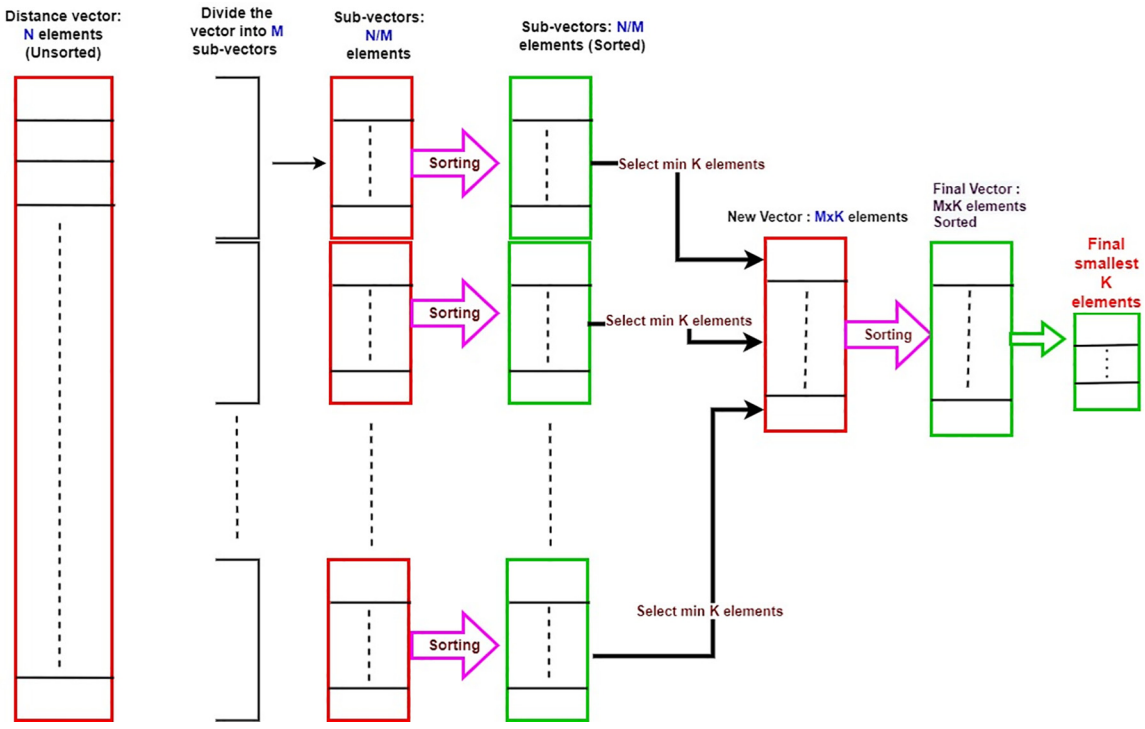



**Fig. 11.** The parallel sorting technique for a distance vector contains *N* elements using MK-sorting.

ii. Solving and reducing dependencies through using local memory and shift-register improves the clock frequency and reduces memory access time and improves the speedup several times.
iii. Loop unrolling techniques increase the amount of work per clock cycle through creating many logic and arithmetic functions and performing several complex operations in a mere one clock cycles. This improves the overall speedup more than ten times.

## 4. Experimental results and discussion

In this study, a pre-filtered COVID-19 dataset was used to investigate the performance of our proposed approach. The considered dataset was created synthetically based on the symptom's guidelines available on the World Health Organization in March 2020 (Booshehri et al., 2013). As shown in Fig. 12; the dataset includes common COVID-19 symptoms, chronic diseases, and daily life activities that cause the infection of COVID-19. The KNN algorithm uses the data shown in Fig. 12 as inputs to make predictions. Predictions are made for each new test sample by searching through the entire training set for the k-nearest neighbors. The resulting classification outcome determines whether the person is COVID-19 diseased or not based on the inputs. The classification results are presented in Fig. 13. The percentages of correct and wrong predictions are depicted in Fig. 14. The number of correct predictions is 5311, while the number of wrong predictions is 119. Therefore, the achieved classification accuracy is 97.8%. To summarize the prediction results of the algorithm, the confusion matrix is provided in Fig. 15. The test set has 4333 true positives, 978 true negatives, 70 false negatives, 49 false positives, which gives an accuracy of 97.8%.

