FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FPGA-based kNN Accelerators via High-Level Synthesis

André Silva

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João M. P. Cardoso

October 31, 2023

© André Silva, 2023

Resumo

Nos últimos anos tem havido limitações no aumento do poder computacional de CPUs convencionais e novas abordagens são necessárias de modo a acelerar sistemas computacionais. Sistemas heterogéneos que combinam um CPU e uma FPGA são uma alternativa a ser explorada, permitindo a desenvolvedores tirar partido das características e qualidades de cada uma das tecnologias.

Nesta dissertação são exploradas várias implementações em FPGA do algoritmo kNN. Tirando partido de características de FPGAs como permitir computações em paralelo para executar o algoritmo kNN, é possível acelerar essas computações e desenvolver módulos de hardware especializados que implementam essas otimizações. Usando estes módulos em conjunto com um CPU é possível obter uma implementação mais rápida do algoritmo kNN em comparação com uma implementação apenas em CPU.

Foi usada síntese de alto nível (HLS) de modo a gerar descrições RTL das implementações em hardware. Para tal foi utilizada a ferramenta Vitis HLS, que sintetiza código C em descrições de hardware que podem ser utilizadas para configurar FPGAs. No entanto, o código C deve ser devidamente estruturado para síntese de alto nível de modo que o Vitis HLS proporcione uma solução que vá de encontro ao desejado.

As diferentes implementações propostas podem ser configuradas com parâmetros introduzidos pelo desenvolvedor, que permitem controlar a aceleração e os recursos utilizados, de modo a garantir uma adequação para diferentes requisitos e recursos disponíveis em diferentes FPGAs. Para além das diferentes versões de código obtidas, foram também utilizados pragmas de modo a controlar os recursos utilizados.

Por fim, as várias implementações do algoritmo kNN foram testadas e avaliadas usando o software Vitis HLS que através da síntese de código C permite obter valores para a latência e recursos usados de uma implementação hardware tendo como alvo uma FPGA, sendo neste trabalho escolhida a FPGA ZYNQ XC7Z020-1CLG400C. Os resultados permitiram aferir as acelerações obtidas e os recursos usados para cada uma das implementações, sendo a maior aceleração registada entre versões em FPGA de 44,49 vezes, e quando comparadas implementações em CPU e FPGA da mesma versão a maior aceleração foi de 352,59 vezes.

Palavras chave: FPGA, High Level Synthesis, Heterogeneous systems, kNN

Abstract

In the last years there have been limitations on the increase of computational power of conventional CPUs, and new approaches are needed to accelerate computational systems. Heterogeneous systems that combine combining both CPUs and FPGAs on the same board are an alternative to be explored, allowing developers to take advantage of the characteristics and qualities of both technologies.

In this dissertation are explored various FPGA implementations of the kNN algorithm using high level synthesis. Taking advantage of the characteristics of FPGAs such as allowing parallel computations to perform the kNN algorithm, it is possible to accelerate the calculations needed to perform the kNN algorithm and develop specialized hardware cores that implement those optimizations. Using this cores in conjunction with a CPU provides a faster implementation of the kNN algorithm when compared to a CPU implementation.

High level synthesis was used to generate the RTL description of the hardware implementation. The framework used is the Vitis HLS, that synthesizes C code in hardware descriptions that can be used to configure FPGAs. However, C code must be properly structured for synthesis for Vitis HLS to deliver a solution that matches the proposed one.

The different proposed implementations can be configured with parameters chosen by the developer, that allow to control the speedup and resources used, to guarantee that the solution is suitable to different requirements and available resources in different FPGAs. Beyond the different code implementations pragmas were used to control the resources used.

Finally, the implementations of the kNN algorithm were tested and evaluated using Vitis HLS, that from synthesis of C code it was possible to obtain values for latency and resource usage of a hardware implementation targeting a FPGA, being the one chosen in this work the FPGA ZYNQ XC7Z020-1CLG400C. The results allowed to assess the obtained speedups and the resources used of each implementation, with the biggest acceleration registered between FPGA versions of 44.49 times, and comparing CPU and FPGA implementations of the same version the biggest acceleration was of 352.59 times.

Keywords: FPGA, High Level Synthesis, Heterogeneous systems, kNN

Acknowledgments

Firstly I would like to thank my supervisor João Cardoso for his guidance in the development of this dissertation. His guidance was fundamental in the conduction of this dissertation, and his advises helped to see the path to a solution clearer than I could myself.

I would also like to thank all the colleagues at FEUP that helped my journey in this university. With them being on the same ship as I was, it was possible to share the feeling that I was not into this alone, sharing the same struggles, but also motivating me.

Lastly and not the least I would also like to thank my family and friends for support during this journey. Their support made the hardest times more manageable and I cherish the moments that we spent together. Without the good times spent with them all of this journey would not have been possible.

André Silva

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 FPGA . . . 2
1.3 High-Level Synthesis . . . 3
1.4 kNN algorithm . . . 3
1.5 Problem and objectives . . . 4
1.6 Document structure . . . 4

2 Background **5**
2.1 kNN algorithm . . . 5
2.2 FPGA . . . 6
2.2.1 Vitis HLS . . . 6

3 Related work **9**
3.1 "An Efficient K-NN Algorithm Implemented on FPGA Based Heterogeneous Computing System Using OpenCL" . . . 9
3.2 "kNN-STUFF: kNN STreaming Unit for Fpgas" . . . 10
3.3 "A Memory-Access-Efficient Adaptive Implementation of kNN on FPGA through HLS" . . . 12
3.3.1 PCA principals . . . 12
3.3.2 PCA implementation . . . 13
3.3.3 FPGA implementation . . . 14
3.3.4 Results . . . 15
3.4 Architectural Considerations for FPGA Acceleration of Machine Learning Applications in MapReduce . . . 16
3.5 New Benchmark Suite . . . 18
3.5.1 New Benchmark Suite kNN code structure . . . 18
3.5.2 New Benchmark Suite data sets structure . . . 20
3.5.3 New Benchmark Suite code optimizations . . . 21
3.6 Summary . . . 22

4 kNN partitions **23**
4.1 Initial code and proposed changes . . . 23
4.2 Initial code changes . . . 26
4.2.1 Synthesis results . . . 28
4.3 kNN partitions versions . . . 31
4.4 Automation script and file organization . . . 33
4.5 Training dataset partition automatization . . . 35

4.6 Function calls and variables automatization 35
4.7 Local version C code 39
4.8 External and multi stage versions extra files 40
4.9 External version C code 41
4.10 Multi stage version 42
4.11 Multi stage version automation 45
4.12 Summary 47

5 Experimental results **49**
5.1 Experimental setup 49
5.2 FPGA results 49
5.3 CPU vs FPGA results 57

6 Conclusion **59**
6.1 Concluding remarks 59
6.2 Future work 60

References **61**

A kNN Added functions **63**
A.1 Initial version added functions 63
A.2 Multistage version added functions 65
A.3 Initial kNN code with 2 partitions 66
A.4 Initial kNN code with 4 partitions 69
A.5 Python script for the local version 74
A.6 Python script for the external version 82
A.7 Python script for the multi stage version 90

List of Figures

1.1 Sequential representation of tasks in an FPGA[1] 2
1.2 Parallel representation of tasks in an FPGA[1] 3

2.1 Visual representation of kNN algorithm for different values of k. [2] 6
2.2 Performance metrics obtained in Vitis HLS after synthesizing C code. Source:[3] 7
2.3 Schedule view obtained in Vitis HLS after synthesizing C code. Source:[3] . . . 7

3.1 Visual representation of distance calculations matrix. Source:[4] 10
3.2 Implementation of the distance calculating algorithm. Source:[5] 11
3.3 Implementation of the sorting algorithm. Source:[5] 11
3.4 Speedup achieved by varying number of accelerators. Source:[5] 12
3.5 Two dimension points being projected in a main direction. Source:[6] 13
3.6 Graphical representation of the modules used in the hardware implementation for an approximate kNN. Source:[7] 14
3.7 comparison of the execution time in a CPU and FPGA solution, varying the number of bits to represent an int and the number of threads running. Source:[7] . . . 16
3.8 Energy consumption of a CPU solution and a FPGA solution. Source:[7] 16
3.9 Representation of a system composed of multiple CPU+FPGA systems to calculate kNN in a cloud. Source:[8] 17

4.1 Block diagram representation of the original kNN algorithm code 24
4.2 Block diagram representation of the kNN algorithm with 2 partitions 25
4.3 Local version block diagram representation 32
4.4 External version block diagram representation 32
4.5 Multi stage version hardware version representation 33
4.6 Array partition example for division without and with remainder 36

5.1 Local version latency and speedup per partition over benchmark 50
5.2 External version latency and speedup per partition over benchmark 50
5.3 Speedups obtained from both local and external versions over benchmark 51
5.4 DSPs used per partition for local and external versions 51
5.5 LUTs used per partition for local and external versions 52
5.6 FFs used per partition for local and external versions 52
5.7 Latency curves for the multistage version 54
5.8 Speedup curves for the multistage version over benchmark 54
5.9 Number of BRAMs used by each multistage solution 55
5.10 Number of DSPs used by each multistage solution 55
5.11 Number of LUTs used by each multistage solution 56
5.12 Number of FFs used by each multistage solution 56

5.13 Speedup of FPGA over CPU implementations of the local and external versions . 57
5.14 Speedup of FPGA over CPU implementations of the multistage version 58

List of Tables

3.1 Results of the implementations in CPU,GPU and FPGA. Source:[4] 10
3.2 Data sets used in tests. Source:[5] 12
3.3 Normalized execution times for solutions changing options such as the CPU, the FPGA and if the FPGA in on or off chip. Source:[8] 18
3.4 Power dissipation for solutions changing variables such as the CPU, the FPGA and if the FPGA in on or off chip. Source:[8] 18
3.5 Example of the train.dat for the wisdm dataset 21

4.1 Latency and speedup table for synthesized versions of the original kNN code and proposed solution 29
4.2 Resource table for synthesized versions of the original kNN code and the proposed solution 30
4.3 Latency and speedup table for all synthesized versions of the kNN algorithm . . . 30
4.4 Resource table for all synthesized versions of the kNN algorithm 30

5.1 Resources used per partition for local version 53
5.2 Resources used per partition for external version 53

Code Listings

4.1 kNN algorithm code for 2 partitions 27
4.2 Declaration of the training dataset arrays 36
4.3 Declaration of the partial distance arrays 37
4.4 Declaration of the partitions of the testing point 37
4.5 Initialization of the partial distance arrays 37
4.6 Loop that copies the testing point into multiple independent variables 38
4.7 Calls of the partial distance predict funtions 38
4.8 Calling of the UpdateBest functions 38
4.9 Calling of the Vote functions 39
4.10 Code for the kNN Predict All function for the local version 39
4.11 Declaration of the training arrays in the external and multistage versions 40
4.12 Calling of the kNN PredictAll function for the external and multistage versions . 41
4.13 Code for the external version 41
4.14 Code for the multistage version with 2 external partitions and 1 local partition . . 43
4.15 Code for the multistage version with 2 external partitions and 2 local partitions . 44
4.16 Code for the multistage function 45
4.17 Code for the multistage version with 1 external partition and 3 local partitions . . 46
4.18 Declaration and code of the kNN_PredictAll function 46

Acronyms

ASIC	Application Specific Integrated circuit
BRAM	Block RAM
CLB	Configurable Logic Blocks
CPU	Central Processing Unit
DSP	Digital Signal Processor
FF	Flip Flop
FPGA	Field Programing Gate Array
HDL	Hardware Description Language
HLS	High Level Synthesis
II	Initiation Interval
LUT	Lookup Table
PCA	Principal Component Analysis
RTL	Register Transfer Level
SoC	System on Chip
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction

1.1 Context

Since 1965, when Moore's Law was introduced, the computational power of CPUs has been increasing more and more throughout the years, until recent times. Recently due to technological limitations, the increase in computational power in CPUs is no longer guaranteed or expected [9] [10]. As long as the computational power of CPUs was expected to increase, little attention was given to Application Specific Integrated Circuit (ASIC) implementations of algorithms since it was expected that the computational power of CPUs would continue to increase, while an ASIC is expensive to design and is limited in its purpose since it can only do the specific task that it was designed for. However, in recent times since the increases in computational power of CPUs are no longer expected, more attention is being given to alternative implementations, that when compared to a generic CPU can achieve higher speeds of processing data for a specific task.

ASIC implementations were not commonly used since its design was significantly harder when comparing to a software implementation of the same algorithm that could be compiled to run on a CPU, and once the hardware is assembled to execute a certain algorithm, it could only execute that specific algorithm, while a CPU could execute a wide array of applications and thus being the preferred choice for developing new applications.

The CPU design allows it to execute a wide array of instructions, to which code of any language is compiled and thus can execute the required application in the CPU. However, one of the shortcomings of the CPU due to its designs it that a CPU has limited parallel execution capability. This can be a hindrance in algorithms where parallelization is present in the algorithm itself, that is, computations that can be run at the same time since there are no data dependencies between them, and as such, there is no conflict from them being made in parallel. However, as a CPU has a limited capacity to execute instructions in parallel, it cannot take full advantage of this natural parallelism present in different algorithms, and as such, slowing down computations that could be made faster if they were parallelized further.

1.2 FPGA

Field programmable gate array(FPGA) [11] is an integrated circuit that contains different hardware components that can be programmed in order to implement an hardware implementation of a task or algorithm. FPGA hardware components can be programmed, and as such, different hardware designs can be implemented. Due to this use of hardware components, it is possible to create designs which allows for computations to be made in parallel more efficiently. By configuring the existing hardware on the FPGA, it is possible to implement multiple hardware cores. By instantiating those hardware cores multiple times, and correctly inputting and outputting data to and from them, it is possible to achieve a design where computations are done in parallel at great extents.

This is extremely useful for implementing an algorithm that presents natural parallelism. For example, a function that takes an array and sums 1 to all elements in that array, is a function that has no data dependencies between all the operations that needs to execute. As such, theoretically all operations could be made at the same time, in parallel, in a single clock cycle. However, as the CPU architecture only allows for sequential instructions, each sum will be made in its own clock cycle, in a sequential manner. When compared to the theoretical one clock cycle this is a huge slowdown. However using FPGA implementations it is possible to approach the theoretical limit of doing all the operations in a single clock cycle, since it is possible to program multiple hardware cores that perform the summation for each element of the array at the same time, and as such achieving a much lower latency for this operation when compared to a CPU implementation, even though an FPGA implementation will not reach the theoretical limit of doing the summation in a single clock cycle due to having to read the values of the array and write its result. In Figures 1.1 and 1.2 the comparison between tasks being executed in a sequential or parallel manner is demonstrated. In Figure 1.1, all operations are done in sequence, regardless of if they have data dependencies between them. In Figure 1.2, as operations B and C have no data dependencies between them, they can be done in parallel, and as such accelerating the whole operation.

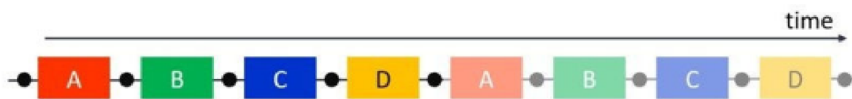

Figure 1.1: Sequential representation of tasks in an FPGA[1]

This example shows the potential that custom hardware implementations have in comparison to a CPU implementation, as CPU implementations have limited parallel execution while by creating custom hardware designs there is more room for creating modules with parallel computations execution capabilities. This is more relevant for heavy computational tasks, such as image processing, machine learning applications and more. Accelerating an operation that is made recurrently in an algorithm can provide greatly reduced latencies when compared to a CPU implementation

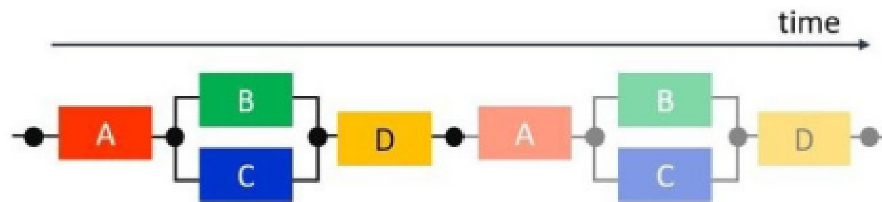

Figure 1.2: Parallel representation of tasks in an FPGA[1]

and as such for heavy computational tasks a custom hardware implementation can greatly reduce the time that takes for that task to complete.

1.3 High-Level Synthesis

FPGAs, as a device that allows configurable hardware implementations, are typically programmed using hardware description languages (HDLs) such as very high speed integrated circuit (VHSIC) hardware description language (VHDL) [12]. However, hardware description languages are comparably harder to implement a task or an application when compared to software languages such as C. As such, designing using HDLs require a deep expertise and hardware design knowledge, increasing costs and time consumed by design.

High-Level Synthesis (HLS)[13] tools allow for a compromise between HDLs and software languages. HLS allows designers to obtain RTL descriptions of their proposed designs from high-level languages such as C. This allows for much faster design of hardware descriptions of tasks and applications when compared to designing the hardware in a hardware description language. The HLS tool takes as input a C code, and based on that, provides a hardware implementation that should do the same task that was programmed in C.

However, special care is still needed when designing new hardware designs. There might be opportunities to implement certain design choices that are not possible in high-level languages, such as the parallelization of a task. As such, pragmas are used to guide the tool during the synthesis process. Pragmas can be used to inline functions, unroll loops, and partition arrays. Furthermore, there are other code changes that can impact the obtained hardware description, such as instantiating two functions with no dependencies between them for them to be able to be executed in parallel in the hardware implementation.

1.4 kNN algorithm

The kNN algorithm [2] is a machine learning algorithm that can be used in a wide range of applications. The kNN algorithm searches for the k nearest neighbours of a point to be classified and classifies the point as the most represented class in its k nearest neighbours. Due to the rising

popularity of machine learning applications for different purposes, it is an interesting algorithm to accelerate that can be used in various contexts. When analysing the calculations that are done by the algorithm, it shows great potential for a FPGA implementation, as the algorithm allows for most of its calculations to be done in parallel, which benefits from a FPGA implementation when compared to a CPU. Such implementation, if well designed and implemented, can greatly speedup the algorithm and reduce its energy consumption.

1.5 Problem and objectives

The goal of this dissertation is to propose fast kNN FPGA based implementations. This means, to create hardware descriptions that implements the kNN algorithm that when programmed in an FPGA achieves a speedup when compared to a CPU implementation of the kNN algorithm.

As such, throughout this dissertation we present the developed implementations of the kNN algorithm that aim to achieve the previously mentioned goal. This is done by synthesizing different C codes in Vitis HLS, exploring different optimizations and opportunities to execute computations in parallel. The different C codes are generated by a script, that will change the code according to some specifications, which allows for varying degrees of speedup and resources to be used by each implementation. Furthermore, the objective is to characterize the different obtained implementations in terms of achieved speedup and resources usage.

1.6 Document structure

This document is organized in 6 chapters. The current chapter (Chapter 1) introduced the problem at hand, the different technologies used and described the problem and the goals to achieve, Chapter 2 presents a compilation of the literary review of the fundamental concepts of the kNN algorithm, FPGAs structure and high-level synthesis. Chapter 3 presents previous related work that was done about related topics to those of this dissertation. In Chapter 4, it is presented all the proposed solutions and how they were achieved. Chapter 5 presents the results from the evaluations that were done post synthesis and after the proposed solutions were programmed in the FPGA+CPU SoC, and Chapter 6 presents the conclusions and future work.

Chapter 2

Background

This chapter presents a brief review of the fundamental concepts used in the dissertation.

2.1 kNN algorithm

The k-Nearest Neighbours algorithm is a widely used machine learning algorithm, that can be used to classify a given data point (instance) by which class it belongs, based on the class of its k nearest data points (instances)[2]. Suppose there is a database of data points in the form of (X,Y), of which it is know what class they belong, and it is chosen an arbitrary integer value for k. To classify a new data point, (x,y), it is needed to find the k points in the database that are nearest to the new data point. The most used distance is the Euclidean distance, but other ways of calculation of the distance between points are valid and may be more appropriate depending on the kind of data used. The algorithm calculates the distance of the new data point, (x,y), to every data point that is stored in the database. As the distances are calculated, they should be ranked, from the nearest to the furthest away to the new data point. After this is done, the classification of the new data point is equal to the more represented class in the nearest data points. Figure 2.1 shows how a new data point is classified depending on the number of nearest neighbours chosen (k). By setting k=1, only the nearest neighbour is considered, and as such the new data point is classified as class B, the class of the nearest neighbour. But by setting k=3, the most represented class in the three nearest neighbours is now class A, and the new data point is classified accordingly.

Due to the associated computations and parallelism involved, it is particularly suited for an implementation in FPGA when compared to a regular CPU, since the kNN algorithm shows great potential for parallelism, and as such can be greatly improved by parallelizing its operations. Due to the capability of instantiating multiple modules of calculating this distance in a FPGA, it is possible to compute multiple distances at the same time, which is a significant improvement when compared to a CPU. Other advantage of an implementation in an FPGA its due to its improved efficiency of ranking the distances computed when compared with a CPU. For example, if it is needed to rank a new distance and it is used an insertion sort algorithm to sort the distances calculated, the complexity of sorting a new value in a software implementation in O(N) while in hardware is

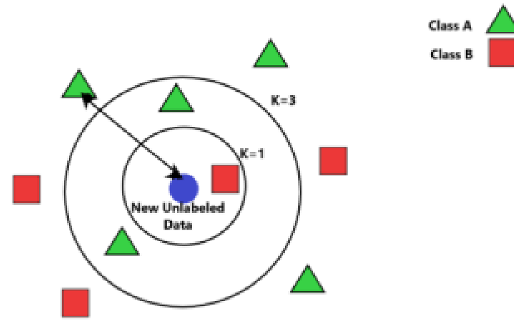

Figure 2.1: Visual representation of kNN algorithm for different values of k. [2]

O(1), which is a huge improvement. Even if the sorting algorithm isn't insertion sort, due to the possibility of parallelization of the computations needed, implementations of other sorting algorithms will be faster in hardware compared to a simple CPU. Lastly, FPGA implementations tend to have a lower energy consumption when compared to a CPU, which makes them suited for saving energy consumption. That happens because of the improvements in the time and complexity required to classify a new data point.

2.2 FPGA

To implement all the algorithms and approaches developed in this dissertation, it is important to know details about the FPGA chosen to be the target to implement the developed approaches. The chosen FPGA was the ZYNQ XC7Z020-1CLG400C, that is present in the board PYNQ-Z2 [14]. This FPGA has 220 DSPs, 53200 LUTs, 106400 FFs and 140 BRAMs.

As is with all FPGAs, this FPGA includes configurable logic blocks (CLBs), that can be programmed to implement a hardware implementation of the desired pieces of code, representing for example diverse functions needed to be implemented in hardware. With the correct programming of the FPGA obtained via synthesis of C code, it is possible to implement specialized hardware modules that present more parallelism when compared to a CPU by, for example, instantiating multiple hardwares that can do multiplications in parallel. While a CPU usually only has one ALU per core, and as such, can only do one operation per clock cycle in each core, meanwhile in an FPGA a designer is able to instantiate multiple operations blocks to be ran in parallel at the same clock cycle, taking advantage of the properties of the FPGA to better implement algorithms with high potential of parallelization.

2.2.1 Vitis HLS

Vitis HLS is [3] a high level synthesis tool, which means that is a program that can take high level code and transform it in a RTL design description of the said code, described in a hardware description language. In Vitis HLS the starting point is a C/C++ code and by running a synthesis of that code, it is possible to set a desired clock frequency and obtain an estimate to the number

of clock cycles that the hardware implementation generated from that code took from the start to the end of the function. Vitis HLS does diverse optimizations that can only be achieved in hardware rather than software by default, such as pipelining loops and in lining functions, so that the designer of the solution can focus on code optimizations that can lead to even further parallelism or having less reads and writes from and into an external memory. These optimizations, once synthesized, should also lead to a lower number of clock cycles that the function takes from its beginning to its end. An optimization that the designer of the system can control is the unrolling of loops, which occurs when the computations in a loop are independent from each other, so that it is possible to instantiate multiple loops running in parallel, leading to a faster solution. The trade-off is the use of greater number of hardware resources and higher power dissipation. Once the code is done it is possible to start the synthesis of that code, and after that a solution is obtained. In Vitis HLS it can be observed the performance estimates view and the schedule viewer, shown in Figures 2.2 and 2.3, respectively.

In performance estimates view it is possible to observe an estimate number of clock cycles that the synthesised function requires, as well as further information about each of the loops or cycles present in it.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.33	2.433	0.90

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2580	2580	2580	2580	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1034	1034	12	1	1	1024	yes
- Loop 2	1280	1280	5	-	-	256	no
- Loop 3	257	257	3	1	1	256	yes

Figure 2.2: Performance metrics obtained in Vitis HLS after synthesizing C code. Source:[3]

In Schedule view we can observe what operations took place in each cycle. This is important if it is intended a higher clock frequency, because it allows to identify which block is the limiting factor.

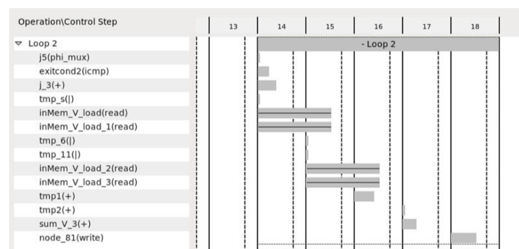

Figure 2.3: Schedule view obtained in Vitis HLS after synthesizing C code. Source:[3]

Chapter 3

Related work

This chapter presents some of the previous work addressing performance improvements of kNN implementations in the context of heterogeneous systems. This mainly consists of a few recent papers about implementations of the kNN algorithm in FPGAs.

