PROJECT ON HETEROGENEOUS COMPUTING

COURSE: EFFICIENT HETEROGENEOUS COMPUTING, M.EIC033
MASTER IN INFORMATICS AND COMPUTING ENGINEERING (M.EIC)
FACULTY OF ENGINEERING OF THE UNIVERSITY OF PORTO (FEUP)

October 2024

This document describes the project for the "Efficient Heterogeneous Computing" course of the Master in Informatics and Computing Engineering (M.EIC), of the Faculty of Engineering of the University of Porto (FEUP).

THE PROGRAM

The k-Nearest Neighbors (*kNN*) [1] is a widely used machine learning technique for classification and regression[1]. For classification, and using the basic version of *kNN*, the training corresponds to the storage of the *n* training instances, each one represented as a vector of d features (dimensions), with each vector labeled by the respective class. For each instance to classify, *kNN* determines the *k* nearest instances, i.e., the *k* training instances nearest the instance and then classifies based on the classes of those *k* instances. One possibility is to classify with the class most frequent in the k closest training instances.

The code of the project implements *kNN* in C targeting the classification of ***m*** instances. All instances consist of vectors of ***d*** features represented in single- or double-precision floating point numbers (float or double types in C). Training considers ***c*** classes, and the Euclidean distance is used for the distance between the instance to classify and each of the training instances.

The configuration of a *kNN* implementation is defined by the tuple (*m, k, n, d, c, t*), where ***m*** represents the number of instances to classify, ***k*** the number of nearest neighbors to consider, ***n*** the number of training instances (stored in the *kNN* database), ***d*** is the number of features, ***c*** is the number of classes, and ***t*** the data type used for the data and distance calculations.

The computational complexity of the basic *kNN* is $O(nd + kn)$.

GOALS

The goal is to accelerate the execution of the function *knn_classifyinstance* in two environments: (1) a PC/desktop; (2) an embedded system with a hardware accelerator. For parallelization targeting a multicore CPU, the intention is to use the OpenMP directive-driven programming model. For optimizations regarding the generation of the hardware accelerator, the intention is to use the optimization directives provided by the specific FPGA compilers.

The implementation of this function and of its auxiliary functions are in the file `knn.c`. However, changes do not need to be limited to the function *knn_classifyinstance*.

The scenarios to be used for the experiments in the desktop and in the embedded device with the hardware accelerator are related to A, B and C. In all of them, data are embedded in the code of the program (i.e., data structures are initialized by static data provided in the code), and the program does not receive any argument from command line. The input data for scenario A1a, A1b, and A1c is from WISDM[2] v1.1 [2], a dataset of Human Activity

[1] A brief description of the kNN technique is presented in https://en.wikipedia.org/wiki/K-neare st neighbors algorithm.

[2] This data has been released by the Wireless Sensor Data Mining (WISDM) Lab: http://www.cis.fordham.edu/wisdm/

Recognition (HAR), with data from sensors in mobile devices and collected by different users, while for scenarios A2 and A3 the input data are synthetic. These two, A2 and A3, scenarios are just used to acquire profiling info when the number of training and testing instances, and the number of features are larger. They can be also used to evaluate the impact of optimizations and of the final implementations.

In those scenarios, the *kNN* implementations to be improved are based on the following parameters:

Scenarios	**Parameters**	**M**	**k**	**n**	**d**	**C**	**T**
	Identification in the code	num_new_points	K	num_points	num_features	num_classes	DATA_TYPE
A1a	Generic implementation (values are for testing)	Variable input: size for tests = 1082	variable (use 3 for testing)	variable (use 4336 for testing)	variable (use 43 for testing)	variable (use 6 for testing)	float or double
A1b	Generic implementation (values are for testing)	Variable input: size for tests = 1082	variable (use 20 for testing)	variable (use 4336 for testing)	variable (use 43 for testing)	variable (use 6 for testing)	float or double
A1c	Specialized implementation (fixed values)	Variable input: size for tests = 1082	3	4336	43	6	float
A2	Generic implementation (values are for testing)	Variable input: size for tests = 1996	20	8004	100	8	float or double
A3	Generic implementation (values are for testing)	Variable input: size for tests = 9998	20	40002	100	8	float or double

The 6 classes (human activities) of the WISDM dataset are represented here using the following numerical identifiers (IDs):

Class: Upstairs ID = 3
Class: Jogging ID = 0
Class: Sitting ID = 2
Class: Standing ID = 1
Class: Walking ID = 4
Class: Downstairs ID = 5

The folders:

- scenario-simple (test case with data with 2 features, and a small number of training and testing instances)
- scenario-wisdm
- scenario-gen100x8x10000
- scenario-gen100x8x50000

provide the correct files:

- params.h // default DT = 1 (double), K=20
- test.dat
- train.dat

for each scenario:

- A1a, A1b, and A1c: scenario-wisdm
- A2: scenario-gen100x8x10000
- A3: scenario-gen100x8x50000

After including the required files for each scenario, the program can be compiled using (note that these examples do not include optimization flags):

A1a (-D DT=2 to compile considering float data types)

```
gcc -Wall -std=gnu99 -lm -c utils.c timer.c io.c
gcc -D K=3 -Wall -std=gnu99 -lm -c knn.c
gcc -D K=3 -Wall -std=gnu99 -lm main.c utils.o timer.o knn.o io.o -o knn
```