A performance evaluation of the DCT-KNN algorithm is compared when implemented in both the single-core-CPU and the FPGA accelerator. A CPU with Intel Core i7-6700@3.4 GHz with a memory of 16 GB is used to run the serial code. The GNU Compiler Collection (GCC/G++) with an O3 standard level of optimization is used for compilation. The Intel De5a-net Arria-10 device is used to implement the optimized parallel code.

Table 2 presents performance metrics in terms of execution time and clock frequency. The execution time for the serial code on CPU was up to 16 times faster than when implementing the same code on FPGA before optimization (conventional design on FPGA). Comparing the execution time when implementing the parallel code on FPGA before optimization (conventional design on FPGA) with that after optimizing the code, the time was found to be up to 700 times slower. The reason for this is the inability of the offline compiler to generate pipelined data-paths due to high data dependencies among loops. Thus, many optimization methods were applied to accelerate the system, as discussed in the previous section. As a result of optimization, the overall system speed was increased up to 45 times as we achieved the same classification accuracy of the serial code (97.8%).

Further, the frequency should be considered while optimizing the design as it may decrease at some points. Therefore, we should compromise the design to obtain an acceptable higher frequency. In this way, an increased amount of works (number of instructions) can be performed per clock cycle. Consequently, the overall system performance is improved. The proposed design is run on the Intel De5a-net FPGA device at a 255.5 MHz clock frequency.

The area analysis report of the system indicates the values of the estimated area utilization. The resources include adaptive loop-up tables (ALUTs), flip-flops (FFs), RAM memory blocks, and DSP blocks. A comparison of resource utilization before and after optimizing is presented in Table 3. After optimization, ALUTs utilization have increased by 71% while the FFs have increased by 19%. Also, the number of utilized RAMs has increased by 112%, while DSPs usage has also increased by 91%. Thus, we can conclude that optimizing the system increases the amount of resource