3.1 "An Efficient K-NN Algorithm Implemented on FPGA Based Heterogeneous Computing System Using OpenCL"

Pu et al. [4] address an FPGA implementation of the kNN algorithm written in OpenCL, which is a framework for heterogeneous platforms, that is, systems that have more than one processing unit with different implementations, performances and energy consumptions. Using OpenCL, it is possible to program multiple computing kernels in order to parallelize computations. It works in both GPUs and FPGAs, which makes it cross-platform and mostly portable.

The focus is on two phases of the the kNN algorithm: The distance calculation and the sorting of the distances. As mentioned before, due to the independence of the calculations of the distances, they can be fully parallelized, which is what happens in the approach implemented in this paper. Multiple distance calculation kernels are created, that can calculate the distance to all points in a data set from multiple new data points, storing the results in a local array stored in the FPGA. Figure 3.1 shows a representation of the distance calculation between a new query point and a reference data set point. As the distances between different points in not dependent on any other points of the same data set, the distance calculations between different points can be made at the same time.

After computing all the distances, it begins the sorting stage where a partial bubble sort[15] is used. Since the k best results matter, the results are divided in bubbles of two distances, of which it is selected the nearest and inserted in another bubble, until there are only k distances, corresponding to the nearest points. After the k nearest points are known, the classification of the new point is equal to the most represented class in the k nearest points obtained.

To test the efficiency of this implementation, the execution time and the energy consumption of the implementation was compared in three different systems, those being a Core i7-3770K CPU

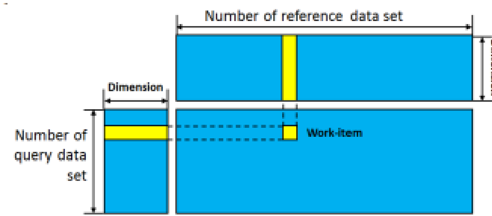

Figure 3.1: Visual representation of distance calculations matrix. Source:[4]

running at 3.5GHz, being used 4 i7 cores, in a AMD Radeon HD7950 GPU with 28 compute units and maximum frequency of 900MHz and in an Terasic DE4 with a Stratix IV 4SGX530 FPGA running at 400MHz, with a OpenCL compiler being used to do the comparison. The data set used in their tests was a quantum physics data set with 20480 points. The testing set were 20 points that needed to be classified with the results shown in Table 3.1.

Platform	CPU	GPU	FPGA
Feature size/nm	22	28	40
Runtime/ms	10211.05	24.85	69.12
Objects/s	1.96	804.96	289.34
Speedup	/	410	148
Power/w	130	200	24
Objects/J	0.015	4.024	12.056
EER	/	268	804

Table 3.1: Results of the implementations in CPU,GPU and FPGA. Source:[4]

The GPU and FPGA achieved a much lower run time when compared to the CPU, and the GPU was better in the run time where it achieved a speedup of 410 compared to the CPU, and the FPGA was the better choice if we need lower energy consumption, needing only 804 times less energy to compute the same number of data points when compared to the CPU.

3.2 "kNN-STUFF: kNN STreaming Unit for Fpgas"

Vieira et al. [5] present an implementation of the kNN algorithm in an FPGA obtained via RTL description. The main advantage of this implementation when compared to the previous [4] one is that it does not fully parallelize every operation possible, having a trade-off of achieving worse speedup but using less hardware resources. Another benefit of using this implementation is the fact that it implements the distance calculation and the sorting of the k nearest solutions in a single component, that can be used either as a standalone accelerator or used by creating multiple instances of it, obtaining a better speedup but using more hardware resources of the FPGA. The implementation of a single accelerator is shown in Figure 3.2 and Figure 3.3.

A single accelerator stores the new point to be classified and an array of all the values in the database. Each clock cycle, the distance between the new data point and a point already in the

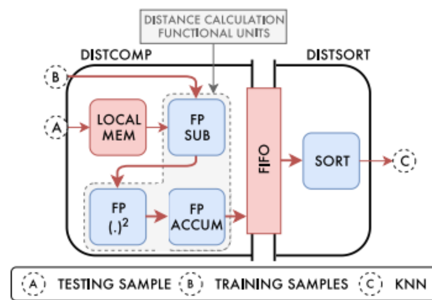

Figure 3.2: Implementation of the distance calculating algorithm. Source:[5]

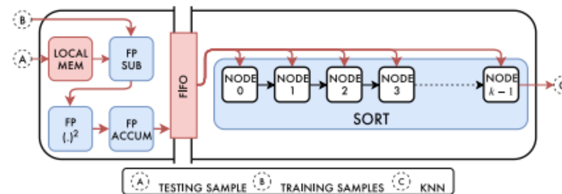

Figure 3.3: Implementation of the sorting algorithm. Source:[5]

database is calculated. In the following cycle, the new distance will be feeded to the insertion sort algorithm, and a new distance, between the new point and the following point in the database is calculated. Since the implementation of the insertion sort in algorithm has a complexity of O(1), it can feed a new distance every clock cycle, which increases the efficiency of this implementation when compared to a software one were the insertion sort could take up to O(N) and would not be able to process the distances calculated continuously. As said before, this accelerator could be instantiated multiple times, to make multiple calculations in parallel. The authors did not compare this implementation with other previous implementations, but compared the speedup and energy consumption of using a single accelerator versus multiple. One of these experiments compared the execution time when using a single accelerator to the execution time using 4 accelerators, with sets with different number of features and different number of elements. Multiple tests were done to compare the implementation in an FPGA and an implementation in software, using multiple sets with different training and testing sets and different number of features. One of those tests compared the speedup between the implementation in software and a hardware implementation with 4 accelerators instantiated, where the maximum speedup achieved was slightly above 11 times. The data sets used are presented in Table 3.2 and the corresponding speedup obtained by using the hardware implementation in Figure 3.4.

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

Table 3.2: Data sets used in tests. Source:[5]

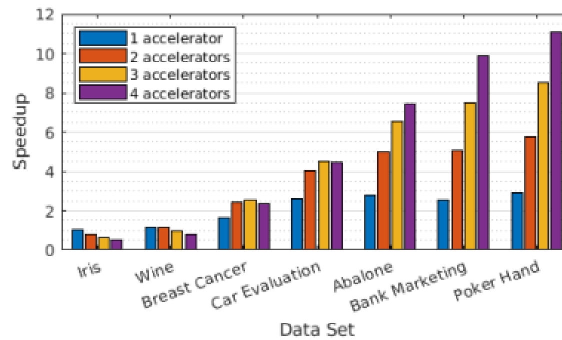

Figure 3.4: Speedup achieved by varying number of accelerators. Source:[5]

3.3 "A Memory-Access-Efficient Adaptive Implementation of kNN on FPGA through HLS"

Song et al. [7] presents a solution that differs from the ones previously discussed by introducing a statistical technique that may help obtaining results faster: the principal component analysis (PCA). The solutions so far did only calculate the distance between the point that we want to classify and all the data points in the data set to classify the desired point. With the addition of the PCA it can greatly speed up the distance calculation process, especially in cases where there is a high number of features (also referred as dimensions) in the data.

3.3.1 PCA principals

Before exploring the implementation, we briefly describe here how PCA works in the context of kNN [6]. PCA allows to obtain an approximate kNN result from ranking not the real distance between the point to classify and its nearest neighbours, but from ranking the distance between a orthogonal projection in the main components of the point and its neighbours. This is possible under the assumption that the main components are the ones that retains the most variance in the data set when isolated from the other components. The main benefit of this is that it reduces the dimensionality of the data set, reducing the number of subtractions and multiplications needed to obtain a distance between two points. This approach is well suited to kNN because it only cares about the classification between the distances and not the real distance between two points, and

when the orthogonal projection is done in the main component there is lost information about the real distance but maintaining in most cases the order of the distances. Figure 8 shows the PCA applied to kNN, where it is wanted to find the 2 nearest neighbours to the point to classify, q. to do that, all the data points, p, and q where projected in the main direction. After this, the distances are ordered in the main direction from the closest to the furthest. As can be seen, the final result for the 2 nearest neighbours in the main direction were the points p_0 and p_1. However, the result is not correct because point p_0 is not in fact a nearest neighbour to the point q, and p_6 is the correct result. This happens because the distance between points q and p_0 is bigger in the non-principal direction, in which case some wrong results can be obtained. However, this issue can mostly be dealt with, and the correction is explained in the next section where it is described the implementation of the PCA.

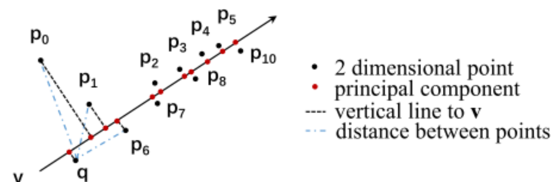

Figure 3.5: Two dimension points being projected in a main direction. Source:[6]

3.3.2 PCA implementation

to use PCA one needs to calculate the main components for the data set. to do this a threshold for the variance explained by the main components is defined. If it is set to 95 % and thus the main components chosen will explain 95 % of the variance in the data set.

After the main components are selected, it is needed to make an orthogonal projection of the data set and the point to classify in the main components, and store the values obtained. Once this is done, it is needed to set up 2 arrays where distances are stored: the main heap, where the k nearest neighbours will be stored ordered by the real distance between the point to classify and the data points, and the filter heap, where the orthogonal projection distances between the point to classify and our data points are stored. The filter heap should contain more points and distances than the real heap to ensure that it is obtained the real nearest neighbours at the end of this process. The points in the filter heap are candidates to being the nearest neighbours, and the main heap is where the final nearest neighbours are stored.

Starting the classification, the first step is to calculate the distance between a data point and the point to classify in the orthogonal projection in the main components. If this distance is bigger than the biggest distance present in the filter heap, this point is discarded as not being a candidate to being a nearest neighbour. However, if this distance is smaller than the biggest distance in the filter heap, this point is a candidate to being a nearest neighbour. If this is the case, it should be confirmed that this is the case when calculating the real distance. It is then proceeded to calculate

the real distance between the point to classify and the data point, and compare it to the biggest distance present in the main heap. If the distance if bigger, this point is not a real nearest neighbour and the point is not inserted in the main heap but kept in the filter heap. If this distance is smaller, this new value should be sorted in both the main and filter heap, and the previous value deleted from both.

The reason that are maintained more points in the filter heap than the real heap, and that are kept points that are not the nearest neighbour in the filter heap, is because there can be a situation like the one presented in Figure 3.5 where a point that is not a nearest neighbour has a smaller distance in the main components direction than one that is a real nearest neighbour. By keeping more values in the filter heap it allows more points to be tested and check if their real distance is in fact lower than the biggest real distance, while still eliminating a significant amount of data points that have bigger distances in both instances. In this way it can be assured that a better accuracy is achieved in the nearest neighbours obtained.

3.3.3 FPGA implementation

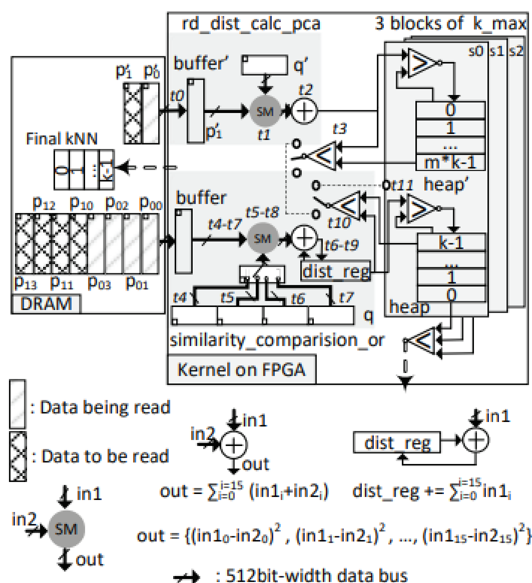

Figure 3.6: Graphical representation of the modules used in the hardware implementation for an approximate kNN. Source:[7]

Figure 3.6 represents the whole hardware system implemented in a FPGA and the modules that compose it. The module *k_max* implements the two heaps, *heap* and *heap'*. The *heap* represents the main heap and *heap'* represents the filter heap previously described. *rd_dist_calc_pca* module is the module used to calculate the distances of the orthogonal projections of the point to classify and the data points. *similarity_comparision_or* represents the module that calculates the real

distances between the point to classify and the data points. These modules interact in the same way as was previously mentioned and implement the PCA in an FPGA.

After this implementation, some further improvements can be made that are only possible in a hardware implementation. One of those improvements is calculating multiple distances of the orthogonal projections in parallel. As mentioned in [6], for 95% of the data points the real distance is not calculated, so by calculating the orthogonal projection distances in parallel it should be encountered a situation where it is needed to calculate more than one real distance for each time that it is calculated the orthogonal projection distances. If this was the case the calculation of multiple orthogonal projection distances in parallel could lead to a slowdown of our system, but as mentioned, those situations are rare.

Other optimization only possible since this solution is implemented in hardware is using low-precision data representations of the data set. The main goal is to increase the data read in each clock cycle. Since there can only be accessed 512 bits of data in each clock cycle, if it is reduced the number of bits that represents the data it can be achieved a greater data bandwidth. An improvement that is possible is using 16 bits to represent an int instead of 32 bits. What this allows is to access more data points in each clock cycles, which goes in end with the first optimization, since there are a larger number of distances to calculate for each clock cycle it becomes worthwhile to have multiple rd_dist_calc_pca working in parallel.

3.3.4 Results

In order to evaluate this solution, they used a data set with one million points with 960 dimensions. The baseline for comparison was the PCA implementation in software described in [6], and that solution run on a CPU server, that in total had 88 running threads and 128 GB of DDR4. The FPGA was set to use 16 main components in its calculations. The first test was comparing the execution time between the two solutions, while changing the number of bits that represents an int in the FPGA and changing the number of threads for the software solution. The results are shown in Figure 3.7.

Although it is not very noticeable, using a 32-bit representation for an int, the software solution needs to have 4 running threads to reach the same execution time that was obtained in the FPGA solution. When the number of bits to represent an int decreased, a much larger number of threads were needed to catch up to the execution time obtained in the FPGA. With this we can conclude that the FPGA is adequate and can run faster than a software implementation in a larger number of situations.

With respect to energy consumption, the FPGA did use 324 times less energy than the PCA software solution, making it much more energy efficient. The results are shown in Figure 3.8.

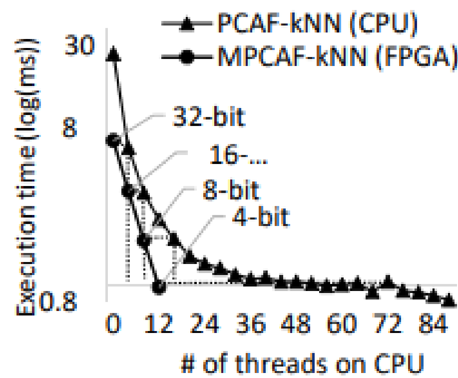

Figure 3.7: comparison of the execution time in a CPU and FPGA solution, varying the number of bits to represent an int and the number of threads running. Source:[7]

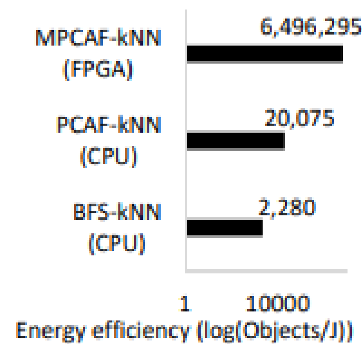

Figure 3.8: Energy consumption of a CPU solution and a FPGA solution. Source:[7]

3.4 Architectural Considerations for FPGA Acceleration of Machine Learning Applications in MapReduce

Neshatpour et al. [8] approaches the kNN implementation from a cloud based point of view. This is, it applies the kNN to a data set that is present not locally, but on a cloud, from which it is needed to actively exchange information to obtain the desired results for the kNN. This is important due to the fact that when it is dealt with huge data sets, they cannot be stored locally but only on cloud servers, making this approach an important one to study and know its implications.

To make the interactions with the cloud server, the MapReduce programming model [16] was used to exchange data with the cloud server, and the Hadoop framework [17] was used to implement that. to compare the different results, it was chosen different CPU and FPGA models in order to test its impact on the execution time and energy consumption of the solutions. The experimental setup used 2 main processors, A Xeon and a Atom core. The Xeon core is a high-performance core and Atom is a low energy consumption type of core. To complement the CPU where the software will be run, it was also chosen 2 types of FPGA, and Artix-7 being the high

end one and a Virtex-6 being the low end one. The final variable was if the FPGA was on or off chip. The main difference between these two approaches is the type of communication between the CPU and the FPGA. If the FPGA is on chip, the communication can be made with an AXI communication, that offers a bigger bit rate than the off chip alternative, a PCI-Express Gen3 communication. This is important because due to technology constraints, it is not always cost efficient to integrate an FPGA with a CPU in a single device, thus having off chip as a solution.

A model of the overall system can be seen in Figure 3.9. As can be seen, there is multiple CPU+FPGA working in parallel in the same data set that is stored in a cloud that contribute all to the same solution.

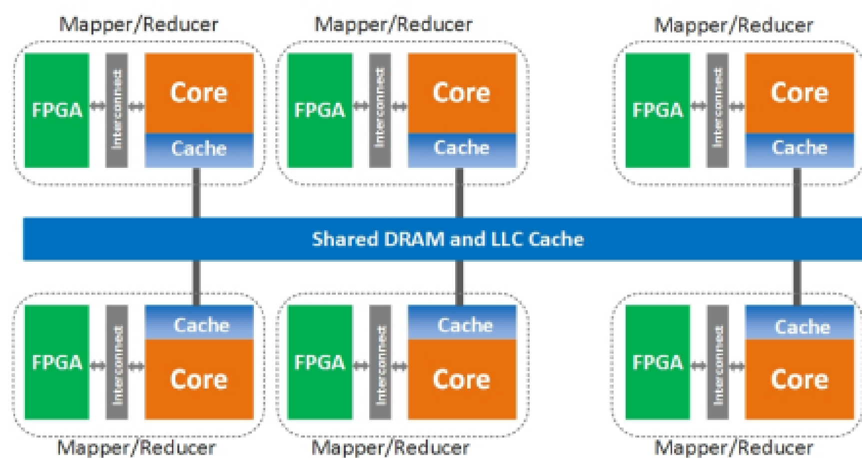

Figure 3.9: Representation of a system composed of multiple CPU+FPGA systems to calculate kNN in a cloud. Source:[8]

The results of experiments changing the above-mentioned options can be seen in Tables 3.3 and 3.4. As can be seen, using a FPGA implementation they achieved speedups and energy gains when compared to only processors implementations.

Also, it should be mentioned that this paper did not focus in the kNN implementation but in showing that using a CPU+FPGA approach is beneficial when compared to only a CPU implementation. The main conclusion of this paper is that a CPU and FPGA implementation is well suited to deal with databases stored in clouds, but we can further make improvements on the kNN implementation used in this paper, since that was not the focus and the implementation was a direct one, without optimizations. Using for example the approach with the approximate kNN refereed in this paper [7] is one way to improve the solution and making it even more suited for cloud applications of the kNN.

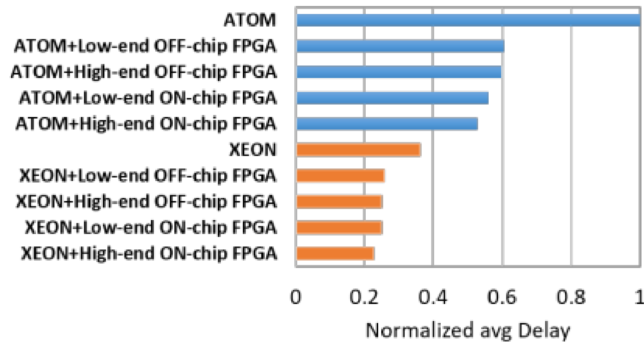

Table 3.3: Normalized execution times for solutions changing options such as the CPU, the FPGA and if the FPGA in on or off chip. Source:[8]

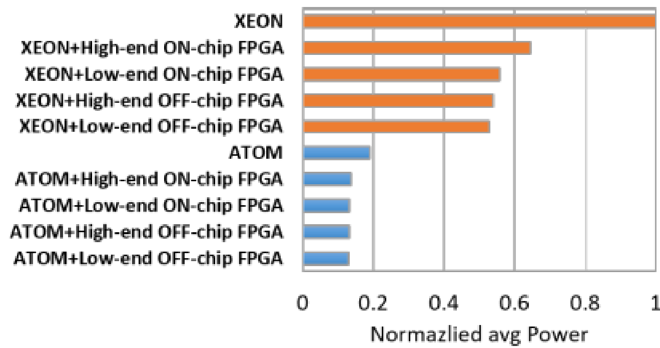

Table 3.4: Power dissipation for solutions changing variables such as the CPU, the FPGA and if the FPGA in on or off chip. Source:[8]

3.5 New Benchmark Suite

The new benchmark suite[18] is a GitHub repository created by Tiago Santos that compiles task heavy applications, that are suited to being accelerated by heterogeneous CPU and FPGA systems. As the tasks have a long execution time in a single CPU but can be accelerated by custom hardware cores that can execute some operations in parallel. This provides a good starting point to explore some possible optimizations regarding the implementation of the kNN algorithm in a hardware core.

3.5.1 New Benchmark Suite kNN code structure

One of the codes provided in the previously referred benchmark is a code example of kNN algorithm used to classify instances in data sets. This code is the base code that is explored in this dissertation to measure the accelerations that are obtained from our improvements to the code.

This code already covers some optimizations, such as changing the types of the variables depending on their size, such as for the data type of the features in our data set. If the N_CLASSES is less than 128 classes, the CLASS_TYPE will be defined as the type char, otherwise it will be defined as a short int, leading to the CLASS_TYPE to occupy less space, and thus accelerating the code.

Our kNN function takes as arguments the following arrays:training_X, an array that containing all the instances in the training data set, training_Y, and array that contains all the classes that the instances in our training data set belong, testing_X with all the features of the instances to classify, testing_Y where we store the predicted class for each instance in the testing data set, and a min and max array with the min and max values for all the features in the training data set, so that the feature values of the testing data set can be normalized to be in the range between 0 and 1 in relation of the values present in the training data set. The arrays are described below:

- DATA_TYPE training_X[N_TRAINING][N_FEATURES] - Contains all the features of all the training instances.
- CLASS_TYPE training_Y[N_TRAINING] - The real class that the training instance belongs.
- DATA_TYPE testing_X[N_TESTING][N_FEATURES] - Contains all the features of all the training instances.
- CLASS_TYPE testing_Y[N_TESTING] - Stores the predicted class for each training instance
- DATA_TYPE min[N_FEATURES] - The minimal value for each feature that was found in the training data set.
- DATA_TYPE max[N_FEATURES] - The maximum value for each feature that was found in the training data set.

It is also important to clarify what the variables that determine the array size mean. Those are variables set globally that help keep the same array size for a certain data set:

- int N_TRAINING - Number of the training instances
- int N_TESTING - Number of the testing instances
- int N_FEATURES - Number of features present in each instance
- int K - The number of nearest neighbours that is used to classify an instance

In addition to these arrays and parameters, there are some arrays declared in the function to allow the calculation of the kNN algorithm:

- double bestDistances[K] - Stores the k shortest distances between the point to classify and the points in the training dataset.

- int bestPointsIdx[K] - Store the index of the k best neighbours in the training data set.

The kNN function iterates for all instances in the testing_X, calculating the distance for each instance to all the data points present in the training_X array, storing the index of the k nearest neighbours to the instance and the distance from the instance to those k points, and updating the best neighbours if a shortest distance than any of the present in the k best is found after that until the distance is calculated between the instance and all points in the training data set. After that, it is checked in the training_Y array the class that the k nearest points belong, and classifies the instance as the most present class in the k nearest neighbours, storing the most present class in the closest neighbours, that is, the prediction, in the array testing_Y, where the predictions are stored.

This is done using some sub function that is important to know to understand how the code works. The sub-functions are as follows:

- void kNN_MinMaxNormalize (min, max, queryDatapoint) - Normalizes all the features of the queryDatapoint in relation to the the max and min values for each feature in the training data set so that the values are stored in a value between 0 and 1.
- void kNN_InitBest (bestDistances, bestPointsIdx) - Initializes the arrays that store the closest k neighbours distances to the point to classify and its index in the training dataset.
- double kNN_UpdateBestCaching (queryDistance, queryIdx ,bestDistances ,bestPointsIdx) - Checks if the most recent calculated distance between the point to classify and one point in the training data set is shorter than the ones previously calculated, and if yes, update the bestDistances and bestPointsIdx arrays.
- CLASS_TYPE kNN_VoteBetweenBest (bestPointsIdx, training_Y) - Calculates the most present class in the best k neighbours and classifies the point into that class.

Algorithm 1 kNN algorithm

```
1: for All_testing_instances do
2:     kNN_MinMaxNormalize
3:     kNN_InitBest
4:     for All_training_instances do
5:         for All_features do
6:             Calculate_distance_between_featuress
7:             Acumulate_distances_features
8:         kNN_UpdateBestCaching
9:     kNN_VoteBetweenBest
```

3.5.2 New Benchmark Suite data sets structure

This code uses two data sets to measure the improvements: the WISDM dataset [19] and a generated data set. In the main function, the code starts by analysing the data set files. Those are

files, train.dat and testing.dat, are text files, which can be read in the code in the main function. They are organized in the following fashion: each line represents an instance, with all its features and in the case of the training set the class that the point belongs. As such, we have N_FEATURES for each point, N_CLASSES representing the total number of the classes and NUM_TRAINING_SAMPLES represents the number of training points and NUM_TESTING_SAMPLES the number of testing samples.