A1b (-D DT=2 to compile considering float data types)

```
gcc  -Wall -std=gnu99 -lm -c utils.c timer.c io.c
gcc -D K=20 -Wall -std=gnu99 -lm -c knn.c
gcc -D K=20 -Wall -std=gnu99 -lm main.c utils.o timer.o knn.o io.o -o knn
```

A1c (-D DT=2 to compile considering float data types)

```
gcc -Wall -std=gnu99 -lm -c utils.c timer.c io.c
gcc -D K=3 -Wall -std=gnu99 -lm -c knn.c
gcc -D K=3 -Wall -std=gnu99 -lm main.c utils.o timer.o knn.o io.o -o knn
```

A2: (-D DT=2 to compile considering float data types)

```
gcc -Wall -std=gnu99 -lm -c utils.c timer.c io.c
gcc -Wall -D K=20 -D NUM_TRAINING_SAMPLES=8004 -D NUM_TESTING_SAMPLES=1996 -D NUM_FEATURES=100
-D NUM_CLASSES=8 -std=gnu99 -lm -c knn.c
gcc -Wall -D K=20 -D NUM_TRAINING_SAMPLES=8004 -D NUM_TESTING_SAMPLES=1996 -D NUM_FEATURES=100
-D NUM_CLASSES=8 -std=gnu99 -lm main.c utils.o timer.o knn.o io.o -o knn
```

A3:

```
gcc -Wall -std=gnu99 -lm -c utils.c timer.c io.c
gcc -Wall -D K=20 -D NUM_TRAINING_SAMPLES=40002 -D NUM_TESTING_SAMPLES=9998 -D
NUM_FEATURES=100 -D NUM_CLASSES=8 -std=gnu99 -lm -c knn.c
gcc -Wall -D K=20 -D NUM_TRAINING_SAMPLES=40002 -D NUM_TESTING_SAMPLES=9998 -D
NUM_FEATURES=100 -D NUM_CLASSES=8 -std=gnu99 -lm main.c utils.o timer.o knn.o io.o -o knn
```

Example

Below are the results of executing the code using the wisdom dataset, k=20, and double floating-point precision.

```
> .\knn
Array data points initialized with data in files ...
Executing kNN...
class id: 3
...
class id: 5
class id: 2
class id: 3
class id: 3
class id: 3
class id: 2
class id: 3
kNN done.

kNN: number of classes = 6
kNN: number of training instances = 4336
```

```
kNN: number of features = 43
kNN: k = 20
kNN: data type used = double
kNN: number of classified instances = 1082
kNN: number of classifications wrong = 338
kNN: number of classifications right = 744
kNN: accuracy = 68.76 %


Time:  1.2916 s
```

At the end of execution, the program reports whether the produced outputs are correct or not and the accuracy[3] achieved in the classification of the input instances. It also prints the computation time to classify the points in the input testing data.

Version	**K**	**Data type**	**Accuracy**
A1	K=3	DT=1 (double)	68.58%
A1	K=20	DT=1 (double)	68.76%
A1	K=3	DT=2 (float)	68.58%
A1	K=20	DT=2 (float)	68.76%

NOTE: Scenarios B and C are not to be used in the project.

STEPS TO BE FOLLOWED AND REPORTED:

A. Analysis and improvements targeting Intel/AMD multicore CPUs in a PC/desktop/laptop machine.

1) Analyze via profiling (*gprof* and *gcov*) and via Intel VTune and Intel Advisor the program given. For example, identify the hotpots, draw the call graph and the task graph of the program, and decorate them with the information you find useful for helping to improve the execution time and the selection of the code regions to be possibly migrated to a hardware accelerator.
2) Considering its execution on a PC/desktop/laptop machine try to improve its overall performance (it can also include parallelization via OpenMP) and use Intel VTune and Intel Advisor to help you (e.g., verify the positions in the Roofline model of the subsequent versions).
3) Characterize the software code versions in the local machine. Use the *gcc* compiler and selected flags and show the energy consumption, power dissipation, and execution time gains that you could obtain with the different versions over the original code provided.

B. Analysis and improvements targeting a SoC with a multicore ARM CPU and hardware accelerators using FPGAs.

4) Analyze the code and decide about the computations to offload to the target FPGA. Characterize and optimize the select code for offloading (use Vitis HLS).
5) Evaluate the hardware/software system and the overall speedup obtained (i.e., CPU + accelerator vs. CPU).

NOTES REGARDING THE WORK AND/OR THE TECHNICAL REPORT:

[3] We note that the use case used has not involved feature selection to improve the kNN accuracy.

The most important experiments, e.g., involving compiler optimizations and/or code transformations, and the respective results shall be reported even if they do not improve execution time, power dissipation, and/or energy consumption.

In addition, the report shall describe:

- The experimental setup (e.g., the machine used where the experiments were conducted) shall be described in detail;
- The results achieved and an analysis of those results;
- Possible code transformations and optimizations that you think can have a significant impact but did not have the time to evaluate.

REFERENCES

[1] T. Cover and P. Hart, "Nearest neighbor pattern classification," in IEEE Transactions on Information Theory, vol. 13, no. 1, pp. 21-27, January 1967, doi: https://doi.org/10.1109/TIT.1967.1053964.

[2] Jennifer R. Kwapisz, Gary M. Weiss, and Samuel A. Moore. 2011. Activity recognition using cell phone accelerometers. SIGKDD Explor. Newsl. 12, 2 (December 2010), 74–82. doi: https://doi.org/10.1145/1964897.1964918.