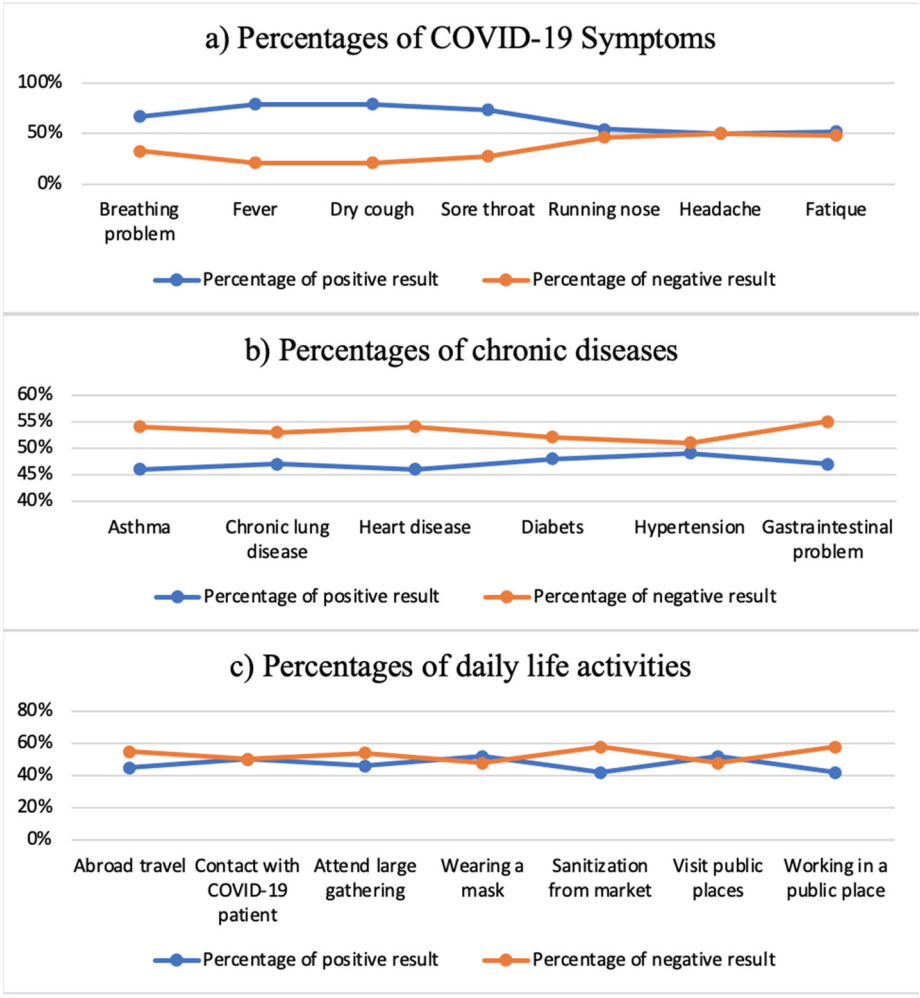



**Fig. 12.** Analysis of COVID-19 dataset.

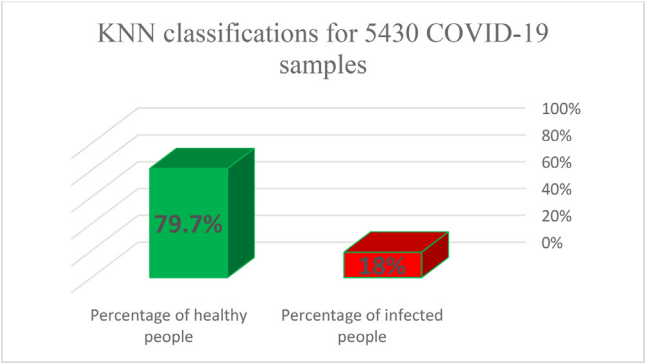



**Fig. 13.** The DCT-KNN classification results.

usage. Consequently, the latency and design complexity may increase. Therefore, we need to compromise the overall system performance. Furthermore, it should be noticed that some resources are reserved when using the FPGA-based OpenCL, as shown in Table 4. This is considered a resource usage overhead when using OpenCL. In contrast, all resources are free when using VHDL or Verilog languages.

Another metric that can be used to compare the performance of our system is the time complexity. Time complexity is used to estimate the performance of an algorithm in terms of speed. The general definition of time complexity is to measure the time consumed by an algorithm to execute each line of code as a function of input size. In our study, the Big-O notation is used to calculate the algorithm complexity based on its running time as the size of input n increases. The O is the growth rate function, while n refers to the length of input. After applying different optimization techniques such as loop unrolling and shift registers techniques, the time complexity for some steps is decreased, as referred to Table 5.

During the run time of the DCT-KNN algorithm, the Euclidean distance function is called N times to calculate the distance between each data point and all other points in the training dataset. This produces a high calling overhead and thus, a large execution time. The time complexity of the Euclidean distance in the DCT-KNN algorithm before optimization is $O(N^2)$, where N is the size of training set. After optimization, the data dependency is removed so the complexity is reduced to be O(N). Furthermore, sorting the distances in ascending order has a time complexity of $O(N^2)$. This comes after applying the loop unrolling technique where the inner loop iterations are performed at the same clock cycle, which means the time complexity decreased to O(N). Also, finding class contribution (CT) has a time complexity of O(N).