Due to this display of the information, each line of the training and testing data set is read in the main function as a point where its stored all its features and the class that it belongs. This allows the computation of the kNN algorithm and after it's calculation it is possible to compare the predicted class to the real one, and it is possible to obtain a percentage of correct predictions.

Table 3.5 shows a representation of the data set files, how its structured and what it represents.

		Feature							Class
		0	1	2	3	4	...	N_FEATURES	
Instance	0	0.04	0.09	0.14	0.12	0.11	...	11.96	0
	1	0.12	0.12	0.06	0.07	0.11	...	12.05	0
	2	0.14	0.09	0.11	0.09	0.09	...	11.99	0
	3	0.06	0.1	0.09	0.09	0.11	...	10.69	2
	4	0.12	0.11	0.1	0.08	0.1	...	10.8	2
	5	0.09	0.09	0.1	0.12	0.08	...	8.63	2
	6	0.12	0.12	0.12	0.13	0.15	...	9.87	5
	7	0.1	0.1	0.1	0.1	0.11	...	9.91	5
⋮					⋮				⋮
	N_TRAINING	0.09	0.09	0.08	0.11	0.1	...	10.65	2

Table 3.5: Example of the train.dat for the wisdm dataset

However, the point struct where are stored both the value of all the features for a point and its class is not very efficient. That is because it is required to either access the feature values to calculate the distance to the point to classify, or to access its class. As such, in the main function, before calling the kNN_PredictAll the points structure for both the training and testing data set are separated in the X and Y arrays, where the X arrays store the values of the features and the Y array stores the class, as was explained previously.

3.5.3 New Benchmark Suite code optimizations

Although the base code of the New Benchmark Suite does not have optimizations regarding pragmas, there is still some optimizations implemented in the code regarding the variables type for the CLASS_TYPE and DATA_TYPE, varying the types of this variables to suit the data set better or if the user wants a more accurate kNN algorithm.

The CLASS_TYPE can be of two types: char or short. The difference between the two types is that a char is stored in one byte, while int is stored in 2 bytes. This way, if there is a number of classes in the data set that is low,the classes of the data set and the predicted ones can be stored in

a variable of the type char, and if there is a large number of classes the classes need to be stored in a variable of the type int so that the larger number of classes can be stored. The decision is made around the number of classes, and CLASS_TYPE will be of the type char if there are less than 129 classes (127 classes and unknown class), and of the type short if there are more.

The DATA_TYPE also varies, but does not depend on the training data set, but rather if the user wants a more accurate kNN in exchange for using more memory. If a lower accuracy is acceptable, the DATA_TYPE will be of the type float, that occupies 4 bytes in memory. However, if a higher accuracy is required for the application, the DATA_TYPE can be of the type double, leading to more accurate results.

3.6 Summary

In the recent works explored in this chapter, there were multiple approaches to the kNN algorithm. However, none of these approaches obtained the RTL description of the hardware to implement in the FPGA via HLS.

By using HLS, it is possible to explore more approaches to the implementation of the kNN algorithm when compared to writing the said RTL description from scratch since code to be synthesized can be written in C.

Another advantage of using HLS is that there is a more direct comparison when comparing CPU implementations of the kNN algorithm and the FPGA version obtained from synthesizing the code that implements the CPU version, allowing for a more accurate calculation of the speedup obtained using a FPGA of different implementations of the kNN algorithm.

Chapter 4

kNN partitions

As mentioned by multiple authors (see, eg, [4] [5]), a way of accelerating applications using an FPGA is to parallelize the computations. As such, the first approach to a more efficient implementation of the kNN algorithm is to consider multiple cores that are able to compute the kNN algorithm for a given training and testing data set in parallel, which should yield a speedup when compared to a simple implementation that runs the computations sequentially.

The name to this approach is kNN partitions, which is described in this chapter. The approach divides the training data set in partitions so that it is possible to compute the distance from the instance to classify to multiple points in the training set at the same time, reducing the time needed to calculate the distances needed and to find the k nearest neighbours to the instance to classify.

4.1 Initial code and proposed changes

The starting code for this approach is the NewBenchmarkCode [18] previously presented in chapter 3. The goal of this approach is to divide the training data set, and for each instance to classify, calculate the distance to multiple points present in the training data set in parallel. To do so, the code used needs to suffer some changes in order to accommodate the fact that multiple calculations need to be done in parallel. The changes in the code to parallelize the computations are:

- Divide the training data set: the data set should be divided in a specified number of parts, that each should have the same size and store a subset of the training set
- Calculate the nearest k neighbours in each partition: In each partition, find the best k neighbours of the instance to classify
- Find the best neighbours between the best points of all partitions: After the best k points for each partition are found, find the best k points among all partitions
- Classify accordingly to the best k points: Classify the instance to the most present class in the best k points in all partitions

In the original implementation of the code, there is only a single training data set, for which the distance of the instance to classify is calculated to all the points present in the training data set, and then it is found the nearest k neighbours to the instance to classify and finally the instance is classifies as the most represented class in nearest k neighbours. A block diagram of the original code is depicted in Figure 4.1. The kNN function takes as arguments the instances to classify, and the instances of the training data set and corresponding classes. The instances to classify are normalized, and for each instance to classify, its distance to all points in the training data set is calculated, from where the best k neighbours are found and then it is voted the most represented class in the best neighbours.

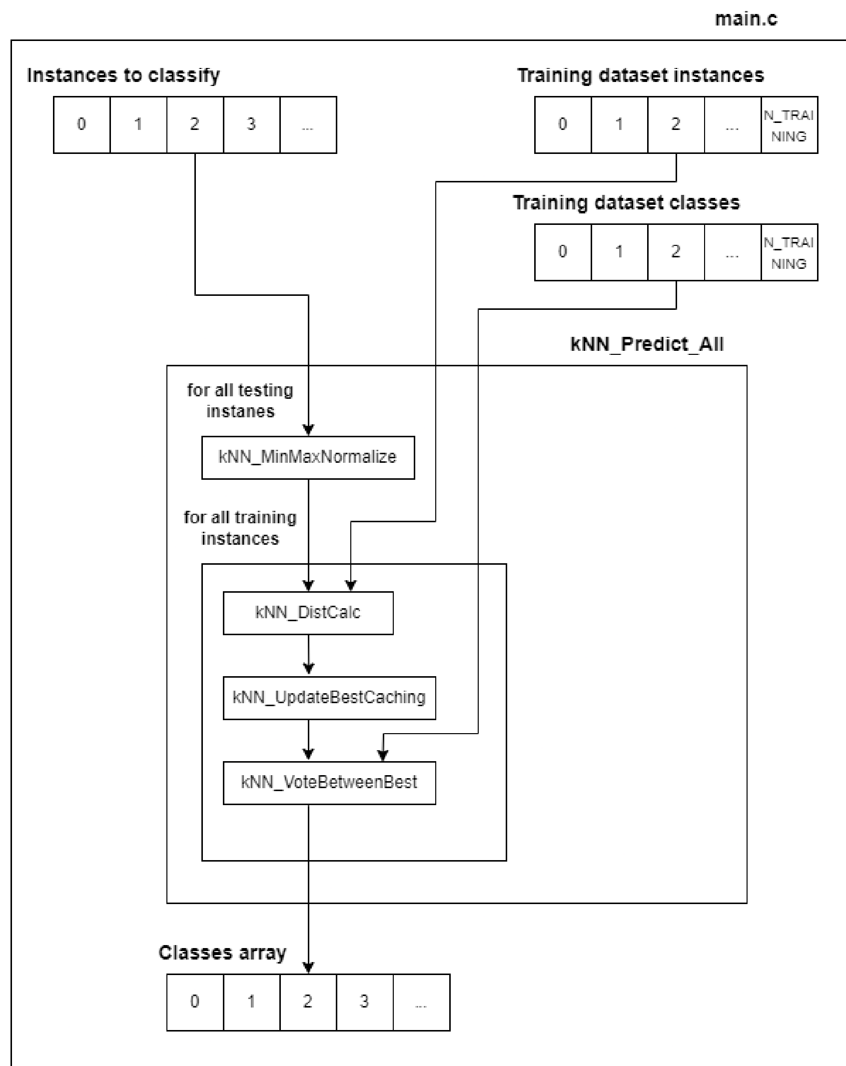

Figure 4.1: Block diagram representation of the original kNN algorithm code

The proposed solution is to first find the best k neighbours for a number of partitions of the training data set. As such, as each partition of the training data set is independent from each other, it is possible to calculate the distance between the instance to classify and points in multiple par-

titions of the training data set in parallel. The parallelism in computing the distances is important in achieving a speedup, since it is computational expensive to calculate all the distances, which incurs in high a latency for classifying an instance. By making those operations in parallel it is possible to achieve a speedup in comparison to an implementation that only allows the computations to be made sequentially. A block diagram representation of the proposed solution is present in Figure 4.2, where the training data set is copied into local variables and divided into two arrays, each representing a partition of the training data set. There are also two instances that calculate distance between the instance to classify and the points in each partitions of the training data set, where the best k neighbours for each partitions are found.

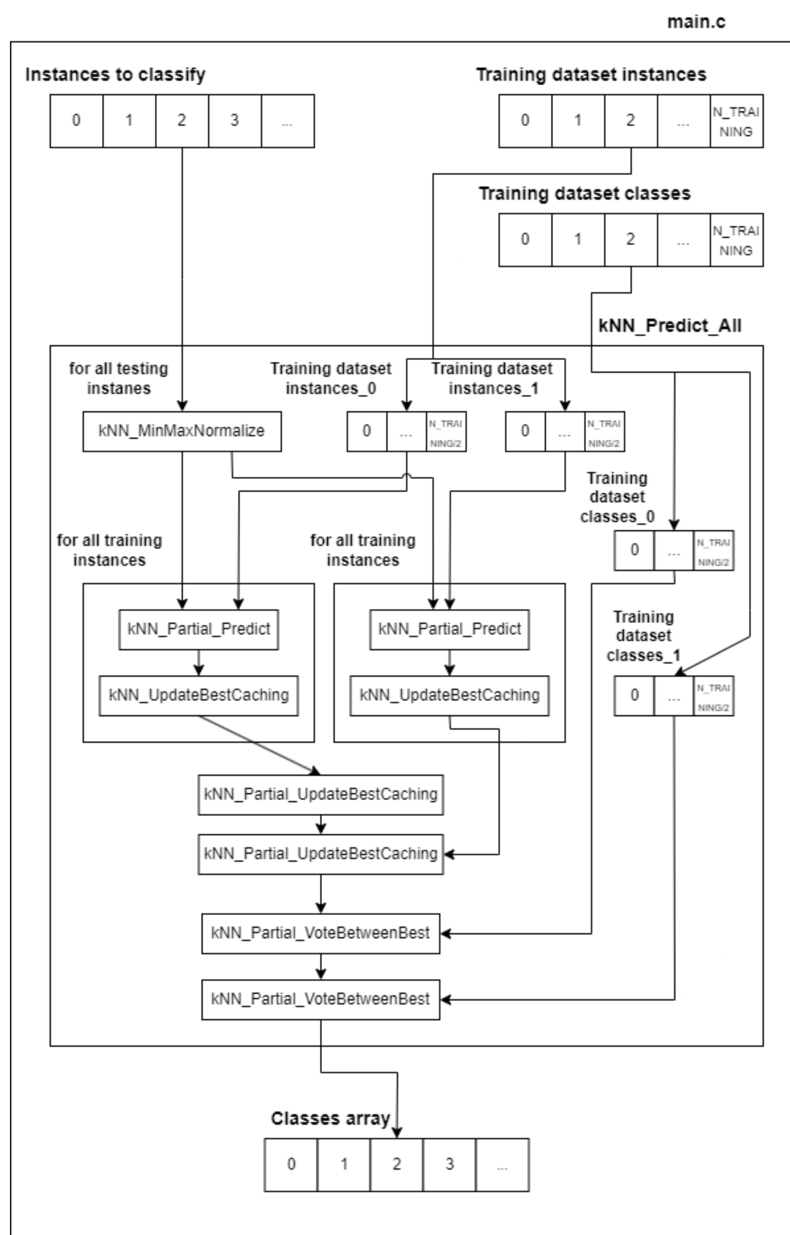

Figure 4.2: Block diagram representation of the kNN algorithm with 2 partitions

To calculate the best k neighbours in a data set while having multiple partitions it is needed to first find the best k neighbours in a single partition, and then rank those points again to find the best k neighbours for all the data set from the best k neighbours from each partition.

In the original code, the best distances are kept in the array bestDistances and the index of the point correspondent to that distance in the array bestPointsIdx, with the distance and the index being stored in the same position in the two arrays.

However, as goal for this implementation is for the training set to be partitioned, the id of a point will be referent to its position in its own partition of the training data set.

As such, the array bestPointsIdx will store the id of the k best points in a single partition, and the array bestPointsArrayIdx will store the id of the partition that the corresponding point belongs. The distance of the point to the instance to classify, the index in its partition and the id of its partition are kept in the same position across all the aforementioned arrays, in order to facilitate the updating of those arrays, since when it is needed to update the values in those arrays it is the same position in all of them that will be updated.

The array bestPointsArrayIdx is important to the voting phase, because while the array bestPointsIdx stores the k best neighbours for all the data set, the best neighbours can belong to different partitions. As such, it is important to know to what partition each point belongs, and only check its class in the partition of the data set that the point belongs. As the voting function can only take partition of the training data set, it is needed that the function is called for all partitions, but only update the voting process if a best point belongs to the partition that is input in the function.

The class of the best k points are kept in a structure named histogram. This histogram has N_CLASSES int variables, where it is stored the number of best points that belong to a class for the training data set. As the voting function is called multiple times, this histogram is passed to the function and only updated if there is a best point in the partition that is being input to the function, and after all partitions are input in the function with the same histogram, it is possible to check in the histogram which is the most represented class, which is the instance that the instance to classify will be classified as.

4.2 Initial code changes

To implement the changes described earlier to the original kNN code, some new arrays were added and some functioned slightly changed in order to accommodate the differences in the computations needed in order to execute the kNN algorithm in partitions. The new arrays created for the proposed implementation are the following:

- double bestDistances[K] - Stores the K best neighbours from all partitions.
- int bestPointsIdx[K] - Stores the index of the K best neighbours in its own partition.
- int bestPointsArrayIdx[K] - Stores the index of the partition of which the best K neighbours belong

These arrays represent the best k neighbours for the whole data set, but for each partition there are two arrays that represent that partition only:

- int bestPointsIdx_0[K] - Stores the index of the best k neighbours in the partition 0
- double bestDistances_0[K]; - Stores the distance to the instance to classify of the best k neighbours in the partition 0

Also, some of the functions were changed slightly in order to accommodate the fact that the training data set is now divided in partitions. The changed functions are:

- void kNN_Partial_Predict (training_X, queryDatapoint, bestPointsIdx, bestDistances, n_part): Predicts the k best neighbours of an instance in a given training data set partition. Includes an int argument that specifies the size of the partition of the training set that is used to calculate the distances,in order for the function to know how many points it as to loop in order to calculate the distances for all the points in a partition.
- double kNN_Partial_UpdateBestCaching (queryDistance, queryIdx, bestDistances, bestPointsIdx, bestPointsArrayIdx, arrayIdx): Takes as arguments both the arrays that store the information about the best points in a single partition and the whole training data set. The goal is to find the best points between all the partitions, and if a new best point is found, updating the arrays for the best points of the whole training data set.
- CLASS_TYPE kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y, histogram, bestPointsArrayIdx, arrayId): Checks if a best point is present in the provided partition, and if it does, checks the classification of the point and adds to the histogram an occurrence of its class. Returns the most popular class in the histogram.

A snippet of the C code written to implement the proposed solution is presented in Code Listings 4.1.

```
1
2
3  // Create and initialize arrays to store the best points(code not shown)
4  // Store the testing dataset in multiple local arrays(code not shown)
5  // Loop across all testing instances
6      for (int i = 0; i < N_TESTING; i++)
7      {
8          // Normalize the testing instance
9          kNN_MinMaxNormalize(min, max, testing_X[i]);
10         //Calculate the best k points from the partitioned training datasets
11         kNN_Partial_Predict(training_X_first, testing_X, min_first, max_first,
12         bestPointsIdx_first,bestDistances_first,2);
13         kNN_Partial_Predict(training_X_second, testing_X, min_second,
14         max_second,bestPointsIdx_second,bestDistances_second,2);
15
16         //Find the best k points from all partioned datasets
```

```
17          for(int i=0;i<K;i++){
18            kNN_Partial_UpdateBestCaching(bestDistances_first[i],
19                bestPointsIdx_first[i],bestDistances, bestPointsIdx,
20                bestPointsArrayIdx,1);
21          }
22
23          for(int i=0;i<K;i++){
24            kNN_Partial_UpdateBestCaching(bestDistances_second[i],
25              bestPointsIdx_second[i],bestDistances, bestPointsIdx,
26              bestPointsArrayIdx,2);
27          }
28
29          // Find the most represented class in the best points and predict the
30          classification for the instance
31          kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y_first,
32          histogram,bestPointsArrayIdx,1);
33
34          testing_Y[i]=kNN_Partial_VoteBetweenBest(bestPointsIdx,
35          training_Y_second,histogram,bestPointsArrayIdx,2);
36
37          //Reset the histogram for the next instance
38          for(int i=0;i<N_CLASSES;i++){
39            histogram[i]=0;
40          }
41      }
42  }
```

Code Listing 4.1: kNN algorithm code for 2 partitions

In the previous code, both the training and testing data sets are copied to local variables. This is done to ensure that the computations can be made in parallel once the code is synthesised via high-level synthesis.

The copying of the training data set leads to the use of BRAMs to store the training data set in local variables of the kNN function and DSPs to copy the contents from the input variable of the kNN function to local variables, but greatly reduces accessing data times. Also, the instance to classify is copied to multiple arrays. This is done so that the arrays that are called by the kNN_Partial_Predict are not the same arrays, and as such, Vitis HLS synthesizes the code with the computations being made in parallel in the synthesized version.

4.2.1 Synthesis results

The previous code, where the training data set was divided in 2 partitions, was synthesized in Vitis HLS and was compared to a synthesized version of the original code without any changes. Both codes were synthesized with a target clock period of 15 nanoseconds, without applying any pragmas to the code, so that the results are comparable between the two solutions. The goal is to confirm if the newly proposed solution does indeed have a speedup when compared to the original kNN algorithm code.

After the high-level synthesis, the latency for each solution was registered, and from it was possible to calculate the speedup obtained from the proposed solution. The results are displayed in Figure 4.1, where a speedup of 7.91 was obtained for the new solution when compared to the original code solution.

	Original Code	2 Partitions
Latency (s)	1.560	0.197
Speedup	1	7.91

Table 4.1: Latency and speedup table for synthesized versions of the original kNN code and proposed solution

The results can be explained due to two factors: the reduced reading times required for the computations requiring consecutive data points in the training data set, as in the 2 partitions code the partitions of the training data set are stored locally, which allowed the computations both to be pipelined with less interval between the current training data set instance and the next, and the fact that the variables were stored in different local variables allowed the parallelization of some computations, namely it was possible to calculate the distance between the instance and all the points of the two partitions in parallel at the same time, greatly reducing the latency of the algorithm. The speedup obtained was more than 2, which could be expected since the number of distance calculations done in parallel doubled, but since also the training data ser is not stored locally, it lead to a further increase in the speedup due to reducing the reading times of instances in the training data set.

However, it is expected that with the parallelization there is an increase in the number of resources used by the synthesized version, since it needs to use more hardware components of the FPGA to calculate the distance between the instance and the two partitions of the training data set at the same time, one for each partition since the operations were parallelized. It is also expected that some resources were allocated to the copying of input variables of the kNN function to the local variables.

The comparison between the resources used in the synthesized version of the original code and the synthesized version of the code with the data set partitioned in two partitions can be seen in the Table 4.2. Just like the speedup, the number of resources used more than doubled. This can be explained since the reading time of the training data set is lower, there is potential for a lower initiation interval between calculating the distances to all points in the data set for consecutive instances to classify, but for this to be achieved, there is a need to accelerate and do more computations in parallel in order to use the new values as soon as possible in the loop, such as multiplications. As such, to achieve this lower initiation interval, more resources will be used.

Given the results for the solution where the training data set was partitioned in 2 partitions, and the training and testing data sets were stored in local variables, a new version with 4 partitions was coded. The new code for 4 partitions code is vastly identical to the code with 2 partitions, but in each step each function is now called 4 times instead of 2.

	Original Code	2 Partitions
BRAM	0	694
DSP	33	293
FF	13059	54300
LUT	11489	59905

Table 4.2: Resource table for synthesized versions of the original kNN code and the proposed solution

This new solution with the training data set divided in 4 partitions was synthesized in the same conditions that the previous solutions were synthesized. The results can be seen in Table 5.3.

	Original Code	2 Partitions	4 Partitions
Latency (s)	1.56	0.197	0.111
Speedup	1	7.91	14.05

Table 4.3: Latency and speedup table for all synthesized versions of the kNN algorithm

The speedup between the synthesized version of the original code and the new one increased again, this time to 14.05 when compared to the original solution. This results shows that the partition of the training data set and copy of the instance to classify to a local variable of the kNN function is a good approach to follow to achieve higher speedups. However, it is also noticeable that the increase in speedup was smaller when comparing to the speedup obtained from the original solution to the 2 partitions solutions: while the speedup increased 6.91 times from the first solution to the second, the speedup only increased 6.14 times from the 2 partitions solution to the 4 partitions solution. As such, it is expected that if more partitions are made, the gains in speedup will be diminishing when increasing the number of partitions. This can be explained due to the increase in the reading times when adding more parallel calculations, as while the distance calculations can be made in parallel, finding the best neighbours for all instances and finding the most represent class in the best points is done sequentially. As such, while increasing the number of partitions, it increases the reading times for the sequential part of the algorithm, which will decrease the speedup obtained between the original code and the 2 partitions version and the 2 partitions version and 4 partitions version.

This paves the way for new experiments, where it is tried to find the best number of partitions that lead to the best speedup possible.

The resources used by the 3 versions are shown in Table 4.4 .

	Original Code	2 Partitions	4 Partitions
BRAM	0	694	700
DSP	33	293	583
FF	13059	543	101775
LUT	11489	59905	112436

Table 4.4: Resource table for all synthesized versions of the kNN algorithm

When comparing the 2 partitions version with the 4 partitions version, is is possible to observe that the number of FFs, LUTs and DSPs nearly double, mirroring the increase in speedup when comparing the two versions. However, the number of BRAMS stayed almost constant, due to the fact that the training data set occupies the same space in memory, be it divided in two partitions or four.

4.3 kNN partitions versions

Given the previous results from partitioning the training data set, the approach to increase the number of partitions shows potential to achieve a higher speedup. As such, the new approach is to develop new codes that partition the training data set even further, in order to achieve a higher speedup.

However, this task is a time-consuming task, since all of the previous codes were "hand written" for each version. A better approach is to automate the code to be written, creating a script that only needs to know in how many partitions the training data set is to be divided, and then the said script automates the correspondent code to the required number of partitions. This can be done since the code was segmented in separate phases of the kNN calculations that now allow for new versions with any number of partitions to follow the same pattern, allowing the task to be automatized. As such, the goal of the new script is to automatize the code written.

Also, new versions of the code are created, all serving a purpose for data sets of varied sizes and FPGAs with varying number of resources: A local version, an external version, and a multi-step version. A brief description of the three versions is:

- Local version: The training data set is directly included into the kNN function
- External version: The training data set is included into the main function and is passed as an argument to the kNN function, without it being copied to local variables of the kNN function
- Multi stage version: The training data set is included into the main function and is passed as an argument to the kNN function, being copied to local variables of the kNN function

In the local version, the training data set is included in the kNN function itself. This is done including the partitioned data set into local variables of the function in compilation time, and as such, the training data set needs to be pre-processed, cannot be changed during run time and is stored in BRAMs. This approach is suited for a combination of data sets and FPGAs where the whole data set can be stored in the BRAMs of the FPGA. This is because in this version the data set is stored locally in the FPGA, and as such, greatly diminishing the read times of the training data set when compared to a version where the data set is passed as an input variable of the kNN function, and as such is stored outside of the FPGA which incurs in higher reading times and a higher interval between reads of the data set. However, this is only possible if the FPGA has the number of BRAMs to store the training data set, and as such an adequate training data set or FPGA

must be chosen in order for this version to be used. A block diagram of this version is present in Figure 4.3.