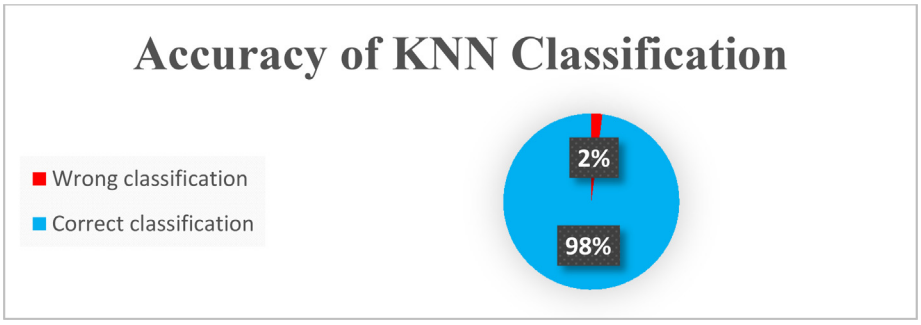



**Fig. 14.** The Accuracy of DCT-KKN classification.

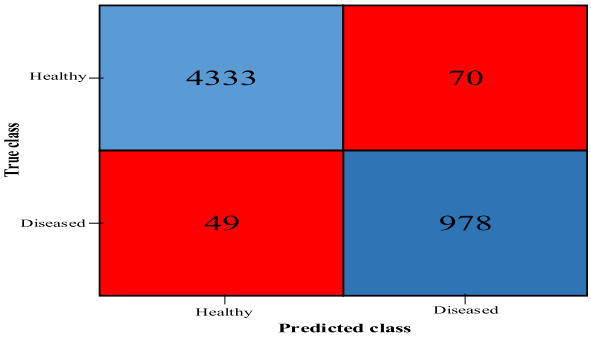



**Fig. 15.** The confusion matrix of the DCT-KNN algorithm.

**Table 2**
Performance metrics for design implementation on both CPU and FPGA.

| Performance metrics | Conventional design on CPU | Conventional design on FPGA | Optimized design on FPGA |
|---|---|---|---|
| Execution time (s) | 198.71 | 3108.249 | 4.45 |
| CLK frequency (MHZ) | 3400 | 207.5 | 255.5 |

Eliminating data dependency using the concept of shift register and implementing loop unrolling technique improves the speed to be O(1) for this step.

In sum, calculating the time complexity for the DCT-KNN algorithm before and after optimization proves the feasibility of using the FPGA accelerator to speed up the proposed system execution time which is our main motivation in this study. A summary of different performance measurements that confirm the effectiveness of using the FPGA accelerator to speed up the heavy computational DCT-KNN algorithm is introduced in Table 6. When comparing our optimized design with the related work in (Vestias and Neto, 2014), our optimized design surpasses the proposed design in (Vestias and Neto, 2014) 3 times, where the achieved speed up in (Vestias and Neto, 2014) was 15 times.

**Table 4**
Amount and percentage of reserved resources in FPGA.

| Resources | Number of reserved resources | Percentage reserved (%) |
|---|---|---|
| ALUTs | 40,650 | 9 |
| FFs | 52,976 | 6 |
| RAM blocks | 283 | 11 |

**Table 5**
Time complexity for the DCT-KNN algorithm.

| Process | Complexity before optimization | Complexity after optimization |
|---|---|---|
| Calculating Euclidean distance | $O(N^2)$ | $O(N)$ |
| Sorting the distances | $O(N^2)$ | $O(N)$ |
| Finding class contribution ($CT$) | $O(N)$ | $O(1)$ |

**Table 6**
Performance comparison when implementing using Intel De5a-net Arria-10.

| Performance parameters | Traditional design on FPGA | Optimized design on FPGA |
|---|---|---|
| Execution time (s) | 3108.249 | 4.45 |
| CLK frequency (MHZ) | 207.5 | 255.5 |
| ALUTs | 80,441 | 121,573 |
| FFs | 63,201 | 125,035 |
| RAM blocks | 1,393 | 1,807 |

**Table 3**
Intel De5a-net Arria-10 resource utilization.

| Resource | After system optimization | | Before system optimization | |
|---|---|---|---|---|
| | Resource utilization | % of utilization | Resource utilization | % of utilization |
| ALUTs | 121,573 | 10 | 80,441 | 6 |
| FFs | 125,035 | 5 | 63,201 | 3 |
| RAMs | 1,807 | 73 | 1,393 | 34 |
| DSPs | 153,5 | 5 | 79.5 | 3 |

## 5. Conclusions

In this paper, an optimized performance tuning approach is proposed to confirm the feasibility of using FPGA accelerator technology to accelerate the computationally complex DCT-KNN algorithm. The OpenCL high-level abstraction language is used to decrease the complexity of the hardware solutions. A COVID-19 dataset was used to evaluate the performance of the proposed work. The speed of execution was compared on both a single-core-CPU device and an Intel FPGA device platform. The regular CPU took 3.31 min to execute the serial KNN code. When implementing the design on the FPGA device, several optimization techniques were applied to improve the overall performance, such as loop unrolling. Thus, the archived speed was 4.45 s. The experimental results conclude the effectiveness of our proposed design, where the system is accelerated 44 times faster when compared to the regular CPU.