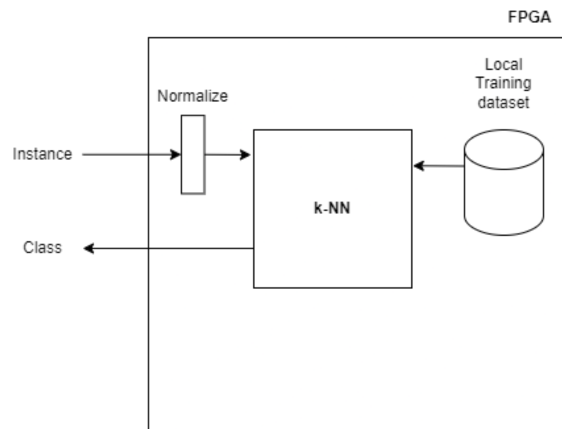

Figure 4.3: Local version block diagram representation

In the external version, the training data set is an input variable of the kNN function and is not stored in local variables. This means that the training data set might be changed, but the most importantly, it is stored in the memory of whole CPU+FPGA board and not in the BRAMs of the FPGA, which are able to store a bigger amount of data. This approach is indicated for when the required training data set cannot be stored in the BRAMs of the FPGA and as such must be stored in the memory of the board. This incurs in higher reading times and interval between consecutive reads of the training data set, but it is still able to calculate the kNN algorithm with multiple partitions. This version is shown in Figure 4.4.

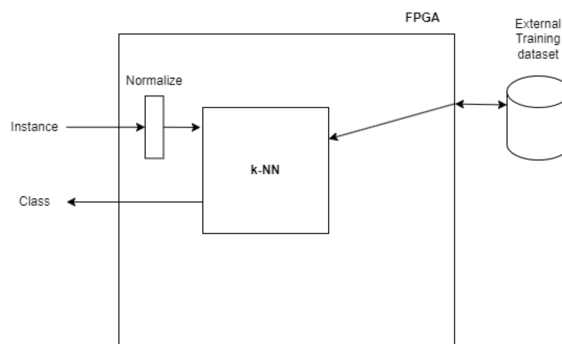

Figure 4.4: External version block diagram representation

Lastly, the multi-stage version is a version where the training data set cannot be completely loaded into the memory of the FPGA due to limitations one the number of BRAMs of FPGA, which occur for bigger data sets. A representation of this version is present in Figure 4.5. The data sets are an input variable of the kNN function, just as the external version. However, in this version, the training data set is copied to local variables, where that partition of the training data set will be stored, and the kNN algorithm calculated for that partition. After the kNN algorithm is

done for that partition, the next partition is copied to the local variables of the function, repeating the process until the kNN is done for all partitions.

The number of partitions should be depending on the training data set size and the number of BRAMS on the FPGA. The goal is that the local variables take as much of the BRAMs of the FPGA as possible, allowing that partition of the training data set to be locally stored while the kNN algorithm is being calculated for the partition, and once the kNN is done copy the next partition to the local variables.

Furthermore, the partitions can be further partitioned. In the local version, the local variables are partitioned allowing for calculations in parallel. In the multi stage version this can also be done, partitioning the partition of the training data set in partitions of the partition that is being worked. The partition of the partition that is stored locally does not affect the number of BRAMs that are needed to store the training data set but can increase the number of computations in parallel done at the same time and as such achieving a higher speedup. However, there will be increased latency due to the copying of the external data set to the local variables for which should be accounted. This version allows a greater degree of flexibility allowing for more control of the resources used by an implementation of the kNN algorithm.

For the multi stage version, external partitions represent the number of partitions of the data set, and local partition will represent in how many partitions is divided the local variables that store a partition of the data set.

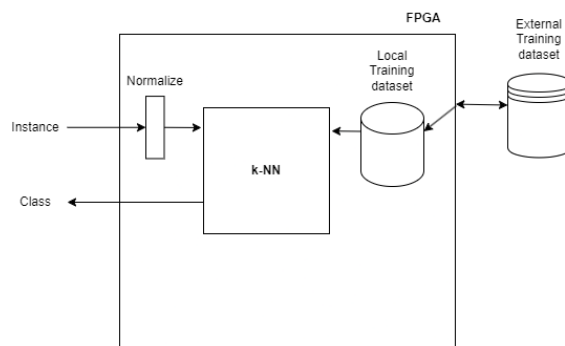

Figure 4.5: Multi stage version hardware version representation

4.4 Automation script and file organization

A script was developed to automate the process of generating code that implement the new versions of the kNN algorithm with varying number of partitions. The goals of the script are two: first, to divide the training data set into partitions before it is included in the kNN code, and second, to automate the function calls that are used to calculate the kNN algorithm in according to the number of partitions required. This process is done by the same script. Each version of the kNN algorithm will have its own script, but they share a lot of the code, namely, the division of the training data set into partitions will be the same for all versions, and also the majority of the files that implement

the function calls are the same for all versions. As such, an overview of the files shared between all versions will be presented and then the few changes between versions will be address later.

The script takes as arguments the number of partitions desired and can take as an argument the ID of the scenario being used. The usage of the script for the local version is the following:

```
1    python local_partition_script.py [Number of local partitions] [ID of the
     scenario(optional)]
```

The files created by the script are included in the code of the versions using includes, allowing for easy changes to the code by just running the script. By adjusting the number of partitions in the script, it is possible to obtain the code for a version with that number of partitions.

The training data sets are represented in files named training_X_id and training_Y_id. The id corresponds to the number of the partition, and it is created as much of those files as there are partitions.

The files that call the sub functions of the kNN algorithm are created once, but its content will vary according to the number of partitions, where more or less functions are called accordingly to the number of partitions.

The script and the files created by it are stored in a folder inside the folder that has the main.c and knn.c files. The structure of the new folder and the files created for the local version with 2 partitions can be found below:

```
local_partition
 ├─ local_partition_script.py
 ├─ training_X_0.dat
 ├─ training_X_1.dat
 ├─ training_Y_0.dat
 ├─ training_Y_1.dat
 ├─ array_dec.h
 ├─ partial_dist_dec.h
 ├─ local_arrays_dec.h
 ├─ partial_dist_init.h
 ├─ local_arrays_init.h
 ├─ partial_predict.h
 ├─ update_best.h
 └─ vote.h
```

The local version was chosen to demonstrate the folder where the script is stored, and the files created by it due to the fact that the local version is the version that has the few files. Since the training data set is directly included in the kNN function, it does not need to receive data set as an argument, regardless of the number of partitions. As such, the declaration and calling of the kNN function for the local version will always be the same, being input an array of instances to classify and the array that stores the attributed class to each instance. For the external and multi step versions, as the arrays are input in the kNN function, its declaration and calling vary with the

number of partitions which leads to more files being used. However, the specificities of the file structure for each version are addressed later.

4.5 Training dataset partition automatization

The first step to automate the new versions is to automate the process of dividing the training data set. The division of the training data set in the hand written versions was a task that was included in the synthesis of the whole kNN function, and with that, took resources that are not needed to calculate the kNN algorithm and added latency that was not due to the kNN algorithm itself, but the divising of the training data set in partitions. As such, it is important to divide the training data set beforehand, so that it can be used without having code to divide the training data set in the kNN code to be synthesized, being either directly included into local variables of the kNN function or passed as arguments in the external and multi-stage versions. This step is also needed to generalize the process so that it is possible to divide the training data set in any number of partitions that is desired, including situations where the division of the number of training points and the number of partitions has a remainder.

As such, it was included in the script code that divided the training data set into any number of partitions that are required. The code starts by reading the training_X.dat and training_Y.dat files, where the features and class for the points in the training data set are stored, each in each file, respectively.

While reading the files, it counts the number of lines in the files, and stores that value as the N_TRAINING, the number of training points in the data set.

After that it divides the N_TRAINING by the number of partitions. If the result has no remainder, the training points are divided into n_partitions of (N_TRAINING/n_partitions) size.

If the result of the division of the number of partitions by the number of partitions has remainder, the first partitions up to the number that is equal to the remainder will store training points, while the following partitions have its features set to the maximum value, in order for all the partitions to have the same size but so that the computations are not affected by the fact that the training data set could not be distributed equally by the partitions, as those newly created points added to the partitions have the maximum value possible and as such will never be best neighbours.

In the end the result of this division is stored in files with the name training_X_id.dat and training_Y_id.dat, where the id represents the number of the partition stored in that file. These files will later be included in all of the versions, and as such the code is useful for all of them.

4.6 Function calls and variables automatization

The next step to automatize the creation of code for varying number of partitions is to automate the process of creating the files mentioned before. For each file it is explained why it is needed and what it does, and some functions called in the file or variables created will be further explained.

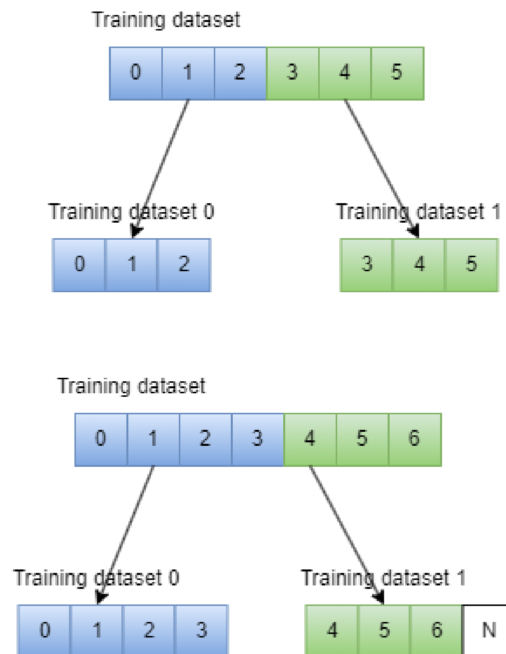

Figure 4.6: Array partition example for division without and with remainder

The contents of the file are also shown, with an example for a partition of 3 parts of the training data set being chosen.

The file array_dec.h includes the training data set files in the code. For the local version, this files are included in the kNN function, and as such, the training data set is stored locally, while for the external and multi step version it will be called in the main function. The size of the variables is adjusted given the number of partitions in order to increase memory usage efficiency.

This file will include the training_X,training_Y, min and max arrays and its contents are present in Code Listing 4.2.

```
1  static DATA_TYPE training_X_0[1446][N_FEATURES]={
2      #include "./localarrays/training_X_0.dat"
3  };
4  static CLASS_TYPE training_Y_0[1446]=
5      #include "./localarrays/training_Y_0.dat"
6  ;
7  static DATA_TYPE min[N_FEATURES]=
8      #include"min.dat"
9  ;
10 static DATA_TYPE max[N_FEATURES]=
11     #include"max.dat"
12 ;
13 static DATA_TYPE training_X_1[1446][N_FEATURES]={
14     #include "./localarrays/training_X_1.dat"
15 };
16 static CLASS_TYPE training_Y_1[1446]=
17     #include "./localarrays/training_Y_1.dat"
```

```
18 ;
19 static DATA_TYPE training_X_2[1446][N_FEATURES]={
20     #include "./localarrays/training_X_2.dat"
21 };
22 static CLASS_TYPE training_Y_2[1446]=
23     #include "./localarrays/training_Y_2.dat"
24 ;
```

Code Listing 4.2: Declaration of the training dataset arrays

The files partial_dist_dec.h and local_arrays_dec.h are used for creating the variables that store the best distance points for each partition in the case of the partial_dist_dec.h file and for the local_arrays_dec.h it is used to create copies of the instance that is being classified in order to be accessed by each partition in a parallel manner, and its contents are displayed in Code Listings 4.3 and 4.4 respectively. Since for both cases the variables being created in this file will be initialized for each new instance to classify, and as such, will be initialized in each loop until the array of instances to be classified is finished, they cannot be initialized in the at the same spot that they are declared, and as such, two additional files are be needed.

```
1     int bestPointsIdx_0[K];
2     double bestDistances_0[K];
3     int bestPointsIdx_1[K];
4     double bestDistances_1[K];
5     int bestPointsIdx_2[K];
6     double bestDistances_2[K];
```

Code Listing 4.3: Declaration of the partial distance arrays

```
1 static DATA_TYPE testing_X_0[N_FEATURES]={0};
2 static DATA_TYPE testing_X_1[N_FEATURES]={0};
3 static DATA_TYPE testing_X_2[N_FEATURES]={0};
```

Code Listing 4.4: Declaration of the partitions of the testing point

The file partial_dist_init.h is shown in Code Listings 4.5 and initializes the values for the arrays bestPointsIdx and bestDistances. This is done before beginning the classification of an instance and set the distance values for the greater value that they can take, so that distances smaller than that can be updated into the arrays.

```
1     kNN_InitBest(bestDistances_0, bestPointsIdx_0);
2     kNN_InitBest(bestDistances_1, bestPointsIdx_1);
3     kNN_InitBest(bestDistances_2, bestPointsIdx_2);
```

Code Listing 4.5: Initialization of the partial distance arrays

The file local_arrays_init.h copies the instance that is being classified into local variables of the kNN function, with the number of variables being equal to the number of partitions in which the training data set is divided and is presented in Code Listing 4.6. This is done to allow the calculations to be done in parallel, because otherwise that would not be the case as two functions cannot read from the same variable at the same time in the hardware implementation. By copying

the contents of the instance to classify in multiple local variables, it is possible to calculate the distance to the partitions of the data set in parallel.

```
for (int k = 0; k < N_FEATURES; k++){
testing_X_0[k]=testing_X[i][k];
testing_X_1[k]=testing_X[i][k];
testing_X_2[k]=testing_X[i][k];
}
```

Code Listing 4.6: Loop that copies the testing point into multiple independent variables

The file partial_predict.h instantiates the calls of the function kNN_Partial_Predict and is presented in Code Listing 4.7. The function kNN_Partial_Predict takes as arguments one of the partitions of the training data set, one copy of the instance to classify, the arrays to store the best neighbours in that partition and its id, and an integer that indicates the size of the partition of the training data set. This function is called as many times as there are partitions, until the calculation of the distances is done for all partitions.

```
    kNN_Partial_Predict(training_X_0, testing_X_0, min, max,bestPointsIdx_0,
    bestDistances_0,1446);
    kNN_Partial_Predict(training_X_1, testing_X_1, min, max,bestPointsIdx_1,
    bestDistances_1,1446);
    kNN_Partial_Predict(training_X_2, testing_X_2, min, max,bestPointsIdx_2,
    bestDistances_2,1446);
```

Code Listing 4.7: Calls of the partial distance predict funtions

The file update_best.h instantiates the calls of the function kNN_Partial_UpdateBestCaching and is shown in Code Listings 4.8. The function kNN_Partial_UpdateBestCaching takes as arguments the arrays that store the best points for a partition and the arrays that store the best points for the whole data set, as well as the id of the partition input to the function. The goal is to find the best k neighbours for the whole data set from the best k neighbours from each partition.

```
    for(int i=0;i<K;i++){

        kNN_Partial_UpdateBestCaching(bestDistances_0[i], bestPointsIdx_0[i],
                                      bestDistances, bestPointsIdx,
    bestPointsArrayIdx,0);

    }

    for(int i=0;i<K;i++){

        kNN_Partial_UpdateBestCaching(bestDistances_1[i], bestPointsIdx_1[i],
                                      bestDistances, bestPointsIdx,
    bestPointsArrayIdx,1);

    }

    for(int i=0;i<K;i++){
```

```
16
17         kNN_Partial_UpdateBestCaching(bestDistances_2[i], bestPointsIdx_2[i],
18                                       bestDistances, bestPointsIdx,
    bestPointsArrayIdx,2);
19
20     }
```

Code Listing 4.8: Calling of the UpdateBest functions

The file vote.h instantiates the calls of the function kNN_Partial_ VoteBetweenBest. The kNN_Partial_ VoteBetweenBest takes as arguments the array with the id of the best k neighbours, the training_Y array correspondent to the partition being input, the id of the said partition and the histogram and is displayed in Code Listing 4.9.

If the id of the partition and the id of a point in the array that stores the id of the partition that that point belongs, then it is checked in the training_Y, at the position of the point its class and it is stored in the histogram. At the final partition, the most represented class in the histogram is the predicted class of the instance to classify.

```
1 kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y_0,histogram,
      bestPointsArrayIdx,0);
2 kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y_1,histogram,
      bestPointsArrayIdx,1);
3 testing_Y[i]=kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y_2,histogram,
      bestPointsArrayIdx,2);
```

Code Listing 4.9: Calling of the Vote functions

4.7 Local version C code

With the aforementioned files it is possible to finally put together the C code of the local version. The local version is the version with the simpler C code, as the declaration and calling of the kNN function does not change with the number of partitions and does not need further files than those described in the previous section. This file is shown in Code Listing 4.10.

```
1 void kNN_PredictAll(DATA_TYPE testing_X[N_TESTING][N_FEATURES],
2                     CLASS_TYPE testing_Y[N_TESTING])
3 {
4     #include"./array_dec.h"
5
6     #include"./localarrays/partial_dist_dec.h"
7
8     int bestPointsIdx[K];
9     double bestDistances[K];
10    //Distances stored in doubles for better accuracy
11    int bestPointsArrayIdx[K];
12
13    CLASS_TYPE histogram[N_CLASSES] = {0};
14
```

```
15     #include "./localarrays/local_arrays_dec.h"
16
17     for (int i = 0; i < N_TESTING; i++)
18     {
19         kNN_MinMaxNormalize(min_0, max_0, testing_X[i]);
20         kNN_InitBest(bestDistances, bestPointsIdx);
21         #include "./localarrays/partial_dist_init.h"
22         #include "./localarrays/local_arrays_init.h"
23         #include "./localarrays/partial_predict.h"
24         #include "./localarrays/update_best.h"
25
26         #include "./localarrays/vote.h"
27
28         for(int i=0;i<N_CLASSES;i++){
29           histogram[i]=0;
30         }
31     }
32 }
```

Code Listing 4.10: Code for the kNN Predict All function for the local version

4.8 External and multi stage versions extra files

As mentioned before, the external and multi stage versions need two more files in order to be automated. These are the files that will declare the kNN function in the knn.h and knn.c files and call the kNN function in the main.c file.

The file that declares the kNN function and is used in the knn.h and knn.c files is called knn_dec.h and is presented in Code Listings 4.11. This file stores the declaration of the kNN function that varies with the number of partitions of the training data set.

```
1 void kNN_PredictAll(DATA_TYPE training_X_0[N_TRAINING][N_FEATURES],
2 DATA_TYPE training_X_1[N_TRAINING][N_FEATURES],
3 DATA_TYPE training_X_2[N_TRAINING][N_FEATURES],
4 CLASS_TYPE training_Y_0[N_TRAINING],
5 CLASS_TYPE training_Y_1[N_TRAINING],
6 CLASS_TYPE training_Y_2[N_TRAINING],
7 DATA_TYPE testing_X[N_TESTING][N_FEATURES],
8 CLASS_TYPE testing_Y[N_TESTING],
9 DATA_TYPE min[N_FEATURES],
10 DATA_TYPE max[N_FEATURES],)
```

Code Listing 4.11: Declaration of the training arrays in the external and multistage versions

The file that will store the calling of the kNN version is called knn_call.h and is shown in Code Listings 4.12. This gives as argument all the partitions of the training data set in the main function. It is important to remember that for the external and multi stage versions the partitions of the training data set are declared in the main function.

```
kNN_PredictAll(training_X_0,training_X_1,training_X_2,training_Y_0,training_Y_1
    ,training_Y_2,testing_X, predicted_testing_Y,min,max);
```

Code Listing 4.12: Calling of the kNN PredictAll function for the external and multistage versions

4.9 External version C code

The C code for the external version is nearly identical to the local code with only two changes: The declaration of the function is now done with an include, and the training data set is not declared in the function. As the name of the other variables and the variables that store the training data set are the same it allows for the use of the same code to generate the other files for the two versions, as described earlier. The code is displayed in Code Listing 4.13

```
#include"./external_arrays/knn_dec.txt"

{

    #include"./external_arrays/partial_dist_dec.txt"

    int bestPointsIdx[K];
    double bestDistances[K];
    int bestPointsArrayIdx[K];

    #include"./external_arrays/local_arrays_dec.txt"

    CLASS_TYPE histogram[N_CLASSES] = {0};

    for (int i = 0; i < N_TESTING; i++){

        #include"./external_arrays/partial_dist_init.txt"

        kNN_InitBest(bestDistances, bestPointsIdx);
        kNN_MinMaxNormalize(min_0, max_0, testing_X[i]);

        #include"./external_arrays/local_arrays_init.txt"

        #include"./external_arrays/partial_predict.txt"

        #include"./external_arrays/update_best.txt"

        #include"./external_arrays/vote.txt"

        for(int i=0;i<N_CLASSES;i++){
          histogram[i]=0;
        }
    }
}
```

Code Listing 4.13: Code for the external version

4.10 Multi stage version

As mentioned before, the multi stage version is a version where the training data set cannot be loaded completely into local variables of the kNN function. As such, the training data set is loaded as many times as there are partitions, and a kNN algorithm is ran on that partition. While this may seem similar to the previous versions, there is one key difference: As the training data set is copied into local variables of the kNN function, once the kNN algorithm is done for the partition it should not be copied again. This changes the approach slightly in comparison to the previous versions. As a partition cannot be access after the kNN algorithm is done for the partition, it is not possible to check its training_Y file after the distances are calculated. As such, the proposed solution is to store the class of a nearest neighbour in an array that is changed at the same time that the best distances array is changed, so that the the same position for the two arrays represent the same point, and as such is possible to associate a best distance with the class that the point belongs. Both the best distances and the array that stores the correspondent class are used by all the partitions, until all partitions are done. To accommodate these changes, some functions were slightly altered. The function kNN_Partial_UpdateBestCaching was changed to receive as an input the array that stores the class associated to a best point, with the new name kNN_Partial_UpdateBestCaching_multistage. The kNN_Partial_VoteBetweenBest function was also changed to kNN_Partial_VoteBetweenBest_multistage, with it only accessing the new classes array, and only with it vote the most represented class.

Other particular important difference between the multi stage version and the other versions is the fact that the multi stage version has two kinds of partitions: external partitions and local partitions.

The external partitions represent the number of times that partitions of the training data set are copied into local variables of the kNN function. After the kNN algorithm for an external partition of the training data set is done, the next partition is copied. This number of external partitions should be chosen around the size of the training data set and the number of BRAMs on the FPGA that will implement the algorithm. A higher number of external partitions will lead to less BRAMs being used while still having that partition of the training data set stored locally, reducing the latency of reading data outside of the FPGA. However, as there are more partitions, there are more times were data outside of the FPGA will be read which will slow down the algorithm.

The number of local partitions represent how many kNN algorithms will be done in parallel for the external partition. The external partition when being copied to local variables is stored in multiple local variables, allowing for the parallelization of the computations. This will use more resources than if no local partitions were used, but greatly helps achieving a higher speedup.

These two kinds of partitions allow for a lot of control of the resources used in this version: The number of external partitions influences the number of BRAMs that will be used, while the number of local partitions influence the other resources such as LUTs, FFs and DSPs.

The code for the multi stage version for two external partitions can be seen in Code Listing 4.14. In this code, there is only one local variable to store the training data set, which is written

twice while the code is executing. After the first partition of the training data set is copied to the local variables, the kNN algorithm function is done for that external partition, with the best distances and correspondent class being stored. After that, the second partition is copied to the local variables, and the kNN is calculated again, taking into consideration the best distances and the respective classes that were previously calculated.

```
void kNN_Multistage(DATA_TYPE training_X[N_TRAINING][N_FEATURES],
                    CLASS_TYPE training_Y[N_TRAINING],
                    DATA_TYPE testing_X[N_TESTING][N_FEATURES],
                    CLASS_TYPE testing_Y[N_TESTING],
                    DATA_TYPE min[N_FEATURES],
                    DATA_TYPE max[N_FEATURES])
{

    DATA_TYPE training_X_local[N_TRAINING/2][N_FEATURES];
    CLASS_TYPE training_Y_local[N_TRAINING/2];

    double bestDistances[N_TESTING][K];
    int bestPointsIdx[N_TESTING][K];
    CLASS_TYPE bestDistancesclass[N_TESTING][K];

    for(int i=0;i<N_TESTING;i++){
        kNN_InitBest(bestDistances[i],bestPointsIdx[i]);
        kNN_MinMaxNormalize(min, max, testing_X[i]);
    }


    for(int i=0;i<N_TRAINING/2;i++){
            for(int j=0;j<N_FEATURES;j++){
                training_X_local[i][j]=training_X[i][j];
            }
        training_Y_local[i]=training_Y[i];
    }


        kNN_PredictAll(training_X_local, training_Y_local, testing_X, testing_Y
    ,bestDistances,bestDistancesclass);

    for(int i=0;i<N_TRAINING/2;i++){
            for(int j=0;j<N_FEATURES;j++){
                training_X_local[i][j]=training_X[i+N_TRAINING/2][j];
            }
        training_Y_local[i]=training_Y[i+N_TRAINING/2];
    }

        kNN_PredictAll(training_X_local, training_Y_local, testing_X, testing_Y
    ,bestDistances,bestDistancesclass);

```

```
41 }
```

Code Listing 4.14: Code for the multistage version with 2 external partitions and 1 local partition

It is also important to understand the structure of the multi stage version where there are more than one local partition. In this case, two set of variables to store the training data set locally are created, that will later be written with values of the training data set. For this to happen, the function kNN_PredictAll should have its declaration changed to accommodate the fact that it will receive two training data sets. This process is similar to the one done for the external version, where the declaration and calling of the function changed with the number of partitions that were calculated in parallel.