## References

Abedalmuhdi, A., Wells, B.E., Nishikawa, K., 2017. Efficient particle-grid space interpolation of an FPGA-accelerated particle-in-cell plasma simulation. In: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 76–79. https://doi.org/10.1109/FCCM.2017.63.

Almomany, A., Al-Omari, A., Jarrah, A., Tawalbeh, M., Alqudah, A., 2020. An OpenCL-based parallel acceleration of a Sobel edge detection algorithm Using Intel FPGA technology. S. Afr. Comput. J. 32 (1).

Alqudah, E., Jarrah, A., 2020. Parallel implementation of genetic algorithm on FPGA using Vivado high level synthesis. Int. J. Bio-Inspired Comput. 15 (2), 90–99.

Andrews, T., 2012. Computation Time Comparison between Matlab and C++ Using Launch Windows. California Polytechnic State University, San Luis Obispo.

Bispo, J., Cardoso, J.M.P., 2017. A MATLAB subset to C compiler targeting embedded systems: A MATLAB Subset to C Compiler Targeting Embedded Systems. Softw. Pract. Exp. 47 (2), 249–272.

Booshehri M., Malekpour A., Luksch P., An improving method for loop unrolling, arXiv preprint arXiv:1308.0698, 2013.

Buana, P.W., Jannet, S., Putra, I., 2012. Combination of k-nearest neighbor and k-means based on term re-weighting for classify Indonesian news. Int. J. Comput. Appl. 50 (11), 37–42.

Chen S., K-nearest neighbor algorithm optimization in text categorization. In: IOP Conference Series: Earth and Environmental Science, 2018, vol. 108, no. 5: IOP Publishing, p. 052074.

I. Corporation. Intel FPGA SDK for OpenCL Overview. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html (accessed 8/13/2021).

Davidson J. W., Jinturkar S., An aggressive approach to loop unrolling, Citeseer, 1995.

Du S., Li J., Parallel processing of improved knn text classification algorithm based on Hadoop. In: 2019 7th International Conference on Information, Communication and Networks (ICICN), 2019: IEEE, pp. 167-170.

Grewal, A. (2018). Review Report on VHDL (VHSIC Hardware description language), Language, 27, 28.

Guo, G., Wang, H., Bell, D., Bi, Y., Greer, K., 2003. KNN model-based approach in classification. In: OTM Confederated International Conferences on the Move to Meaningful Internet Systems. Springer, pp. 986–996.

Han, E.-H.-S., Karypis, G., Kumar, V., 2001. Text categorization using weight adjusted k-nearest neighbor classification. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. Springer, pp. 53–65.

Hari H. Symptoms and COVID Presence. Kaggle. https://www.kaggle.com/hemanthhari/symptoms-and-covid-presence (accessed 8/11/2021.

Hill K., Craciun S., George A., Lam H., Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA. In: *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015, pp. 189-193, doi: 10.1109/ASAP.2015.7245733.

Howes L., Munshi A., The OpenCL Specification, 2015.

Huang J., Wei Y., Yi J., Liu M., An improved kNN based on class contribution and feature weighting. In: 2018 10th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), 2018: IEEE, pp. 313-316.

Hussain H. M., Benkrid K., Seker H., An adaptive implementation of a dynamically reconfigurable K-nearest neighbour classifier on FPGA. In: 2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2012: IEEE, pp. 205-212.

Jarrah, A., Almomany, A., Alsobeh, A.M.R., Alqudah, E., 2021. High-performance implementation of wideband coherent Signal-Subspace (CSS)-based DOA algorithm on FPGA. J. Circuits Syst. Comput. 30 (11), 2150196.

Keller, J.M., Gray, M.R., Givens, J.A., 1985. A fuzzy k-nearest neighbor algorithm. IEEE Trans. Syst. Man Cybern. 4, 580–585.

Kück, M., Freitag, M., 2021. Forecasting of customer demands for production planning by local k-nearest neighbor models. Int. J. Prod. Econ. 231, 107837.

Kuon, I., Tessier, R., Rose, J., 2007. FPGA Architecture: Survey and Challenges. Publishers Inc..