The code for a multi stage version with 2 external partitions and 2 local partitions is shown in Code Listing 4.15. There are declared two sets of local variables, which will be written two times during the kNN function.

```
1  void kNN_Multistage(DATA_TYPE training_X[N_TRAINING][N_FEATURES],
2                      CLASS_TYPE training_Y[N_TRAINING],
3                      DATA_TYPE testing_X[N_TESTING][N_FEATURES],
4                      CLASS_TYPE testing_Y[N_TESTING],
5                      DATA_TYPE min[N_FEATURES],
6                      DATA_TYPE max[N_FEATURES])
7  {
8
9      DATA_TYPE training_X_0[N_TRAINING/4][N_FEATURES];
10     CLASS_TYPE training_Y_0[N_TRAINING/4];
11     DATA_TYPE training_X_1[N_TRAINING/4][N_FEATURES];
12     CLASS_TYPE training_Y_1[N_TRAINING/4];
13
14     double bestDistances[N_TESTING][K];
15     int bestPointsIdx[N_TESTING][K];
16     CLASS_TYPE bestDistancesclass[N_TESTING][K];
17
18     for(int i=0;i<N_TESTING;i++){
19         kNN_InitBest(bestDistances[i],bestPointsIdx[i]);
20         kNN_MinMaxNormalize(min, max, testing_X[i]);
21     }
22
23 //Copying the first external partition to the 2 local variables (code not shown
       )
24
25         kNN_PredictAll(training_X_0,training_X_1, training_Y_0,training_Y_1,
       testing_X, testing_Y,bestDistances,bestDistancesclass);
26
27 //Copying the second external partition to the 2 local variables (code not
       shown)
28
29
30         kNN_PredictAll(training_X_0,training_X_1, training_Y_0,training_Y_1,
       testing_X, testing_Y,bestDistances,bestDistancesclass);
```

```
31
32 }
```

Code Listing 4.15: Code for the multistage version with 2 external partitions and 2 local partitions

4.11 Multi stage version automation

The automation of the multi stage version is done by instantiating the loops that copy the training data set to local variables and by calling the kNN function with the local variables once the external partition of the data set is copied, and changing the declaration and calling of the kNN_Predict_All function in accordance to the number of local partitions. This is done with includes, being the code generated by the python script. For each external partition, there will be a block of loops that copy the training data set followed by the kNN function, in an equal number to the number of external partitions. For each local partition, there will be additional loops for coping the data set in each block. For this to be done, the first step is generating the code for the function kNN_Multistage, that will be called from the main function with the training dataset with no partitions. Its declaration and code are displayed in Code Listings 4.16.

```
1
2 void kNN_Multistage(DATA_TYPE training_X[N_TRAINING][N_FEATURES],
3                     CLASS_TYPE training_Y[N_TRAINING],
4                     DATA_TYPE testing_X[N_TESTING][N_FEATURES],
5                     CLASS_TYPE testing_Y[N_TESTING],
6                     DATA_TYPE min[N_FEATURES],
7                     DATA_TYPE max[N_FEATURES])
8 {
9
10  DATA_TYPE local_min[N_FEATURES];
11  DATA_TYPE local_max[N_FEATURES];
12
13     for(int i=0;i<N_FEATURES;i++){
14         local_min[i]=min[i];
15         local_max[i]=max[i];
16     }
17
18     #include"./part_arrays/knn_multistage_code.h"
19
20 }
```

Code Listing 4.16: Code for the multistage function

The bulk of the code where the copying loops and calls of the kNN_PredictAll functions are done in the file "/part_arrays/knn_multistage_code.h", since the loops and calling of the kNN_PredictAll function will vary with the number of local and external partitions desired. An example of the code generates for an example of 1 external partition and 3 local partitions in the file "/part_arrays/knn_multistage_code.h" is described in Code Listing 4.17.

```
    for(int i=0;i<1446;i++){
for(int j=0;j<N_FEATURES;j++){
training_X_0[i][j]=training_X[i+0][j];
}
training_Y_0[i]=training_Y[i+0];
}

    for(int i=0;i<1446;i++){
for(int j=0;j<N_FEATURES;j++){
training_X_1[i][j]=training_X[i+1445][j];
}
training_Y_1[i]=training_Y[i+1445];
}

    for(int i=0;i<1446;i++){
for(int j=0;j<N_FEATURES;j++){
training_X_2[i][j]=training_X[i+2890][j];
}
training_Y_2[i]=training_Y[i+2890];
}

kNN_PredictAll(training_X_0,training_X_1,training_X_2,training_Y_0,training_Y_1
    ,training_Y_2,testing_X, testing_Y,bestDistances,bestDistancesclass);
```

Code Listing 4.17: Code for the multistage version with 1 external partition and 3 local partitions

The function kNN_PredictAll is also changes with the number of local partitions, taking as many arrays of the training dataset as arguments as there are local partitions. This means that the declaration of the kNN_PredictAll function will also vary, as well as the contents of the files included in it. The structure of the kNN_PredictAll function is presented in Code Listings 4.18

```

#include"./part_arrays/knn_dec.h"
{
    #include"./part_arrays/partial_dist_dec.h"
    #include"./part_arrays/local_arrays_dec.h"
    for (int i = 0; i < N_TESTING; i++){

        #include"./part_arrays/partial_dist_init.h"

        #include"./part_arrays/local_arrays_init.h"

        #include"./part_arrays/partial_predict.h"

        #include"./part_arrays/update_best.h"

        testing_Y[i] = kNN_VoteBetweenBest_Class(bestDistancesclass[i]);
    }
}
```

Code Listing 4.18: Declaration and code of the kNN_PredictAll function

The contents of the files included are almost identical of the files with the same name described before, except the changes to the kNN_Partial_UpdateBestCaching and kNN_VoteBetweenBest functions described before. As the array bestDistancesclass stores the class of the best points, that is being changed by the function kNN_Partial_UpdateBestCaching_Class as new best points are found, the function kNN_VoteBetweenBest_Class is only called once at the end of the classification.

The automatization of the multi stage version is fundamental to achieve a good implementation of the kNN algorithm. Since this is the most flexible code, it is the one that can better adjust to the required training data sets and FPGA to be used.

4.12 Summary

In this chapter three new versions of the kNN algorithm were presented: The local version, the external version and the multi stage version. All the versions have its uses and applications, and as the solutions are synthesized and programmed into the FPGA, a better description of them will be achieved. The partitions are the main theme of all versions, with the goal of achieving parallel computations by using different partitions. To implement any number of partitions for a version of the kNN algorithm, a script that automatizes the writing of the code was developed for each version. The generated code properly inserted in the code that implements each partition, allowing flexibility in choosing the number of partitions that are desired for a version of the kNN algorithm.

Chapter 5

Experimental results

5.1 Experimental setup

In order to evaluate the different implementations of the kNN algorithm, multiple experiments were made in order to evaluate and compare the implementations.

The first evaluation metric uses the Vitis HLS synthesis report. This is the report obtained after the synthetization of a C code that provides estimates to the latency and minimum clock period of the RTL implementation of the corresponding C code, as well as estimates for the number of BRAMs, DSPs, LUTs and FFs used by the RTL implementation. These estimates are useful in comparing the synthesized version of different implementations but as the place and route did not take place, those are just rough estimates that need to be verified when programmed in the FPGA.

The second metric was the latency of the function in the FPGA versus the latency of the same function ran in a CPU. This better displays the acceleration that FPGA implementations of a function can achieve when compared to a CPU version of the same function. Clock cycles were used to compare the two since an FPGA and a CPU are usually ran at different clock frequencies to better represent the acceleration obtained via FPGA implementations.

The target FPGA for the Vitis HLS synthesis was the ZYNQ XC7Z020-1CLG400C, that has 220 DSPs, 53200 LUTs, 106400 FFs and 140 BRAMs. The data set used in the evaluation was the WISDM data set [19]. The data set chosen was the WI_K3_F data set, a data set with a low k of 3 and single precision floating point for the training and testing instances, with 4336 training samples, 1082 testing samples, where each data point has 43 features.

5.2 FPGA results

The first evaluation was the synthesis evaluation, where the solutions were evaluated in Vitis HLS after the synthesis of their C code. This evaluation provides estimated results reported by Vitis HLS for latency, clock cycles, speedup in comparison to a benchmark and resources used in each implementation. This provides parameters that characterize the implementation and allows for comparison between the implementations.

The benchmark for the results obtained in this section are the results obtained from synthesizing the kNN original code[18]. This benchmark is be used when calculating the speedups obtained from the different implementations that were evaluated. The chosen target clock was 15 nanoseconds, which corresponds to a frequency clock of 75 MHz. This relatively low frequency was chosen in order to prevent negative slack from occurring when analysing the different implementations, and thus assuring that all implementations are comparable by using a clock frequency that makes it less likely to have negative slack when synthesizing the code.

The data set used was the WI_K3_F . The obtained latencies and speedups for different implementations were obtained and measured.

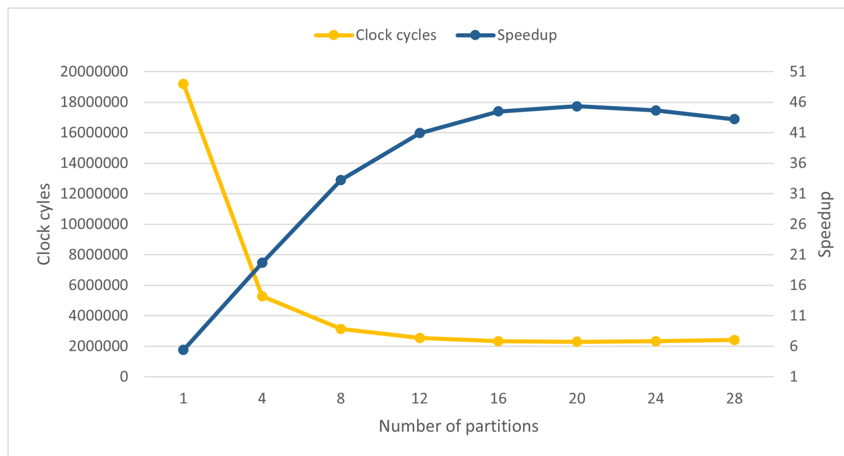

Figure 5.1: Local version latency and speedup per partition over benchmark

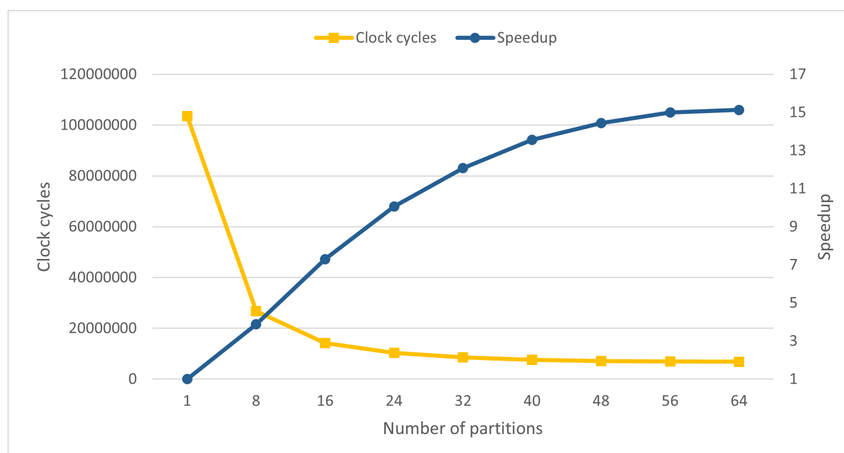

Figure 5.2: External version latency and speedup per partition over benchmark

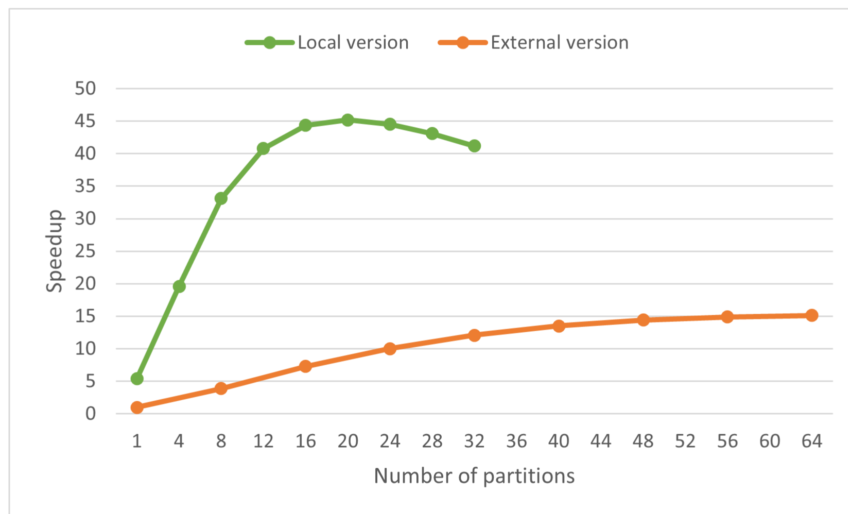

Figure 5.3: Speedups obtained from both local and external versions over benchmark

As expected, as the number of partitions increases, so does the speedup obtained from each partition, although at a slowing rate each time the number of partitions is increased. The number of partitions in the local version increased by 4 each time, while in the external version increased by 8. The goal was to find if at a high number of partitions, the speedup starts to decrease instead of increasing due to the higher reading times, and if does, for which number of partitions does that occur and which number of partitions provides the highest speedup.

For the local version, that did occur for 20 partitions, where the speedup obtained was 44.49 . After that, as the number of partitions increased, the speedup began to decrease. For the external version, although the speedup always increased for the measured partitions, the increases were lower for each partition, from which it is expected that also for the external version the speedup would begin to decrease after a certain number of partitions. The highest speedup measured for the external version was of 15.10.

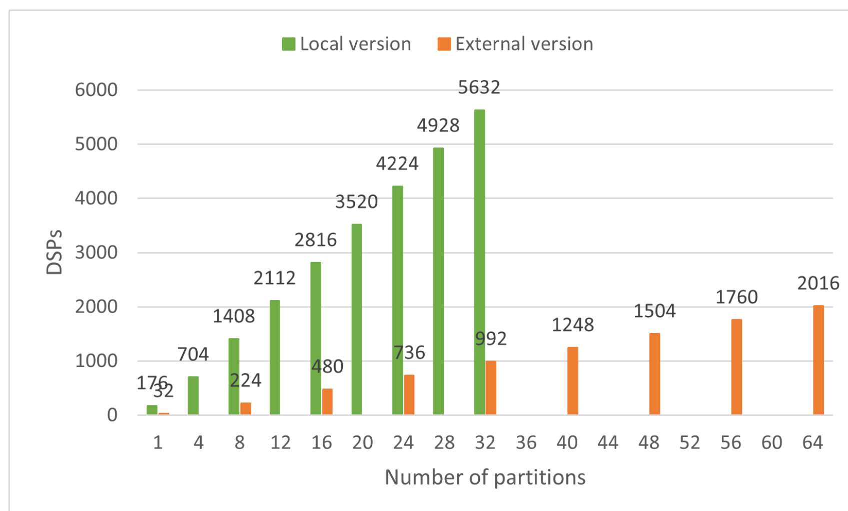

Figure 5.4: DSPs used per partition for local and external versions

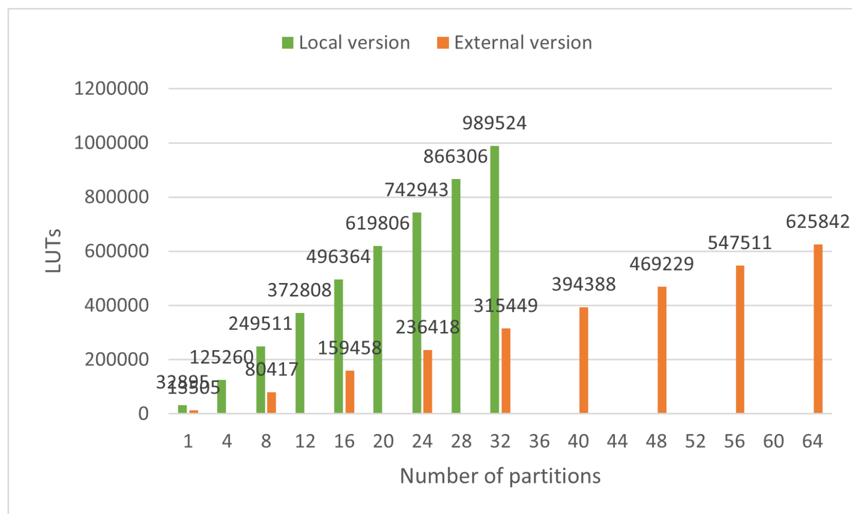

Figure 5.5: LUTs used per partition for local and external versions

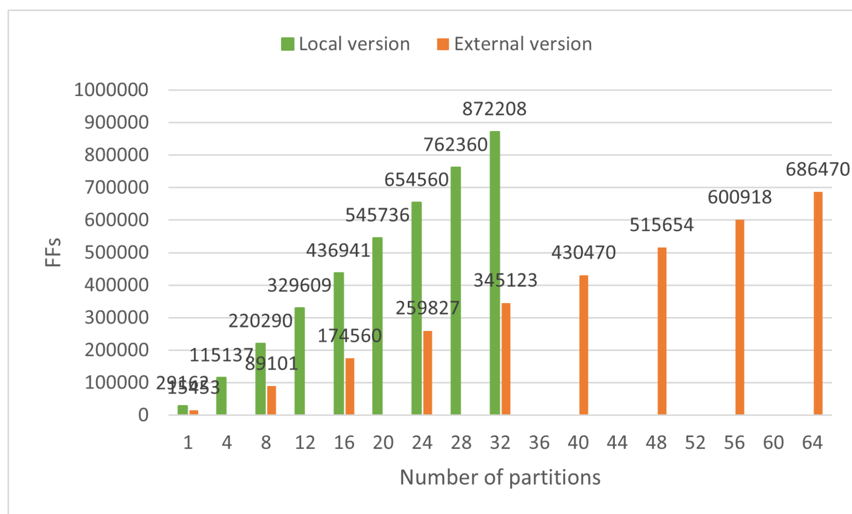

Figure 5.6: FFs used per partition for local and external versions

The number of DSPs, LUTs and FFs used by each implementation increases as the number of partitions increases, as expected. The number of resources used by implementation when comparing the local version to the external version, for the same number of partitions, is approximately four times higher. This can be explained due to the higher number of operations that are made in parallel in each partition of the local version, in order to take advantage of the lower initiation interval possible due to the data being stored locally with faster reading times, as explained in the previous chapter, which leads to a higher usage of resources.

Num partitions	1	4	8	12	16	20	24	28	32
Latency (ns)	2.88E+08	7.93E+07	4.69E+07	3.81E+07	3.50E+07	3.44E+07	3.49E+07	3.61E+07	3.77E+07
Clock cycles	1.92E+07	5.28E+07	3.13E+06	2.54E+06	2.34E+06	2.29E+07	2.33E+06	2.41E+06	2.51E+06
BRAM	678	686	698	550	706	886	1066	1238	1426
DSP	176	704	1408	2112	2816	3520	4224	4928	5632
FF(k)	29162	115137	220290	329609	436941	545736	654560	762360	872208
LUT(k)	32895	125260	249511	372808	496364	619806	742943	866306	989524

Table 5.1: Resources used per partition for local version

Num partitions	1	8	16	24	32	40	48	56	64
Latency (ns)	1.55E+09	4.01E+08	2.14E+08	1.55E+08	1.29E+08	1.15E+08	1.08E+08	1.04E+08	1.03E+08
Clock cycles	1.04E+08	2.68E+07	1.42E+07	1.03E+07	8.57E+06	7.67E+06	7.19E+06	6.95E+06	6.86E+06
BRAM	0	0	0	0	0	0	0	0	0
DSP	32	224	480	736	992	1248	1504	1760	2016
FF	15453	89101	174560	259827	345123	430470	515654	600918	686470
LUT	13505	80417	159458	236418	315449	394388	469229	547511	625842

Table 5.2: Resources used per partition for external version

The main difference between the two versions is the use of BRAMs by the local version. As in the local version the data set is stored locally, it is stored using BRAMs. Initially, in the local version, the number of BRAMs is almost the same as the number of partitions increase. However, for higher number of partitions the number of BRAMs begin to increase instead of staying the same. This can be explained since for higher partitions, the size of each partition is increasingly small, leading to all the memory of some of the BRAMs not being used, while still requiring more BRAMs to be used as each partition is stored in its own BRAM.

The increase in the number of DSPs, FFs and LUTs, for both versions, is approximately linear, meaning that the increase in the number of this resources is proportional to the increase in the number of partitions.

It should also be noted that while Vitis HLS provides number of resources to be used by each number of partitions for each version, in fact from the versions displayed here only the external version with 1 partitions is able to be ran in the ZYNQ XC7Z020-1CLG400C FPGA, due to the limited number of resources available. If any other version with more partitions was to be ran in an FPGA, it would need to be ran in an FPGA with more resources.

The multistage version was also synthesized in Vitis HLS in order to verify if the assumptions about the effect of the number of local and external partitions were right, namely, if by increasing the number of external partitions the use of BRAMs did decrease and if by increasing the number of local partitions the latency decreased and the number of resources used increased.

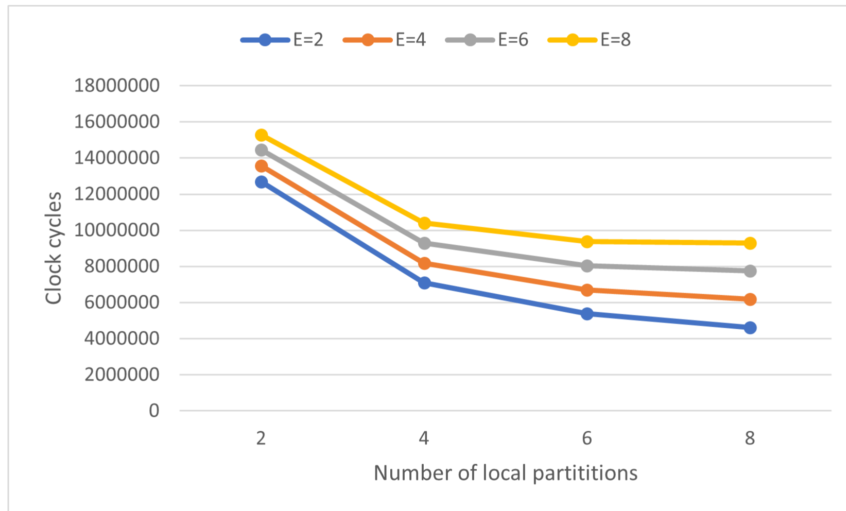

Figure 5.7: Latency curves for the multistage version

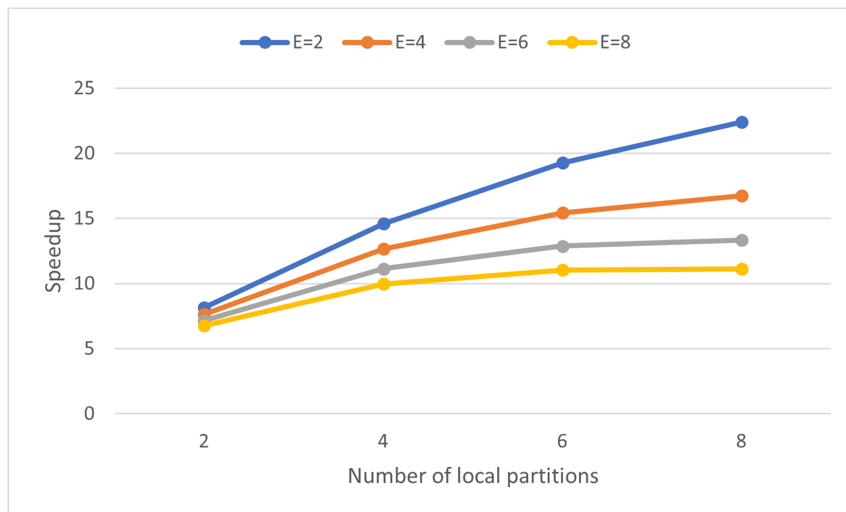

Figure 5.8: Speedup curves for the multistage version over benchmark

As expected, the latency of each solution decreased with the increase in the number of local partitions. The number of external partitions did not have a great impact in the latency, being the latency for versions with the same number of local partitions nearly the same, with the increase in external partitions leading to small increases in the latency due to the overhead introduced when copying external data into the accelerator.

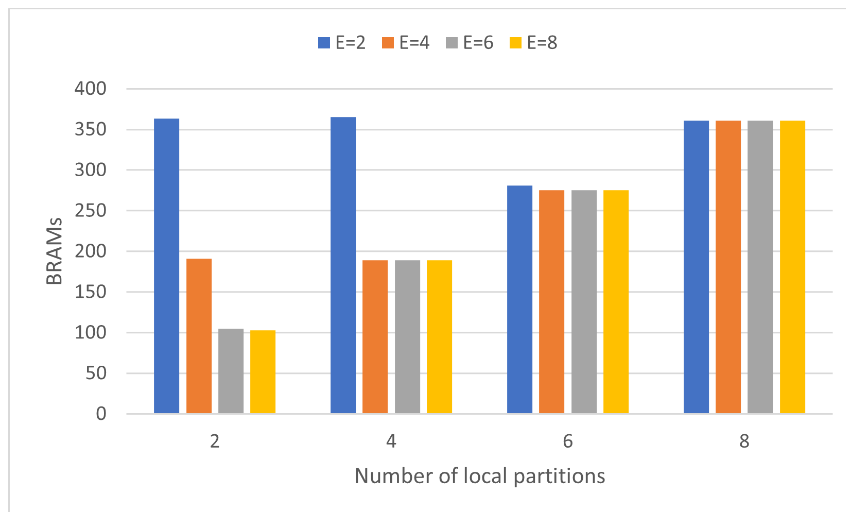

Figure 5.9: Number of BRAMs used by each multistage solution

As for the BRAM use, although it tends to decrease with the increase in the number of external partitions, it did not always happen, leading to situations where an increase in the number of external partitions did not decrease the number of BRAMs, but still there was not a situation where the number of BRAMs did increase. This leads to the conclusion that arbitrarily increasing the number of external partitions might not reduce the number of BRAMs being used, and a solution must be carefully analysed in order to pick the lowest number of external partitions that will use maximum number of BRAMs that its desired.