Kuznetsov, V., Kinder, J., Bucur, S., Candea, G., 2012. Efficient state merging in symbolic execution. ACM Sigplan Notices 47 (6), 193–204.

Lamba, A., Kumar, D., 2016. Survey on KNN and its variants. Int. J. Adv. Res. Comput. Commun. Eng. 5 (5), 430–435.

Liu L., Khalid M. A., Acceleration of k-Nearest Neighbor Algorithm on FPGA using Intel SDK for OpenCL. In: 2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS), 2018: IEEE, pp. 1070-1073.

Liu, L., Khalid, M.A.S., 2018. Acceleration of k-nearest neighbor algorithm on FPGA using Intel SDK for OpenCL. In: 2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS), pp. 1070–1073. https://doi.org/10.1109/MWSCAS.2018.8623861.

Meivel S., Maheswari S., Standard Agricultural Drone Data Analytics using KNN Algorithm, 82, 206-215, 2020.

Mittal, S., Vetter, J.S., 2015. A survey of CPU-GPU heterogeneous computing techniques. ACM Comput. Surv. (CSUR) 47 (4), 1–35.

Munshi A., The OpenCL Specification 1.0. Accessed: 7/1/2021.

Muslim, F.B., Ma, L., Roozmeh, M., Lavagno, L., 2017. Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis. IEEE Access 5, 2747–2762.

Nane, R. et al., 2015. A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 35 (10), 1591–1604.

Oninda M. A. M., FPGA-Based Acceleration of the Self-Organizing Map (SOM) Algorithm using High-Level Synthesis, University of Windsor (Canada), 2019.

Owaida M., Bellas N., Daloukas K., Antonopoulos C. D., Synthesis of platform architectures from OpenCL programs. In: 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, 2011: IEEE, pp. 186-193.

Pawlovsky A. P., An ensemble based on distances for a kNN method for heart disease diagnosis. In: 2018 International Conference on Electronics, Information, and Communication (ICEIC), 2018: IEEE, pp. 1-4.

Peng H., Huang L., Chen J., An efficient FPGA implementation for odd-even sort based KNN algorithm using OpenCL. In: 2016 International SoC Design Conference (ISOCC), 2016: IEEE, pp. 207-208.

Pu Y., Peng J., Huang L., Chen J., 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, 2015: IEEE, pp. 167-170.

Qian, F., Hendren, L., 2005. A study of type analysis for speculative method inlining in a JIT environment. In: International Conference on Compiler Construction. Springer, pp. 255–270.

Ray S., A quick review of machine learning algorithms. In: 2019 International conference on machine learning, big data, cloud and parallel computing (COMITCon), 2019: IEEE, pp. 35-39.

Sinhashthita, W., Jearanaitanakij, K., 2020. Improving KNN algorithm based on weighted attributes by pearson correlation coefficient and PSO fine tuning. In: 2020–5th International Conference on Information Technology (InCIT), pp. 27–32. https://doi.org/10.1109/InCIT50588.2020.9310938.

Song X., Xie T., Fischer S., A memory-access-efficient adaptive implementation of kNN on FPGA through HLS. In: 2019 IEEE 37th International Conference on Computer Design (ICCD), 2019: IEEE, pp. 177-180.

Stamoulias, I., Manolakos, E.S., 2013. Parallel architectures for the kNN classifier–design of soft IP cores and FPGA implementations. ACM Trans. Embed. Comput. Syst.ems (TECS) 13 (2), 1–21.

Suganuma T., Yasue T., Nakatani T., An empirical study of method inlining for a java just-in-time compiler. In: Proceedings of the Java Virtual Machine Research and Technology Symposium, 2002: USENIX Association Berkeley, CA, pp. 91-104.

Sugiharti E., Putra A., 2020. Facial recognition using two-dimensional principal component analysis and k-nearest neighbor: a case analysis of facial images, J. Phys.: Conf., 1567(3) IOP Publishing, p. 032028.

Suguna, N., Thanushkodi, K., 2010. An improved k-nearest neighbor classification using genetic algorithm. Int. J. Comput. Sci. Issues 7 (2), 18–21.

Tang Q. Y., FPGA Based Acceleration of Matrix Decomposition and Clustering Algorithm Using High Level Synthesis, University of Windsor (Canada), 2016.