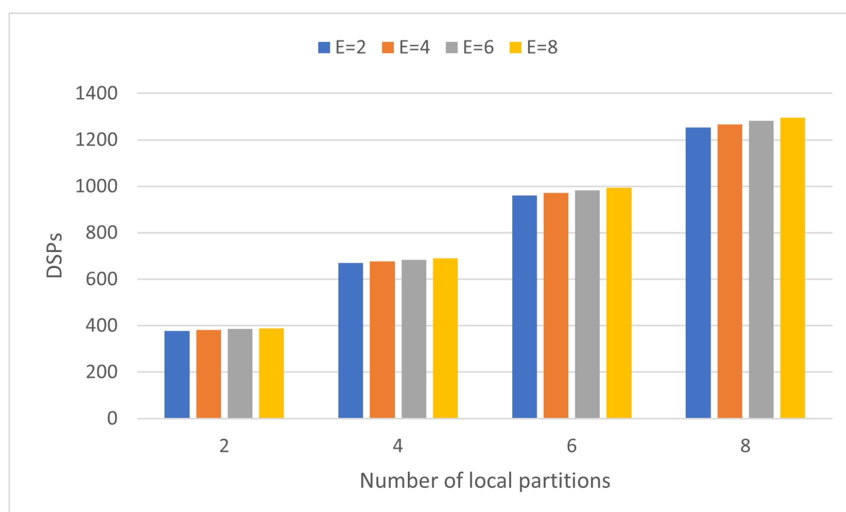

Figure 5.10: Number of DSPs used by each multistage solution

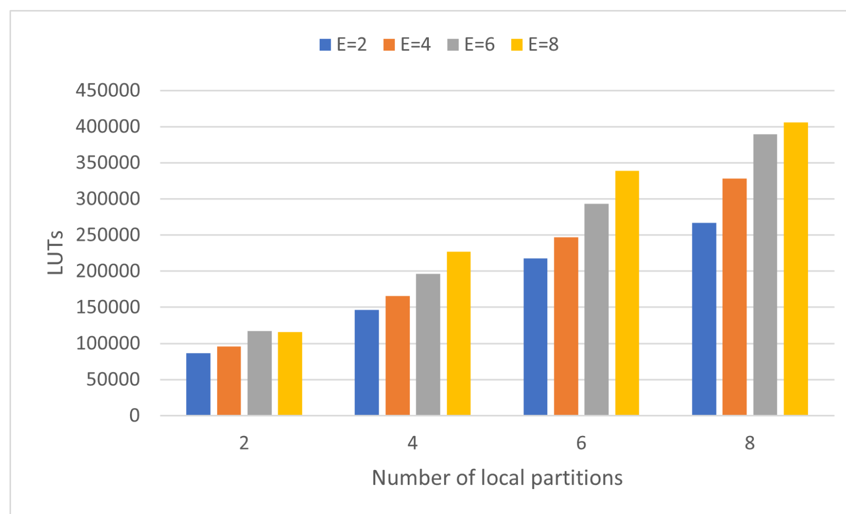

Figure 5.11: Number of LUTs used by each multistage solution

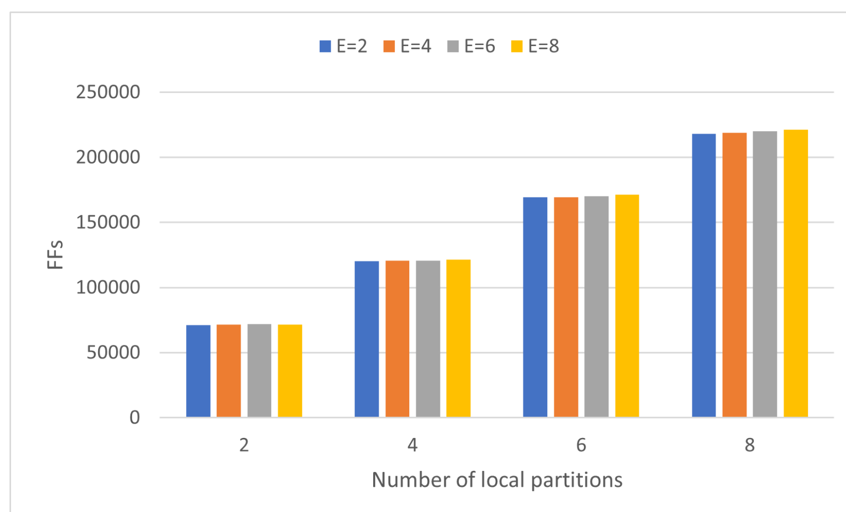

Figure 5.12: Number of FFs used by each multistage solution

Analysing the resources used for the multistage version, it is possible to observe that when increasing the number of external partitions, for the same number of local partitions, the number of DSPs and FFs stayed nearly the same, while the number of LUTs increased with the increase in the number of external partitions. This can be explained with the fact that when increasing the number of external partitions there are no more kNN computation being done in parallel, but there is only an increase in the loops used to copy the external variables into local ones. As there are no more computations being done in parallel, the number of DSPs stayed nearly the same, as well as the number of FFs. Since when synthesized C loops will mostly use LUTs, this explains the fact that when increasing the number of loops used to copy external variables into local ones it will lead to an increase in the number of LUTs used.

5.3 CPU vs FPGA results

To compare the obtained implementations in an FPGA with a CPU version, the kNN code was ran in an ARM processor, the same type of typically processor used in CPU+FPGA SoC. The CPU estimates were obtained from running the kNN code in an RaspberryPi 4, which uses the ARM based processor Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz, running raspbian OS, while the FPGA estimates were the ones obtained in Vitis HLS via synthesizing C code, with the target clock frequency for the FPGA of 75 MHz. This CPU estimates were preferred instead of running the multiple versions in a CPU of a SoC with a FPGA due to the ease of running multiple different versions. Although the conditions are very different from the ones found in SoC CPU+FPGA where only a CPU is running, namely due to the fact that by running an operating system the time measurements of the duration of the kNN function might be affected from other processes ran in the same OS, the time estimates can be used to give a rough estimate of the number of clock cycles used by each implementation.

The C code of each version with various number of partitions was compiled in gcc with the optimization flag of -O3, in order to have an optimized CPU version of each version. To obtain the clock cycles, each version with varying number of partitions was ran 100 times in the RaspberryPi, and the average time of those runs was used to get an average running time for each implementation. This time was then used to estimate the number of clock cycles used by dividing the time with the clock frequency of the CPU, which is 1.8 GHz.

This number of estimated clock cycles then was used to calculate an estimation for the speedup achieved for each version with different number of partitions when comparing an FPGA implementation with a CPU one. The results are shown in Figures 5.13 and 5.14.

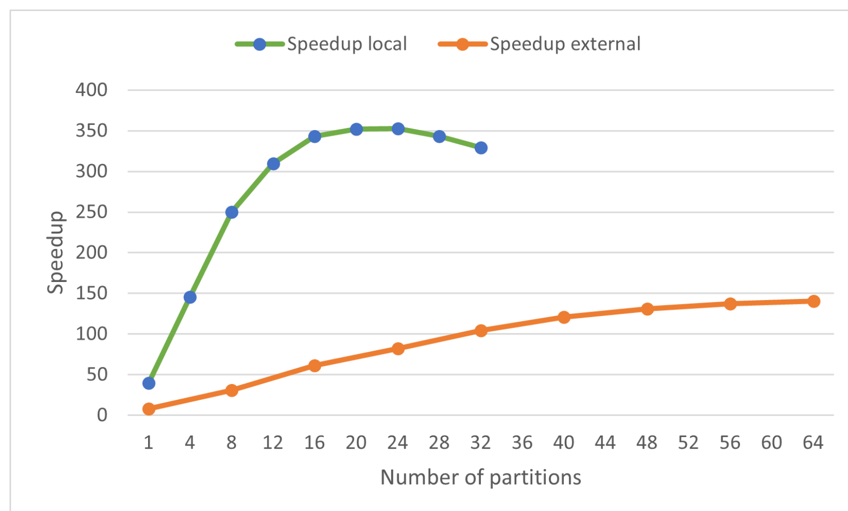

Figure 5.13: Speedup of FPGA over CPU implementations of the local and external versions

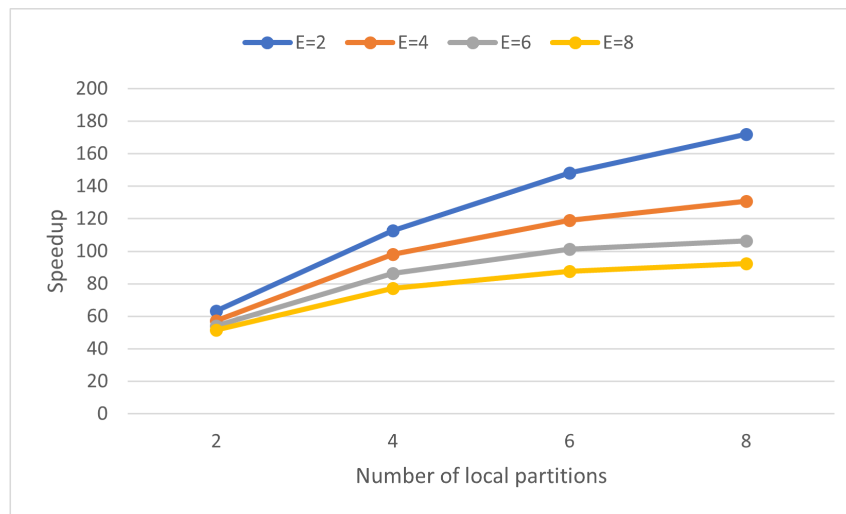

Figure 5.14: Speedup of FPGA over CPU implementations of the multistage version

The speedup curves are nearly identical to the ones obtained before where the different versions were compared to a FPGA implementation of the original code. This is due to the fact that the running time of the kNN function was nearly constant, only increasing in the order of the hundredths of second with the increase in the number of partitions. Also, the speedups presented here should be taken with scepticism, due to the reasons referred before about the CPU estimates. The lowest speedup estimation was obtained for the external version with 1 partition (that is identical to the original code) with a speedup of 7.71, showing that even in the worst circumstance there is still a significant speedup achieved by using a FPGA implementation. The highest speedup estimated was for the local version with 24 partitions with a estimated speedup of 352.59.

Chapter 6

Conclusion

6.1 Concluding remarks

Throughout this dissertation it was shown that fast kNN machines are made possible by the use of heterogeneous systems. By using code transformations and automations it was possible to propose a flexible approach to the kNN algorithm that is able to be adapted for different circumstances regarding the target FPGA and the training data set used, with the maximum speedup when compared to the original code achieved by the local version with a speedup of 44.49 times. The external version achieved the maximum speedup at 64 partitions with a speedup of 15.10, and the multistage version achieved a speedup of 22.41 for 2 external partitions and 8 local partitions.

The code transformations were fundamental in achieving efficient solutions. Proper care was taken when developing C code that was to be synthesized via high level synthesis as the tools used are still dependent of the input code styles to produce more efficient solutions.

A flexible solution to target multiple implementations of the kNN algorithm was achieved. This is important since not all FPGA+CPU platforms are able to support large data sets that might be required to be used by some applications, and as such, a lower speedup may be obtained in such platforms, and in the other hand if the board can store the training data set in its memory it is possible to achieve a higher degree of speedup. The trade-off between resources used and achieved speedup should be set around the training data set and the CPU+FPGA platform used.

The varying number of partitions is the factor of flexibility to the solutions. As demonstrated, for a small number of partitions of the local version the number of BRAMs used is nearly constant, for the external version no BRAMs are used and for the multistage version the number of BRAMs used tends to decrease with the increase in the number of external partitions, and as such the three proposed versions are suited for varying scenarios depending on the training data set size and resources available at the FPGA+CPU platform.

The number of partitions must be carefully chosen depending on the requirements of the specific application, for example, if the application requires a latency below a certain threshold or if the highest speedup is wished for a certain FPGA+CPU SoC platform that has a given number of resources, giving the developer the possibility to choose the better option for the given application.

The C code for the multiple versions proposed was created using a script that automated the creation of C code that once synthesized implemented the proposed solution, with the script taking as arguments the number of partitions desired for one of the proposed solutions. As the script is able to generate code for an arbitrary number of partitions, it makes possible for developers to explore and implement multiple solutions with different versions and varying number of partitions in an efficient away.

The division of the training data set is also an approach that can be used for other applications that require the same partitions that were made in the implemented solutions. By reusing the code, it is possible to use it in other contexts for accelerating different machine learning algorithms using FPGAs, facilitating future work that might be done with other algorithms.

6.2 Future work

As future work, we propose the following:

- Using a hardware implementation of the insertion algorithm that inserts a new instance in a single clock cycle. This could be used in the update best functions in order to decrease the initiation interval between consecutive calls of the function;
- Using pragmas systematically to control the number of resources used. In the local version, an increase in the iteration interval could lead to a decrease in the number of resources used by that version, while still providing a speedup when compared to the external version;
- Parallelizing the calculation of the distance between an instance to classify and a point in the training data set. This might be more relevant for implementations that only classify one single instance with a high number of features, since the calculation of the distance of an instance to a point in the training data set for all of their features is not parallelized;
- Use of analytical models to estimate FPGA resource and performance to decide about configuration to be used according to the data set and target FPGA;
- Research implementations of pipelining and streaming considering successive points to classify.

References

[1] Xilinx. Vitis high-level synthesis user guide. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Producer-Consumer-Paradigm, 2023.

[2] Kashvi Taunk, Sanjukta De, Srishti Verma, and Aleena Swetapadma. A brief review of nearest neighbor algorithm for learning and classification. In *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, pages 1255–1260, 2019. doi:10.1109/ICCS45141.2019.9065747.

[3] Xilinx. *Vitis Unified Software Platform Documentation : Application Acceleration Development*, 12 2022. URL: https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration.

[4] Yuliang Pu, Jun Peng, Letian Huang, and John Chen. An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 167–170, 2015. doi:10.1109/FCCM.2015.7.

[5] João Vieira, Rui P. Duarte, and Horácio C. Neto. knn-stuff: knn streaming unit for fpgas. *IEEE Access*, 7:170864–170877, 2019. doi:10.1109/ACCESS.2019.2955864.

[6] Huan Feng, David Eyers, Steven Mills, Yongwei Wu, and Zhiyi Huang. Principal component analysis based filtering for scalable, high precision k-nn search. *IEEE Transactions on Computers*, 67(2):252–267, 2018. doi:10.1109/TC.2017.2748131.

[7] Xiaojia Song, Tao Xie, and Stephen Fischer. A memory-access-efficient adaptive implementation of knn on fpga through hls. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 177–180, 2019. doi:10.1109/ICCD46524.2019.00030.

[8] Katayoun Neshatpour, Hosein Mohammadi Mokrani, Avesta Sasan, Hassan Ghasemzadeh, Setareh Rafatirad, and Houman Homayoun. Architectural considerations for fpga acceleration of machine learning applications in mapreduce. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '18, page 89–96, New York, NY, USA, 2018. Association for Computing Machinery. URL: https://doi.org/10.1145/3229631.3229639, doi:10.1145/3229631.3229639.

[9] Andrew A. Chien and Vijay Karamcheti. Moore's law: The first ending and a new beginning. *Computer*, 46(12):48–53, 2013. doi:10.1109/MC.2013.431.

[10] R. Stanley Williams. What's next? [the end of moore's law]. *Computing in Science Engineering*, 19(2):7–13, 2017. doi:10.1109/MCSE.2017.31.

[11] Hideharu Amano. *Principles and Structures of FPGAs*. Springer Singapore, 1 edition, 2018.

[12] Douglas E. Ott and Thomas J. Wilderotter. *A Designer's Guide to VHDL Synthesis*. Springer New York, NY, 1 edition, 2010.

[13] Philippe Coussy and Adam Morawiec. *High-Level Synthesis*. Springer Dordrecht, 1 edition, 2008.

[14] Xilinx. Pynq-z2 board featuring the zynq xc7z020-1clg400c soc. https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html, 2023.

[15] Ashrak Rahman Lipu, Ruhul Amin, Md. Nazrul Islam Mondal, and Md. Al Mamun. Exploiting parallelism for faster implementation of bubble sort algorithm using fpga. In *2016 2nd International Conference on Electrical, Computer Telecommunication Engineering (ICECTE)*, pages 1–4, 2016. doi:10.1109/ICECTE.2016.7879576.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008. URL: https://doi.org/10.1145/1327452.1327492, doi:10.1145/1327452.1327492.

[17] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 4 edition, 2009.

[18] Tiago Santos. Newbenchmarksuite. https://github.com/tiagolascasas/Taskmark, 2023.

[19] Gary Weiss. WISDM Smartphone and Smartwatch Activity and Biometrics Dataset . UCI Machine Learning Repository, 2019. DOI: https://doi.org/10.24432/C5HK59.

Appendix A

kNN Added functions

A.1 Initial version added functions

```
double kNN_Partial_UpdateBestCaching(double queryDistance, int queryIdx,
                                    double bestDistances[K], int bestPointsIdx[K],
                  int bestPointsArrayIdx[K], int arrayIdx)
{
    double worstOfBest = 0;
    int worstOfBestIdx = -1;
    double secondWorstOfBest = 0;
    int secondWorstOfBestIdx = -1;
    static int count=0;

    for (int i = 0; i < K; i++)
    {
        if (worstOfBest < bestDistances[i])
        {
            secondWorstOfBest = worstOfBest;
            secondWorstOfBestIdx = worstOfBestIdx;

            worstOfBest = bestDistances[i];
            worstOfBestIdx = i;
        }
        else if (secondWorstOfBest < bestDistances[i])
        {
            secondWorstOfBest = bestDistances[i];
            secondWorstOfBestIdx = i;
        }
    }

    if (queryDistance < worstOfBest)
    {
        bestDistances[worstOfBestIdx] = queryDistance;
        bestPointsIdx[worstOfBestIdx] = queryIdx;
        bestPointsArrayIdx[worstOfBestIdx] = arrayIdx;
```

```
33      }
34      return (queryDistance > secondWorstOfBest) ? queryDistance
35                                                 : secondWorstOfBest;
36  }
37
38  void kNN_Partial_Predict(DATA_TYPE training_X[N_TRAINING][N_FEATURES],
39                           DATA_TYPE queryDatapoint[N_FEATURES],
40                           DATA_TYPE min[N_FEATURES], DATA_TYPE max[N_FEATURES],
41                           int bestPointsIdx[K], double bestDistances[K],
42                           int array_size)
43  {
44      double bestDistanceMax = DBL_MAX;
45
46      kNN_InitBest(bestDistances, bestPointsIdx);
47
48      for (int i = 0; i < (array_size); i++)
49      {
50          double distance = 0.0F;
51
52          for (int j = 0; j < N_FEATURES; j++)
53          {
54              DATA_TYPE feature = queryDatapoint[j];
55              double diff = feature - training_X[i][j];
56              distance += diff * diff;
57
58          }
59
60          if (distance < bestDistanceMax)
61          {
62              bestDistanceMax = kNN_UpdateBestCaching(distance, i, bestDistances,
63              bestPointsIdx);
64          }
65      }
66  }
67
68  CLASS_TYPE kNN_Partial_VoteBetweenBest(int bestPointsIdx[K],
69                                         CLASS_TYPE training_Y[N_TRAINING],
70                                         CLASS_TYPE histogram[N_CLASSES],
71                                         int bestPointsArrayIdx[K],
72                                         int arrayId)
73  {
74      for (int i = 0; i < K; i++)
75      {
76          if(bestPointsArrayIdx[i]==arrayId)
77              {
78                  int bestIdx = bestPointsIdx[i];
79                  CLASS_TYPE cl = training_Y[bestIdx];
80                  histogram[(int)cl]++;
81              }
```

```
82     }
83
84     CLASS_TYPE mostPopular = -1;
85     int mostPopularCount = -1;
86
87     for (int i = 0; i < N_CLASSES; i++)
88     {
89         if (histogram[i] > mostPopularCount)
90         {
91             mostPopularCount = histogram[i];
92             mostPopular = (CLASS_TYPE)i;
93         }
94     }
95     return mostPopular;
96 }
```

A.2 Multistage version added functions

```
1 CLASS_TYPE kNN_Partial_VoteBetweenBest_multistage(CLASS_TYPE bestDistancesclass
      [K])
2 {
3     CLASS_TYPE histogram[N_CLASSES] = {0};
4
5     for (int i = 0; i < K; i++)
6     {
7         CLASS_TYPE cl = bestDistancesclass[i];
8         histogram[(int)cl]++;
9     }
10
11    CLASS_TYPE mostPopular = -1;
12    int mostPopularCount = -1;
13
14    for (int i = 0; i < N_CLASSES; i++)
15    {
16        if (histogram[i] > mostPopularCount)
17        {
18            mostPopularCount = histogram[i];
19            mostPopular = (CLASS_TYPE)i;
20        }
21    }
22    return mostPopular;
23 }
24
25 double kNN_Partial_UpdateBestCaching_multistage(double queryDistance,
      CLASS_TYPE queryClass,
26                                                   double bestDistances[K], CLASS_TYPE
      bestDistancesclass[K])
27 {
```

```
28      double worstOfBest = 0;
29      int worstOfBestIdx = -1;
30      double secondWorstOfBest = 0;
31      int secondWorstOfBestIdx = -1;
32      static int count=0;
33
34      for (int i = 0; i < K; i++)
35      {
36          if (worstOfBest < bestDistances[i])
37          {
38              secondWorstOfBest = worstOfBest;
39              secondWorstOfBestIdx = worstOfBestIdx;
40
41              worstOfBest = bestDistances[i];
42              worstOfBestIdx = i;
43          }
44          else if (secondWorstOfBest < bestDistances[i])
45          {
46              secondWorstOfBest = bestDistances[i];
47              secondWorstOfBestIdx = i;
48          }
49      }
50
51      if (queryDistance < worstOfBest)
52      {
53          bestDistances[worstOfBestIdx] = queryDistance;
54          bestDistancesclass[worstOfBestIdx] = queryClass;
55      }
56      return (queryDistance > secondWorstOfBest) ? queryDistance
57                                                  : secondWorstOfBest;
58  }
```

A.3 Initial kNN code with 2 partitions

```
1   void kNN_PredictAll(DATA_TYPE training_X[N_TRAINING][N_FEATURES],
2                       CLASS_TYPE training_Y[N_TRAINING],
3                       DATA_TYPE testing_X[N_TESTING][N_FEATURES],
4                       CLASS_TYPE testing_Y[N_TESTING], DATA_TYPE min[N_FEATURES],
5                       DATA_TYPE max[N_FEATURES])
6   {
7
8       // Declare training and min max arrays
9
10      DATA_TYPE training_X_first[(N_TRAINING/2)][N_FEATURES];
11      DATA_TYPE training_X_second[(N_TRAINING/2)][N_FEATURES];
12
13      CLASS_TYPE training_Y_first[N_TRAINING/2];
14      CLASS_TYPE training_Y_second[N_TRAINING/2];
```

```
15
16     DATA_TYPE min_first[N_FEATURES];
17     DATA_TYPE min_second[N_FEATURES];
18
19     DATA_TYPE max_first[N_FEATURES];
20     DATA_TYPE max_second[N_FEATURES];
21
22     // Separate the training points array into two arrays
23
24     for (int i = 0; i < (N_TRAINING/2); i++)
25     {
26         training_Y_first[i]=training_Y[i];
27
28         for (int j = 0; j < N_FEATURES; j++)
29         {
30     training_X_first[i][j]=training_X[i][j];
31         }
32     }
33
34     for (int i = (N_TRAINING/2); i < N_TRAINING; i++)
35     {
36         training_Y_second[(i-(N_TRAINING/2))]=training_Y[i];
37
38         for (int j = 0; j < N_FEATURES; j++)
39         {
40     training_X_second[(i-(N_TRAINING/2))][j]=training_X[i][j];
41         }
42     }
43
44     // Duplicate the min max arrays
45
46     for (int i = 0; i < N_FEATURES; i++)
47     {
48         min_first[i]=min[i];
49         min_second[i]=min[i];
50         max_first[i]=max[i];
51         max_second[i]=max[i];
52     }
53
54     // Create arrays to store the best points
55
56     int bestPointsIdx_first[K];
57     int bestPointsIdx_second[K];
58     double bestDistances_first[K];
59     double bestDistances_second[K];
60     int bestPointsIdx[K];
61     double bestDistances[K];
62     int bestPointsArrayIdx[K];
63
```

```
64    kNN_InitBest(bestDistances, bestPointsIdx);
65
66    CLASS_TYPE voteResult;
67
68    CLASS_TYPE histogram[N_CLASSES] = {0};
69
70    static DATA_TYPE testing_X_0[N_FEATURES] = {0};
71    static DATA_TYPE testing_X_1[N_FEATURES] = {0};
72
73
74    for (int i = 0; i < N_TESTING; i++)
75    {
76        kNN_MinMaxNormalize(min, max, testing_X[i]);
77        for (int k = 0; k < N_FEATURES; k++)
78        {
79          testing_X_0[k]=testing_X[i][k];
80          testing_X_1[k]=testing_X[i][k];
81        }
82
83        //Init best distance arrays
84
85        kNN_InitBest(bestDistances, bestPointsIdx);
86        kNN_InitBest(bestDistances_first, bestPointsIdx_first);
87        kNN_InitBest(bestDistances_second, bestPointsIdx_second);
88
89        //Calculate the best points for each partition
90
91        kNN_Partial_Predict(training_X_first, testing_X_0, min_first,
92                            max_first,bestPointsIdx_first,
93                            bestDistances_first,2);
94        kNN_Partial_Predict(training_X_second, testing_X_1, min_second,
95                            max_second,bestPointsIdx_second,
96                            bestDistances_second,2);
97
98        //Calculate the best points between all partitions
99
100       for(int i=0;i<K;i++)
101       {
102           kNN_Partial_UpdateBestCaching(bestDistances_first[i],
103           bestPointsIdx_first[i],bestDistances,bestPointsIdx,
104           bestPointsArrayIdx,1);
105       }
106
107       for(int i=0;i<K;i++)
108       {
109           kNN_Partial_UpdateBestCaching(bestDistances_second[i],
110           bestPointsIdx_second[i],bestDistances,bestPointsIdx,
111           bestPointsArrayIdx,2);
112       }
```

```
113
114         //Vote the most represented class in best points
115
116         kNN_Partial_VoteBetweenBest(bestPointsIdx,training_Y_first,
117                                     histogram,bestPointsArrayIdx,1);
118
119         testing_Y[i]=kNN_Partial_VoteBetweenBest(bestPointsIdx,
120                                                  training_Y_second,histogram,
121                                                  bestPointsArrayIdx,2);
122
123         for(int i=0;i<N_CLASSES;i++)
124         {
125            histogram[i]=0;
126         }
127     }
128 }
```

A.4 Initial kNN code with 4 partitions

```
1  void kNN_PredictAll(DATA_TYPE training_X[N_TRAINING][N_FEATURES],
2                     CLASS_TYPE training_Y[N_TRAINING],
3                     DATA_TYPE testing_X[N_TESTING][N_FEATURES],
4                     CLASS_TYPE testing_Y[N_TESTING], DATA_TYPE min[N_FEATURES],
5                     DATA_TYPE max[N_FEATURES])
6  {
7      // Declare training and min max arrays
8
9      DATA_TYPE training_X_first[N_TRAINING/4][N_FEATURES];
10     DATA_TYPE training_X_second[N_TRAINING/4][N_FEATURES];
11     DATA_TYPE training_X_third[N_TRAINING/4][N_FEATURES];
12     DATA_TYPE training_X_fourth[N_TRAINING/4][N_FEATURES];
13
14     CLASS_TYPE training_Y_first[N_TRAINING/4];
15     CLASS_TYPE training_Y_second[N_TRAINING/4];
16     CLASS_TYPE training_Y_third[N_TRAINING/4];
17     CLASS_TYPE training_Y_fourth[N_TRAINING/4];
18
19     DATA_TYPE min_first[N_FEATURES];
20     DATA_TYPE min_second[N_FEATURES];
21     DATA_TYPE min_third[N_FEATURES];
22     DATA_TYPE min_fourth[N_FEATURES];
23
24     DATA_TYPE max_first[N_FEATURES];
25     DATA_TYPE max_second[N_FEATURES];
26     DATA_TYPE max_third[N_FEATURES];
27     DATA_TYPE max_fourth[N_FEATURES];
28
29     // Separate the training points array into four arrays
```

```
30
31      for (int i = 0; i < (N_TRAINING/4); i++)
32      {
33          training_Y_first[i]=training_Y[i];
34          for (int j = 0; j < N_FEATURES; j++)
35          {
36              training_X_first[i][j]=training_X[i][j];
37          }
38      }
39
40      for (int i = (N_TRAINING/4); i < (2*(N_TRAINING/4)); i++)
41      {
42          training_Y_second[(i-(N_TRAINING/4))]=training_Y[i];
43          for (int j = 0; j < N_FEATURES; j++)
44          {
45              training_X_second[(i-(N_TRAINING/4))][j]=training_X[i][j];
46          }
47      }
48
49      for (int i = (2*(N_TRAINING/4)); i < (3*(N_TRAINING/4)); i++)
50      {
51          training_Y_third[(i-(2*(N_TRAINING/4)))]=training_Y[i];
52          for (int j = 0; j < N_FEATURES; j++)
53          {
54              training_X_third[(i-(2*(N_TRAINING/4)))][j]=training_X[i][j];
55          }
56      }
57
58      for (int i = (3*(N_TRAINING/4)); i < N_TRAINING; i++)
59      {
60          training_Y_fourth[(i-(3*(N_TRAINING/4)))]=training_Y[i];
61          for (int j = 0; j < N_FEATURES; j++)
62          {
63              training_X_fourth[(i-(3*(N_TRAINING/4)))][j]=training_X[i][j];
64          }
65      }
66
67      //Duplicate the min max arrays
68
69      for (int i = 0; i < N_FEATURES; i++)
70      {
71          min_first[i]=min[i];
72          min_second[i]=min[i];
73          min_third[i]=min[i];
74          min_fourth[i]=min[i];
75
76          max_first[i]=max[i];
77          max_second[i]=max[i];
78          max_third[i]=max[i];
```

```
79          max_fourth[i]=max[i];
80      }
81
82      // Create arrays to store the best points
83
84      int bestPointsIdx_first[K];
85      int bestPointsIdx_second[K];
86      int bestPointsIdx_third[K];
87      int bestPointsIdx_fourth[K];
88      double bestDistances_first[K];
89      double bestDistances_second[K];
90      double bestDistances_third[K];
91      double bestDistances_fourth[K];
92      int bestPointsIdx[K];
93      double bestDistances[K];
94      int bestPointsArrayIdx[K];
95
96      CLASS_TYPE histogram[N_CLASSES] = {0};
97
98      static DATA_TYPE testing_X_first[N_FEATURES]={0};
99      static DATA_TYPE testing_X_second[N_FEATURES]={0};
100     static DATA_TYPE testing_X_third[N_FEATURES]={0};
101     static DATA_TYPE testing_X_fourth[N_FEATURES]={0};
102
103     for (int i = 0; i < N_TESTING; i++)
104     {
105         kNN_InitBest(bestDistances, bestPointsIdx);
106
107         kNN_MinMaxNormalize(min, max, testing_X[i]);
108
109         for (int k = 0; k < N_FEATURES; k++)
110         {
111             testing_X_first[k]=testing_X[i][k];
112             testing_X_second[k]=testing_X[i][k];
113             testing_X_third[k]=testing_X[i][k];
114             testing_X_fourth[k]=testing_X[i][k];
115         }
116
117         //Init best distance arrays
118
119         kNN_InitBest(bestDistances, bestPointsIdx);
120         kNN_InitBest(bestDistances_first, bestPointsIdx_first);
121         kNN_InitBest(bestDistances_second, bestPointsIdx_second);
122         kNN_InitBest(bestDistances_third, bestPointsIdx_third);
123         kNN_InitBest(bestDistances_fourth, bestPointsIdx_fourth);
124
125         //Calculate the best points for each partition
126
127         kNN_Partial_Predict(training_X_first, testing_X_first,
```

```
128                             min_first ,max_first ,bestPointsIdx_first ,
129                             bestDistances_first ,1084);
130         kNN_Partial_Predict(training_X_second , testing_X_second ,
131                             min_second ,max_second ,bestPointsIdx_second ,
132                             bestDistances_second ,1084);
133         kNN_Partial_Predict(training_X_third , testing_X_third ,
134                             min_third , max_third ,bestPointsIdx_third ,
135                             bestDistances_third ,1084);
136         kNN_Partial_Predict(training_X_fourth , testing_X_fourth ,
137                             min_fourth , max_fourth ,bestPointsIdx_fourth ,
138                             bestDistances_fourth ,1084);
139 
140         //Calculate the best points between all partitions
141 
142         for(int j=0;j<K;j++)
143         {
144             kNN_Partial_UpdateBestCaching(bestDistances_first[j],
145                                           bestPointsIdx_first[j],bestDistances ,
146                                           bestPointsIdx ,bestPointsArrayIdx ,1);
147         }
148 
149         for(int j=0;j<K;j++)
150         {
151             kNN_Partial_UpdateBestCaching(bestDistances_second[j],
152                                           bestPointsIdx_second[j],
153                                           bestDistances ,
154                                           bestPointsIdx ,bestPointsArrayIdx ,2);
155         }
156 
157         for(int j=0;j<K;j++)
158         {
159             kNN_Partial_UpdateBestCaching(bestDistances_third[j],
160                                           bestPointsIdx_third[j],bestDistances ,
161                                           bestPointsIdx ,bestPointsArrayIdx ,3);
162         }
163 
164         for(int j=0;j<K;j++)
165         {
166             kNN_Partial_UpdateBestCaching(bestDistances_fourth[j],
167                                           bestPointsIdx_fourth[j],
168                                           bestDistances ,
169                                           bestPointsIdx ,bestPointsArrayIdx ,4);
170         }
171 
172         //Vote the most represented class in best points
173 
174         kNN_Partial_VoteBetweenBest(bestPointsIdx , training_Y_first ,histogram ,
175                                     bestPointsArrayIdx ,1);
176 
```

```
177             kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y_second,
178                                          histogram,bestPointsArrayIdx,2);
179 
180             kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y_third,
181                                          histogram,bestPointsArrayIdx,3);
182 
183             testing_Y[i]=kNN_Partial_VoteBetweenBest(bestPointsIdx,
184                                                       training_Y_fourth,histogram,
185                                                       bestPointsArrayIdx,4);
186 
187             for(int i=0;i<N_CLASSES;i++)
188             {
189                 histogram[i]=0;
190             }
191         }
192 }
```

A.5 Python script for the local version

```
import sys
import os
import subprocess

# Check if at least one argument was provided
if len(sys.argv) == 1:
    print("Usage: local_arrays.py [Num of local partitions] [Name of the scenario]"
                                                )
    exit()
else:
    if sys.argv[1] == 0:
        print("Number of partitions cannot be 0")
        exit()
    else:
        n_part=int(sys.argv[1])
        print("Number of local partitions:"+str(n_part))

#clean previous files
current_directory = os.getcwd()
extension_to_remove = '.dat'

for root, dirs, files in os.walk(current_directory):
    for file in files:
        if file.endswith(extension_to_remove):
            file_path = os.path.join(root, file)
            os.remove(file_path)
            print(f"Removed: {file_path}")


#get the parent directory
parent_directory = os.path.dirname(os.getcwd())

#check if there is a scenario specified
if(len(sys.argv) > 2):
    scenario_str = str(sys.argv[2])
    #print("Scenario str:")
    #print(scenario_str)

# Construct the command to compile the C program
compilation_command_str = f"gcc dataset_norm.c dataconverter.c utils.c -o
                                          dataset_norm "

if(len(sys.argv) > 2):
    compilation_command_str=compilation_command_str + f"-D SCENARIO=" +
                                                scenario_str

print("compilation command str:")
```

```
print(compilation_command_str)

# Use subprocess.run() to execute the compilation command
gcc_result = subprocess.run(compilation_command_str, shell=True, text=True,
                                          capture_output=True,cwd=parent_directory)

# Check the result
if gcc_result.returncode == 0:
    print(f"Dataset normalization program compiled successfully.")
else:
    print("Compilation encountered an error.")
    print("Error Output:")
    print(gcc_result.stderr)

dataset_norm_program_str=f"dataset_norm"
print("Dataset normalization command str:")
print(dataset_norm_program_str)

# Use subprocess.run() to execute the program
data_norm_result = subprocess.run(dataset_norm_program_str, shell=True, text=True,
                                          capture_output=True,cwd=parent_directory)

# Check the result
if data_norm_result.returncode == 0:
    print(f"Dataset normalization program ran successfully.")
else:
    print("Compilation encountered an error.")
    print("Error Output:")
    print(data_norm_result.stderr)

#check if training dataset files exist

try:
    training_X_file = open("../training_X.dat", "r")
except FileExistsError:
    print("Ficheiro training_X.dat n o encontrado")

try:
    training_Y_file = open("../training_Y.dat", "r")
except FileExistsError:
    print("Ficheiro training_Y.dat n o encontrado")

training_X_lines = training_X_file.readlines()
training_Y_lines = training_Y_file.readlines()

training_Y=[]
training_X=[]
features=[]
temp_line=[]
```

```
min=[]
max=[]

for line in training_X_lines:
    temp_line=line.replace("{","")
    temp_line=temp_line.replace("}","")
    if(temp_line[len(temp_line)-2]==','):
        temp_line=temp_line[:len(temp_line)-2]
    else:
        temp_line=temp_line[:len(temp_line)-1]
    features=temp_line.split(",")
    training_X.append(features)

for line in training_Y_lines:
    temp_line=line.replace("{","")
    temp_line=temp_line.replace("}","")
    features=temp_line.split(",")
    training_Y.append(features)

training_X_file.close()
training_Y_file.close()

new_size=(len(training_X)//n_part)
remainder=(len(training_X)%n_part)

training_X_new_file_name="training_X_0.dat"
training_Y_new_file_name="training_Y_0.dat"

training_X_new_file_array=[]
training_Y_new_file_array=[]

#open array files

for i in range(n_part):
    f_x=open(training_X_new_file_name,"w")
    f_y=open(training_Y_new_file_name,"w")
    training_X_new_file_array.append(f_x)
    training_Y_new_file_array.append(f_y)
    print("New training_X file name:" + training_X_new_file_name)
    print("New training_Y file name:" + training_Y_new_file_name)
    training_X_new_file_name=training_X_new_file_name.replace(str(i),str(i+1))
    training_Y_new_file_name=training_Y_new_file_name.replace(str(i),str(i+1))

#write in training X array

for i in range(n_part):
    for j in range(new_size):
        training_X_new_file_array[i].write("{")
        for k in range(len(training_X[0])):
```

```
            training_X_new_file_array[i].write(training_X[(i*new_size)+j][k])
            if(k<(len(training_X[0])-1)):
                training_X_new_file_array[i].write(",")
        training_X_new_file_array[i].write("}")
        if(j<(new_size-1) or (remainder>0)):
            training_X_new_file_array[i].write(",")
        training_X_new_file_array[i].write("\n")

#if there are remainder write them in training X

if(remainder>0):
    for i in range(n_part):
        if(i<remainder):
            training_X_new_file_array[i].write("{")
            for k in range(len(training_X[0])):
                training_X_new_file_array[i].write(training_X[(n_part*new_size)+i][
                                                          k])
                if(k<(len(training_X[0])-1)):
                    training_X_new_file_array[i].write(",")
            training_X_new_file_array[i].write("}")
            training_X_new_file_array[i].write("\n")
        else:
            training_X_new_file_array[i].write("{")
            for k in range(len(training_X[0])):
                training_X_new_file_array[i].write("0.900000")
                if(k<(len(training_X[0])-1)):
                    training_X_new_file_array[i].write(",")
            training_X_new_file_array[i].write("}")
            training_X_new_file_array[i].write("\n")

#write in training Y array

for i in range(n_part):
    training_Y_new_file_array[i].write("{")
    for j in range(new_size):
        training_Y_new_file_array[i].write(training_Y[0][(i*new_size)+j])
        if(j<new_size-1 or (remainder>0)):
                training_Y_new_file_array[i].write(",")
    if(remainder==0):
        training_Y_new_file_array[i].write("}")
        training_Y_new_file_array[i].write("\n")

#if there are remainder write them in training Y

if(remainder>0):
    for i in range(n_part):
        if(i<remainder):
            training_Y_new_file_array[i].write(training_Y[0][(n_part*new_size)+i])
            training_Y_new_file_array[i].write("}")
```

```
            training_Y_new_file_array[i].write("\n")
        else:
            training_Y_new_file_array[i].write("0")
            training_Y_new_file_array[i].write("}")
            training_Y_new_file_array[i].write("\n")

#write partial dist dec file

partial_dist_dec_file=open("./partial_dist_dec.h", "w")

bestpointsstr="    int bestPointsIdx_0[K];\n"
bestdiststr="    double bestDistances_0[K];\n"

for i in range(n_part):
    partial_dist_dec_file.write(bestpointsstr)
    partial_dist_dec_file.write(bestdiststr)
    bestpointsstr=bestpointsstr.replace(str(i),str(i+1))
    bestdiststr=bestdiststr.replace(str(i),str(i+1))

#write partial dist init file

partial_dist_init_file=open("./partial_dist_init.h", "w")

initbeststr="    kNN_InitBest(bestDistances_0, bestPointsIdx_0);\n"

for i in range(n_part):
    partial_dist_init_file.write(initbeststr)
    initbeststr=initbeststr.replace(str(i),str(i+1),2)

#write partial predict file

partial_predict_file=open("./partial_predict.h", "w")

predictstr="    kNN_Partial_Predict(training_X_0, testing_X_0, min_0, max_0,
                                         bestPointsIdx_0,bestDistances_0,n_part);\
                                         n"
if(remainder==0):
    predictstr=predictstr.replace("n_part",str(new_size))
else:
    predictstr=predictstr.replace("n_part",str(new_size+1))

for i in range(n_part):
    partial_predict_file.write(predictstr)
    predictstr=predictstr.replace(str(i),str(i+1),6)

#write update best file

update_best_file=open("./update_best.h", "w")
```

```
update_best_str="""    for(int i=0;i<K;i++){

        kNN_Partial_UpdateBestCaching(bestDistances_0[i], bestPointsIdx_0[i],
                                  bestDistances, bestPointsIdx,bestPointsArrayIdx,0
                                                 );


    }\n\n"""

for i in range(n_part):
    update_best_file.write(update_best_str)
    if(i==0):
        update_best_str=(update_best_str.replace(str(i), str(i+1))).replace("1","0"
                                                 ,1)
    else:
        update_best_str=update_best_str.replace(str(i), str(i+1))

#write vote file

vote_file=open("./vote.h", "w")

vote_str="kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y_0,histogram,
                                          bestPointsArrayIdx,0);\n"

vote_last_str="testing_Y[i]=kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y_0
                                          ,histogram,bestPointsArrayIdx,0);\n"

for i in range(n_part-1):
    vote_file.write(vote_str)
    vote_str=vote_str.replace(str(i), str(i+1),2)

vote_last_str=vote_last_str.replace("0",str(n_part-1),2)
vote_file.write(vote_last_str)

#write array dec file

array_dec_file=open("../array_dec.h", "w")

training_X_str="""static DATA_TYPE training_X_0[N_TRAINING][N_FEATURES]={
    #include "./localarrays/training_X_0.dat"
};\n\n"""

if(remainder==0):
    training_X_str=training_X_str.replace("N_TRAINING",str(new_size))
else:
    training_X_str=training_X_str.replace("N_TRAINING",str(new_size+1))

training_Y_str="""static CLASS_TYPE training_Y_0[N_TRAINING]=
    #include "./localarrays/training_Y_0.dat"
;\n\n"""
```

```
if(remainder==0):
    training_Y_str=training_Y_str.replace("N_TRAINING",str(new_size))
else:
    training_Y_str=training_Y_str.replace("N_TRAINING",str(new_size+1))

min_str="""static DATA_TYPE min_0[N_FEATURES]=
    #include"min.dat"
;\n\n"""

max_str="""static DATA_TYPE max_0[N_FEATURES]=
    #include"max.dat"
;\n\n"""

for i in range(n_part):
    array_dec_file.write(training_X_str)
    array_dec_file.write(training_Y_str)
    array_dec_file.write(min_str)
    array_dec_file.write(max_str)
    training_X_str=training_X_str.replace(str(i),str(i+1),1)
    training_Y_str=training_Y_str.replace(str(i),str(i+1),1)
    training_X_str=training_X_str.replace((str(i)+".dat"),(str(i+1)+".dat"))
    training_Y_str=training_Y_str.replace((str(i)+".dat"),(str(i+1)+".dat"))
    min_str=min_str.replace(str(i),str(i+1))
    max_str=max_str.replace(str(i),str(i+1))

#declare local arrays dec file

local_arrays_dec_file=open("./local_arrays_dec.h", "w")

local_arrays_dec_str="static DATA_TYPE testing_X_0[N_FEATURES]={0};\n"

for i in range(n_part):
    local_arrays_dec_file.write(local_arrays_dec_str)
    local_arrays_dec_str=local_arrays_dec_str.replace(str(i),str(i+1),1)

local_arrays_dec_file.write("\n\n\n")

#initiate local arrays file

local_arrays_init_file=open("./local_arrays_init.h", "w")

local_arrays_copy_str="testing_X_0[k]=testing_X[i][k];\n"

local_arrays_init_file.write("for (int k = 0; k < N_FEATURES; k++){\n")
local_arrays_init_file.write("    //#pragma HLS UNROLL\n")

for i in range(n_part):
    local_arrays_init_file.write(local_arrays_copy_str)
```

```
    local_arrays_copy_str=local_arrays_copy_str.replace(str(i),str(i+1),1)
local_arrays_init_file.write("}\n")
```