Theerthagiri, P., Jeena Jacob, I., Usha Ruby, A., Yendapalli, V., 2021. Prediction of COVID-19 possibilities using K-nearest neighbour classification algorithm. Int. J. Curr. Res. Rev. 13 (06), 156.

Trimberger, S.M., 2012. Field-Programmable Gate Array Technology. Springer Science & Business Media.

Uludağ, O., Gürsoy, A., 2020. On the financial situation analysis with KNN and naive Bayes classification algorithms. J. Inst. Sci. Technol. 10 (4), 2881–2888.

Vestias M., Neto H., Trends of CPU, GPU and FPGA for high-performance computing. In: 2014 24th International Conference on Field Programmable Logic and Applications (FPL), 2014: IEEE, pp. 1-6.

Waidyasooriya H. M., Hariyama M., Uchiyama K., Design of FPGA-based Computing Systems with OpenCL. Springer, 2018.

Wang, Q., Wang, S., Wei, B., Chen, W., Zhang, Y., 2021. Weighted K-NN classification method of bearings fault diagnosis with multi-dimensional sensitive features. IEEE Access 9, 45428–45440. https://doi.org/10.1109/ACCESS.2021.3066489.

Wu, X., Wang, S., Zhang, Y., 2017. Survey on theory and application of k-Nearest-Neighbors algorithm. Comput. Eng. Appl. 53 (21), 1–7.

Xiao, H.-H., Duan, Y.-M., 2013. Improvement of kNN algorithm, based on attribute value correlation distance. Comput. Sci. 40 (11a), 157–159.

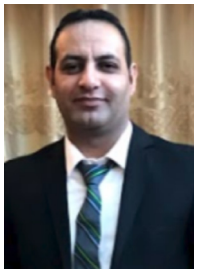
Xiaoming, Y., 2014. Weighted kNN classification algorithm based on average distance of class. J. Comput. Appl. 23 (2), 128–132.

Xing, W., Bei, Y., 2019. Medical health big data classification based on KNN classification algorithm. IEEE Access 8, 28808–28819.

Zhang, Z., 2016. Introduction to machine learning: k-nearest neighbors. Ann. Transl. Med. 4 (11).

Zhang H., Berg A. C., Maire M., Malik J., SVM-KNN: Discriminative nearest neighbor classification for visual category recognition. In: 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), 2006, vol. 2: IEEE, pp. 2126-2136.

Zhang, S., Li, X., Zong, M., Zhu, X., Cheng, D., 2017. Learning k for knn classification. ACM Trans. Intell. Syst. Technol. (TIST) 8 (3), 1–19.



**Abedalmuhdi Almomany** received his BSc in Computer Engineering from Yarmouk University in 2005, the PhD in Computer Engineering from University of Alabama in Huntsville in 2017. In 2017 he joined the department of Computer Engineering, College of Engineering, Yarmouk University, Irbid, Jordan, where he serves as a professor in the CPE department. His research interests include Reconfigurable computing, Parallel processing and Embedded systems.
He received grants from different institutes to pursue his research.

**Amin Jarrah** Biographical notes: Amin Jarrah received his MSc and PhD in Computer Engineering from the Jordan University of Science and Technology and The University of Toledo, USA, in 2010 and 2014, respectively. During 2014–2016, he was appointed as an Embedded Software Engineer in Magna Electronics and TATA group companies. In January 2016, he joined the Yarmouk University as an Assistant Professor. Currently, he is an Associate Professor at the Department of Computer Engineering at Yarmouk University, Irbid, Jordan.
His research is in the area of real-time embedded hardware implementation, high performance computing, parallel processing using FPGAs, system on chip (SoC), and GPU.

**Walaa R. Ayyad** is a M.Sc. student at Computer Engineering Department at Yarmouk University, Irbid, Jordan. She earned her B.Sc. degree in Computer Engineering from Yarmouk University, in 2014. Her research interests are in the areas of parallel processing using FPGAs, GPU, machine learning, artificial intelligence, computer vision, and face recognition. She was awarded for her undergraduate graduation project since it was one of the best graduation projects at Computer Engineering Department at Yarmouk University in the academic year of 2013/2014.