A.6 Python script for the external version

```
import sys
import os
import subprocess

# Check if at least one argument was provided
if len(sys.argv) == 1:
    print("Usage: external_arrays.py [Num of external partitions] [Name of the
                                                scenario]")
    exit()
else:
    if sys.argv[1] == 0:
        print("Number of partitions cannot be 0")
        exit()
    else:
        n_part=int(sys.argv[1])
        print("Number of external partitions:"+str(n_part))

#clean previous files
current_directory = os.getcwd()
extension_to_remove = '.dat'

for root, dirs, files in os.walk(current_directory):
    for file in files:
        if file.endswith(extension_to_remove):
            file_path = os.path.join(root, file)
            os.remove(file_path)
            print(f"Removed: {file_path}")

#get the parent directory
parent_directory = os.path.dirname(os.getcwd())

#check if there is a scenario specified
if(len(sys.argv) > 2):
    scenario_str = str(sys.argv[2])
    #print("Scenario str:")
    #print(scenario_str)

# Construct the command to compile the C program
compilation_command_str = f"gcc dataset_norm.c dataconverter.c utils.c -o
                                        dataset_norm "

if(len(sys.argv) > 2):
    compilation_command_str=compilation_command_str + f"-D SCENARIO=" +
                                                scenario_str

print("compilation command str:")
print(compilation_command_str)
```

```
# Use subprocess.run() to execute the compilation command
gcc_result = subprocess.run(compilation_command_str, shell=True, text=True,
                                          capture_output=True,cwd=parent_directory)

# Check the result
if gcc_result.returncode == 0:
    print(f"Dataset normalization program compiled successfully.")
else:
    print("Compilation encountered an error.")
    print("Error Output:")
    print(gcc_result.stderr)

dataset_norm_program_str=f"dataset_norm"
print("Dataset normalization command str:")
print(dataset_norm_program_str)

# Use subprocess.run() to execute the program
data_norm_result = subprocess.run(dataset_norm_program_str, shell=True, text=True,
                                          capture_output=True,cwd=parent_directory)

# Check the result
if data_norm_result.returncode == 0:
    print(f"Dataset normalization program ran successfully.")
else:
    print("Compilation encountered an error.")
    print("Error Output:")
    print(data_norm_result.stderr)

try:
    training_X_file = open("../training_X.dat", "r")
except FileExistsError:
    print("Ficheiro training_X.dat n o encontrado")

try:
    training_Y_file = open("../training_Y.dat", "r")
except FileExistsError:
    print("Ficheiro training_Y.dat n o encontrado")

training_X_lines = training_X_file.readlines()
training_Y_lines = training_Y_file.readlines()

training_Y=[]
training_X=[]
features=[]
temp_line=[]
min=[]
max=[]
```

```
for line in training_X_lines:
    temp_line=line.replace("{","")
    temp_line=temp_line.replace("}","")
    if(temp_line[len(temp_line)-2]==',' ):
        temp_line=temp_line[:len(temp_line)-2]
    else:
        temp_line=temp_line[:len(temp_line)-1]
    features=temp_line.split(",")
    training_X.append(features)

for line in training_Y_lines:
    temp_line=line.replace("{","")
    temp_line=temp_line.replace("}","")
    features=temp_line.split(",")
    training_Y.append(features)

training_X_file.close()
training_Y_file.close()

new_size=(len(training_X)//n_part)
remainder=(len(training_X)%n_part)

training_X_new_file_name="training_X_0.dat"
training_Y_new_file_name="training_Y_0.dat"

training_X_new_file_array=[]
training_Y_new_file_array=[]

#open array files

for i in range(n_part):
    f_x=open(training_X_new_file_name,"w")
    f_y=open(training_Y_new_file_name,"w")
    training_X_new_file_array.append(f_x)
    training_Y_new_file_array.append(f_y)
    print("New training_X file name:" + training_X_new_file_name)
    print("New training_Y file name:" + training_Y_new_file_name)
    training_X_new_file_name=training_X_new_file_name.replace(str(i),str(i+1))
    training_Y_new_file_name=training_Y_new_file_name.replace(str(i),str(i+1))

#write in training X array

for i in range(n_part):
    for j in range(new_size):
        training_X_new_file_array[i].write("{")
        for k in range(len(training_X[0])):
            training_X_new_file_array[i].write(training_X[(i*new_size)+j][k])
            if(k<(len(training_X[0])-1)):
                training_X_new_file_array[i].write(",")
```

```
        training_X_new_file_array[i].write("}")
        if(j<(new_size-1) or (remainder>0)):
            training_X_new_file_array[i].write(",")
        training_X_new_file_array[i].write("\n")

#if there are remainder write them in training X

if(remainder>0):
    for i in range(n_part):
        if(i<remainder):
            training_X_new_file_array[i].write("{")
            for k in range(len(training_X[0])):
                training_X_new_file_array[i].write(training_X[(n_part*new_size)+i][
                                                            k])
                if(k<(len(training_X[0])-1)):
                    training_X_new_file_array[i].write(",")
            training_X_new_file_array[i].write("}")
            training_X_new_file_array[i].write("\n")
        else:
            training_X_new_file_array[i].write("{")
            for k in range(len(training_X[0])):
                training_X_new_file_array[i].write("0.900000")
                if(k<(len(training_X[0])-1)):
                    training_X_new_file_array[i].write(",")
            training_X_new_file_array[i].write("}")
            training_X_new_file_array[i].write("\n")

#write in training Y array

for i in range(n_part):
    training_Y_new_file_array[i].write("{")
    for j in range(new_size):
        training_Y_new_file_array[i].write(training_Y[0][(i*new_size)+j])
        if(j<new_size-1 or (remainder>0)):
                training_Y_new_file_array[i].write(",")
    if(remainder==0):
        training_Y_new_file_array[i].write("}")
        training_Y_new_file_array[i].write("\n")

#if there are remainder write them in training Y

if(remainder>0):
    for i in range(n_part):
        if(i<remainder):
            training_Y_new_file_array[i].write(training_Y[0][(n_part*new_size)+i])
            training_Y_new_file_array[i].write("}")
            training_Y_new_file_array[i].write("\n")
        else:
            training_Y_new_file_array[i].write("0")
```

```
            training_Y_new_file_array[i].write("}")
            training_Y_new_file_array[i].write("\n")

#create array dec file

array_dec_file=open("../array_dec.txt", "w")

training_X_str="""static DATA_TYPE training_X_0[N_TRAINING][N_FEATURES]={
    #include "./external_arrays/training_X_0.dat"
};\n\n"""

if(remainder==0):
    training_X_str=training_X_str.replace("N_TRAINING",str(new_size))
else:
    training_X_str=training_X_str.replace("N_TRAINING",str(new_size+1))

training_Y_str="""static CLASS_TYPE training_Y_0[N_TRAINING]=
    #include "./external_arrays/training_Y_0.dat"
;\n\n"""

if(remainder==0):
    training_Y_str=training_Y_str.replace("N_TRAINING",str(new_size))
else:
    training_Y_str=training_Y_str.replace("N_TRAINING",str(new_size+1))

min_str="""static DATA_TYPE min_0[N_FEATURES]=
    #include"./min.dat"
;\n\n"""

max_str="""static DATA_TYPE max_0[N_FEATURES]=
    #include"./max.dat"
;\n\n"""

for i in range(n_part):
    array_dec_file.write(training_X_str)
    array_dec_file.write(training_Y_str)
    array_dec_file.write(min_str)
    array_dec_file.write(max_str)
    training_X_str=training_X_str.replace(str(i),str(i+1))
    training_Y_str=training_Y_str.replace(str(i),str(i+1))
    min_str=min_str.replace(str(i),str(i+1))
    max_str=max_str.replace(str(i),str(i+1))

#create knn_predictAll call (in main function)

knn_call_file=open("./knn_call.txt", "w")

training_X_str="training_X_0"
```

```
training_Y_str="training_Y_0"

min_str="min_0"

max_str="max_0"

knn_call_file.write("kNN_PredictAll(")

for i in range(n_part):
    knn_call_file.write(training_X_str)
    knn_call_file.write(",")
    training_X_str=training_X_str.replace(str(i),str(i+1))

for i in range(n_part):
    knn_call_file.write(training_Y_str)
    knn_call_file.write(",")
    training_Y_str=training_Y_str.replace(str(i),str(i+1))

knn_call_file.write("testing_X, predicted_testing_Y,")

for i in range(n_part):
    knn_call_file.write(min_str)
    knn_call_file.write(",")
    min_str=min_str.replace(str(i),str(i+1))

for i in range(n_part-1):
    knn_call_file.write(max_str)
    knn_call_file.write(",")
    max_str=max_str.replace(str(i),str(i+1))
knn_call_file.write(max_str)
knn_call_file.write(");\n")

#create knn_predictAll declaration (in knn.h and knn.c)

knn_dec_file=open("./knn_dec.txt", "w")

training_X_str="DATA_TYPE training_X_0[N_TRAINING][N_FEATURES]"

training_Y_str="CLASS_TYPE training_Y_0[N_TRAINING]"

min_str="DATA_TYPE min_0[N_FEATURES]"

max_str="DATA_TYPE max_0[N_FEATURES]"

knn_dec_file.write("void kNN_PredictAll(")

for i in range(n_part):
    knn_dec_file.write(training_X_str)
    knn_dec_file.write(",\n")
```

```
    training_X_str=training_X_str.replace(str(i),str(i+1))

for i in range(n_part):
    knn_dec_file.write(training_Y_str)
    knn_dec_file.write(",\n")
    training_Y_str=training_Y_str.replace(str(i),str(i+1))

knn_dec_file.write("DATA_TYPE testing_X[N_TESTING][N_FEATURES],\nCLASS_TYPE
                                          testing_Y[N_TESTING],\n")

for i in range(n_part):
    knn_dec_file.write(min_str)
    knn_dec_file.write(",\n")
    min_str=min_str.replace(str(i),str(i+1))

for i in range(n_part-1):
    knn_dec_file.write(max_str)
    knn_dec_file.write(",\n")
    max_str=max_str.replace(str(i),str(i+1))
knn_dec_file.write(max_str)
knn_dec_file.write(")\n")

###From here, most of the code was directly copied from the code_local version of
                                          this file

#write partial dist dec file

partial_dist_file=open("./partial_dist_dec.txt", "w")

bestpointsstr="    int bestPointsIdx_0[K];\n"
bestdiststr="    double bestDistances_0[K];\n"

for i in range(n_part):
    partial_dist_file.write(bestpointsstr)
    partial_dist_file.write(bestdiststr)
    bestpointsstr=bestpointsstr.replace(str(i),str(i+1))
    bestdiststr=bestdiststr.replace(str(i),str(i+1))

#write partial dist init file

partial_dist_file=open("./partial_dist_init.txt", "w")

initbeststr="    kNN_InitBest(bestDistances_0, bestPointsIdx_0);\n"

for i in range(n_part):
    partial_dist_file.write(initbeststr)
    initbeststr=initbeststr.replace(str(i),str(i+1),2)

#write partial predict file
```

```
partial_predict_file=open("./partial_predict.txt", "w")

predictstr="    kNN_Partial_Predict(training_X_0, testing_X_0, min_0, max_0,
                                          bestPointsIdx_0,bestDistances_0,n_part);\
                                          n"
if(remainder==0):
    predictstr=predictstr.replace("n_part",str(new_size))
else:
    predictstr=predictstr.replace("n_part",str(new_size+1))

for i in range(n_part):
    partial_predict_file.write(predictstr)
    predictstr=predictstr.replace(str(i),str(i+1),6)

#write update best file

update_best_file=open("./update_best.txt", "w")

update_best_str="""    for(int i=0;i<K;i++){

        kNN_Partial_UpdateBestCaching(bestDistances_0[i], bestPointsIdx_0[i],
                                  bestDistances, bestPointsIdx,bestPointsArrayIdx,0
                                                 );

    }\n\n"""

for i in range(n_part):
    update_best_file.write(update_best_str)
    if(i==0):
        update_best_str=(update_best_str.replace(str(i), str(i+1))).replace("1","0"
                                                 ,1)
    else:
        update_best_str=update_best_str.replace(str(i), str(i+1))

#write vote file

vote_file=open("./vote.txt", "w")

vote_str="kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y_0,histogram,
                                          bestPointsArrayIdx,0);\n"

vote_last_str="testing_Y[i]=kNN_Partial_VoteBetweenBest(bestPointsIdx, training_Y_0
                                          ,histogram,bestPointsArrayIdx,0);\n"

for i in range(n_part-1):
    vote_file.write(vote_str)
    vote_str=vote_str.replace(str(i), str(i+1),2)
```

```
vote_last_str=vote_last_str.replace("0",str(n_part-1),2)
vote_file.write(vote_last_str)

#write local arrays dec file

local_arrays_dec_file=open("./local_arrays_dec.txt", "w")

local_arrays_dec_str="static DATA_TYPE testing_X_0[N_FEATURES]={0};\n"

for i in range(n_part):
    local_arrays_dec_file.write(local_arrays_dec_str)
    local_arrays_dec_str=local_arrays_dec_str.replace(str(i),str(i+1),1)

local_arrays_dec_file.write("\n\n\n")

#write local arrays init file

local_arrays_init_file=open("./local_arrays_init.txt", "w")

local_arrays_copy_str="testing_X_0[k]=testing_X[i][k];\n"

local_arrays_init_file.write("for (int k = 0; k < N_FEATURES; k++){\n")
local_arrays_init_file.write("    #pragma HLS UNROLL\n")

for i in range(n_part):
    local_arrays_init_file.write(local_arrays_copy_str)
    local_arrays_copy_str=local_arrays_copy_str.replace(str(i),str(i+1),1)
local_arrays_init_file.write("}\n")
```

A.7 Python script for the multi stage version

```
import subprocess
import sys
import os

# Check if at least one argument was provided
if len(sys.argv) < 3:
    print("Usage: local_arrays.py [Num of local partitions] [Num of external
                                              partitions] [Name of the scenario]")
    exit()
else:
    if sys.argv[1] == 0 or sys.argv[2] == 0:
        print("Number of partitions cannot be 0")
        exit()
    else:
        n_part=int(sys.argv[1])
        n_part_local=int(sys.argv[1])
        n_part_external=int(sys.argv[2])
```

```
        print("Number of local partitions:"+str(n_part_local))
        print("Number of external partitions:"+str(n_part_external))

#clean previous files
current_directory = os.getcwd()
extension_to_remove = '.dat'

for root, dirs, files in os.walk(current_directory):
    for file in files:
        if file.endswith(extension_to_remove):
            file_path = os.path.join(root, file)
            os.remove(file_path)
            print(f"Removed: {file_path}")

#get the parent directory
parent_directory = os.path.dirname(os.getcwd())

#check if there is a scenario specified
if(len(sys.argv) > 3):
    scenario_str = str(sys.argv[3])
    #print("Scenario str:")
    #print(scenario_str)

# Construct the command to compile the C program
compilation_command_str = f"gcc dataset_norm.c dataconverter.c utils.c -o
                                          dataset_norm "

if(len(sys.argv) > 3):
    compilation_command_str=compilation_command_str + f"-D SCENARIO=" +
                                              scenario_str

print("compilation command str:")
print(compilation_command_str)

# Use subprocess.run() to execute the compilation command
gcc_result = subprocess.run(compilation_command_str, shell=True, text=True,
                                          capture_output=True,cwd=parent_directory)

# Check the result
if gcc_result.returncode == 0:
    print(f"Dataset normalization program compiled successfully.")
else:
    print("Compilation encountered an error.")
    print("Error Output:")
    print(gcc_result.stderr)

dataset_norm_program_str=f"dataset_norm"
print("Dataset normalization command str:")
print(dataset_norm_program_str)
```

```
# Use subprocess.run() to execute the program
data_norm_result = subprocess.run(dataset_norm_program_str, shell=True, text=True,
                                          capture_output=True,cwd=parent_directory)

# Check the result
if data_norm_result.returncode == 0:
    print(f"Dataset normalization program ran successfully.")
else:
    print("Compilation encountered an error.")
    print("Error Output:")
    print(data_norm_result.stderr)

try:
    training_X_file = open("../training_X.dat", "r")
except FileExistsError:
    print("Ficheiro training_X.dat n o encontrado")

try:
    training_Y_file = open("../training_Y.dat", "r")
except FileExistsError:
    print("Ficheiro training_Y.dat n o encontrado")

training_X_lines = training_X_file.readlines()
training_Y_lines = training_Y_file.readlines()

training_Y=[]
training_X=[]
features=[]
temp_line=[]
min=[]
max=[]

for line in training_X_lines:
    temp_line=line.replace("{","")
    temp_line=temp_line.replace("}","")
    if(temp_line[len(temp_line)-2]==',') :
        temp_line=temp_line[:len(temp_line)-2]
    else:
        temp_line=temp_line[:len(temp_line)-1]
    features=temp_line.split(",")
    training_X.append(features)

for line in training_Y_lines:
    temp_line=line.replace("{","")
    temp_line=temp_line.replace("}","")
    features=temp_line.split(",")
    training_Y.append(features)
```

```
training_X_file.close()
training_Y_file.close()

n_part_total=n_part_local*n_part_external
new_size=((len(training_X)//n_part_total))
remainder=(len(training_X)%n_part_total)

#write partial dist dec file

partial_dist_file=open("./partial_dist_dec.h", "w")

bestpointsstr="    int bestPointsIdx_0[K];\n"
bestdiststr="    double bestDistances_0[K];\n"

for i in range(n_part_local):
    partial_dist_file.write(bestpointsstr)
    partial_dist_file.write(bestdiststr)
    bestpointsstr=bestpointsstr.replace(str(i),str(i+1))
    bestdiststr=bestdiststr.replace(str(i),str(i+1))

#write local arrays dec file

local_arrays_dec_file=open("./local_arrays_dec.h", "w")

local_arrays_dec_str="static DATA_TYPE testing_X_0[N_FEATURES]={0};\n"

for i in range(n_part_local):
    local_arrays_dec_file.write(local_arrays_dec_str)
    local_arrays_dec_str=local_arrays_dec_str.replace(str(i),str(i+1),1)

local_arrays_dec_file.write("\n\n\n")

#write partial dist init file

partial_dist_file=open("./partial_dist_init.h", "w")

initbeststr="    kNN_InitBest(bestDistances_0, bestPointsIdx_0);\n"

for i in range(n_part_local):
    partial_dist_file.write(initbeststr)
    initbeststr=initbeststr.replace(str(i),str(i+1),2)

#write local arrays init file

local_arrays_init_file=open("./local_arrays_init.h", "w")

local_arrays_copy_str="testing_X_0[k]=testing_X[i][k];\n"

local_arrays_init_file.write("for (int k = 0; k < N_FEATURES; k++){\n")
```

```
local_arrays_init_file.write("    #pragma HLS UNROLL\n")

for i in range(n_part_local):
    local_arrays_init_file.write(local_arrays_copy_str)
    local_arrays_copy_str=local_arrays_copy_str.replace(str(i),str(i+1),1)
local_arrays_init_file.write("}\n")

#write partial predict file

partial_predict_file=open("./partial_predict.h", "w")

predictstr="    kNN_Partial_Predict(training_X_0, testing_X_0,bestPointsIdx_0,
                                                bestDistances_0,n_part);\n"
if(remainder==0):
    predictstr=predictstr.replace("n_part",str(new_size))
else:
    predictstr=predictstr.replace("n_part",str(new_size+1))

for i in range(n_part_local):
    partial_predict_file.write(predictstr)
    predictstr=predictstr.replace(str(i),str(i+1),4)

#write update best file

update_best_file=open("./update_best.h", "w")

update_best_str="""    for(int j=0;j<K;j++){

        kNN_Partial_UpdateBestCaching_Class(bestDistances_0[j], training_Y_0[
                                                        bestPointsIdx_0[j]],
                                bestDistances[i], bestDistancesclass[i]);

    }\n\n"""

for i in range(n_part_local):
    update_best_file.write(update_best_str)
    if(i==0):
        update_best_str=(update_best_str.replace(str(i), str(i+1))).replace("1","0"
                                                        ,1)
    else:
        update_best_str=update_best_str.replace(str(i), str(i+1))

#create knn_predictAll declaration (in knn.h and knn.c)

knn_dec_file=open("./knn_dec.h", "w")

training_X_str="DATA_TYPE training_X_0[N_TRAINING][N_FEATURES]"

training_Y_str="CLASS_TYPE training_Y_0[N_TRAINING]"
```

```
knn_dec_file.write("void kNN_PredictAll(")

for i in range(n_part_local):
    knn_dec_file.write(training_X_str)
    knn_dec_file.write(",\n")
    training_X_str=training_X_str.replace(str(i),str(i+1))

for i in range(n_part_local):
    knn_dec_file.write(training_Y_str)
    knn_dec_file.write(",\n")
    training_Y_str=training_Y_str.replace(str(i),str(i+1))

knn_dec_file.write("DATA_TYPE testing_X[N_TESTING][N_FEATURES],\nCLASS_TYPE
                                          testing_Y[N_TESTING],\ndouble
                                          bestDistances[N_TESTING][K],\nCLASS_TYPE
                                          bestDistancesclass[N_TESTING][K])")

#Create multistage file

knn_multistage_file=open("./knn_multistage_code.h", "w")

local_arrays_X_dec_str="static DATA_TYPE training_X_0[N_TRAINING][N_FEATURES];\n"
local_arrays_Y_dec_str="static CLASS_TYPE training_Y_0[N_TRAINING];\n"

if(remainder==0):
    local_arrays_X_dec_str=local_arrays_X_dec_str.replace("N_TRAINING",str(new_size
                                              ))
    local_arrays_Y_dec_str=local_arrays_Y_dec_str.replace("N_TRAINING",str(new_size
                                              ))
else:
    local_arrays_X_dec_str=local_arrays_X_dec_str.replace("N_TRAINING",str(new_size
                                              +1))
    local_arrays_Y_dec_str=local_arrays_Y_dec_str.replace("N_TRAINING",str(new_size
                                              +1))

for i in range(n_part_local):
    knn_multistage_file.write(local_arrays_X_dec_str)
    knn_multistage_file.write(local_arrays_Y_dec_str)
    local_arrays_X_dec_str=local_arrays_X_dec_str.replace(str(i),str(i+1),1)
    local_arrays_Y_dec_str=local_arrays_Y_dec_str.replace(str(i),str(i+1),1)

knn_multistage_file.write("double bestDistances[N_TESTING][K];\nint bestPointsIdx[
                                          N_TESTING][K];\nCLASS_TYPE
                                          bestDistancesclass[N_TESTING][K];\n\n")

knn_multistage_file.write("    for(int i=0;i<N_TESTING;i++){\nkNN_InitBest(
                                          bestDistances[i],bestPointsIdx[i]);\
```

```
                                        nkNN_MinMaxNormalize(min, max, testing_X[
                                        i]);\n}\n\n\n\n\n\n")

copy_str="    for(int i=0;i<N_TRAINING;i++){\nfor(int j=0;j<N_FEATURES;j++){\
                                        ntraining_X_0[i][j]=training_X[i][j];\n}\
                                        ntraining_Y_0[i]=training_Y[i];\n}\n\n\n"

copy_str_part=" for(int j=0;j<N_FEATURES;j++){\ntraining_X_0[new_size][j]=1;\n}\
                                        ntraining_Y_0[new_size]=training_Y[0];\n\
                                        n\n"

copy_str_part=copy_str_part.replace("new_size",str(new_size))

copy_str_part_new="    for(int i=0;i<N_TRAINING;i++){\nfor(int j=0;j<N_FEATURES;j
                                        ++){\ntraining_X_0[i][j]=training_X[i][j
                                        ];\nif(i==new_size_part) training_X_0[i][
                                        j]=0.9;\n}\ntraining_Y_0[i]=training_Y[i
                                        ];\nif(i==new_size_part) training_Y_0[i]=
                                        training_Y[0];\n}\n\n\n"

copy_str_part_new=copy_str_part_new.replace("new_size_part",str(new_size))

if(remainder==0):
    copy_str=copy_str.replace("N_TRAINING",str(new_size))
else:
    copy_str=copy_str.replace("N_TRAINING",str(new_size+1))

if(remainder==0):
    copy_str_part_new=copy_str_part_new.replace("N_TRAINING",str(new_size))
else:
    copy_str_part_new=copy_str_part_new.replace("N_TRAINING",str(new_size+1))

copy_str_original=copy_str
copy_str_part_original=copy_str_part
copy_str_part_new_original=copy_str_part_new

training_X_0_str="training_X_0,"
training_Y_0_str="training_Y_0,"

part_count=0

part_count_int=0
part_count_part=0

for i in range(n_part_external):
    for j in range(n_part_local):
        if(remainder==0):
            copy_str=copy_str.replace("training_X[i]","training_X[i+"+str(
                                                    part_count_int*new_size)+"]")
```

```
        copy_str=copy_str.replace("training_Y[i]","training_Y[i+"+str(
                                            part_count_int*new_size)+"]")
    else:
        if(part_count_part==0):
            copy_str=copy_str.replace("training_X[i]","training_X[i+"+str((
                                            part_count_int*(new_size+
                                            1))+(part_count_part*(
                                            new_size)))+"]")
            copy_str=copy_str.replace("training_Y[i]","training_Y[i+"+str((
                                            part_count_int*(new_size+
                                            1))+(part_count_part*(
                                            new_size)))+"]")
            copy_str_part_new=copy_str_part_new.replace("training_X[i]","
                                            training_X[i+"+str((
                                            part_count_int*(new_size+
                                            1))+(part_count_part*(
                                            new_size)))+"]")
            copy_str_part_new=copy_str_part_new.replace("training_Y[i]","
                                            training_Y[i+"+str((
                                            part_count_int*(new_size+
                                            1))+(part_count_part*(
                                            new_size)))+"]")
        else:
            copy_str=copy_str.replace("training_X[i]","training_X[i+"+str((
                                            part_count_int*(new_size+
                                            1))+(part_count_part*(
                                            new_size))+1)+"]")
            copy_str=copy_str.replace("training_Y[i]","training_Y[i+"+str((
                                            part_count_int*(new_size+
                                            1))+(part_count_part*(
                                            new_size))+1)+"]")
            copy_str_part_new=copy_str_part_new.replace("training_X[i]","
                                            training_X[i+"+str((
                                            part_count_int*(new_size+
                                            1))+(part_count_part*(
                                            new_size))+1)+"]")
            copy_str_part_new=copy_str_part_new.replace("training_Y[i]","
                                            training_Y[i+"+str((
                                            part_count_int*(new_size+
                                            1))+(part_count_part*(
                                            new_size))+1)+"]")
    if(remainder==0):
        knn_multistage_file.write(copy_str)
    else:
        if(part_count_part==0):
            knn_multistage_file.write(copy_str)
        else:
            knn_multistage_file.write(copy_str_part_new)
    if(remainder==0):
```

```
        copy_str=copy_str.replace("training_X[i+"+str(part_count_int*new_size)+
                                             "]","training_X[i]")
        copy_str=copy_str.replace("training_Y[i+"+str(part_count_int*new_size)+
                                             "]","training_Y[i]")
    else:
        if(part_count_part==0):
            copy_str=copy_str.replace("training_X[i+"+str((part_count_int*(
                                                new_size+1))+(
                                                part_count_part*(new_size
                                                )))+"]","training_X[i]")
            copy_str=copy_str.replace("training_Y[i+"+str((part_count_int*(
                                                new_size+1))+(
                                                part_count_part*(new_size
                                                )))+"]","training_Y[i]")
            copy_str_part_new=copy_str_part_new.replace("training_X[i+"+str((
                                                part_count_int*(new_size+
                                                1))+(part_count_part*(
                                                new_size)))+"]","
                                                training_X[i]")
            copy_str_part_new=copy_str_part_new.replace("training_Y[i+"+str((
                                                part_count_int*(new_size+
                                                1))+(part_count_part*(
                                                new_size)))+"]","
                                                training_Y[i]")
        else:
            copy_str=copy_str.replace("training_X[i+"+str((part_count_int*(
                                                new_size+1))+(
                                                part_count_part*(new_size
                                                ))+1)+"]","training_X[i]"
                                                )
            copy_str=copy_str.replace("training_Y[i+"+str((part_count_int*(
                                                new_size+1))+(
                                                part_count_part*(new_size
                                                ))+1)+"]","training_Y[i]"
                                                )
            copy_str_part_new=copy_str_part_new.replace("training_X[i+"+str((
                                                part_count_int*(new_size+
                                                1))+(part_count_part*(
                                                new_size))+1)+"]","
                                                training_X[i]")
            copy_str_part_new=copy_str_part_new.replace("training_Y[i+"+str((
                                                part_count_int*(new_size+
                                                1))+(part_count_part*(
                                                new_size))+1)+"]","
                                                training_Y[i]")
    if(remainder==0):
        part_count_int=part_count_int+1
    else:
        if(part_count_int+1<remainder):
```

```
                part_count_int=part_count_int+1
            else:
                part_count_part=part_count_part+1
        #if(part_count_part>1):
        #    knn_multistage_file.write(copy_str_part)
        copy_str=copy_str.replace(str(j)+"[",str(j+1)+"[")
        #copy_str_part=copy_str_part.replace(str(j)+"[",str(j+1)+"[")
        copy_str_part_new=copy_str_part_new.replace(str(j)+"[",str(j+1)+"[")
    knn_multistage_file.write("kNN_PredictAll(")
    for j in range(n_part_local):
        knn_multistage_file.write(training_X_0_str)
        training_X_0_str=training_X_0_str.replace(str(j),str(j+1))
    for j in range(n_part_local):
        knn_multistage_file.write(training_Y_0_str)
        training_Y_0_str=training_Y_0_str.replace(str(j),str(j+1))
    knn_multistage_file.write("testing_X, testing_Y,bestDistances,
                                         bestDistancesclass);\n")
    training_X_0_str="training_X_0,"
    training_Y_0_str="training_Y_0,"
    copy_str=copy_str_original
    copy_str_part=copy_str_part_original
    copy_str_part_new=copy_str_part_new_original
```