Lab 2: Digit Recognition System (Part 1)
Due Friday, September 29, 2023, 11:59pm
Late submission: 4% penalty per day; cannot be late by more than 6 days

1 Introduction

Handwriting recognition refers to the computer's ability to intelligently interpret handwritten inputs and is broadly applied in document processing, signature verification, and bank check clearing. An important step in handwriting recognition is classification, which classifies data into one of a fixed number of classes. In this lab, you will design and implement a handwritten digit recognition system based on the **k-nearest-neighbors (k-NN)** classifier algorithm [1], and eventually implement it on an FPGA device.

In the first part of this lab (this particular assignment), you are provided with a set of already classified handwritten digits (called the training sets), and will implement a k-NN algorithm in C++ that is able to identify any input handwritten digit (called the testing instance) using the training sets. In addition, you will use two commonly-used high-level synthesis (HLS) optimizations to parallelize the synthesized hardware and explore design trade-offs in performance and area. Later in the second part of this lab (Lab 3), you will further optimize the performance of your hardware design and implement the complete digital recognition system on the Xilinx Zynq field-programmable system-on-chip.

2 Materials

You are given a zip file named *lab2.zip* on *ecelinux* under */classes/ece6775/labs*. It contains the following files for you to build the project.

- *digitrec.cpp*: an **incomplete source** file where you write your k-NN based digit recognition algorithm in C++.
- *digitrec.h*: the header file that defines the interface for the core functions `update_knn` and `knn_vote`.
- *typedefs.h*: the header file that defines the key data types used in the design.
- *data/training_set_#.dat*: training set for digit #, where # = 0, 1, 2, ..., 9.
- *training_data.h*: the header file that combines all the training data sets (i.e., `data/testing_set.dat`) into a constant array.
- *data/testing_set.dat*: a set of testing instances with corresponding expected values to help you test your design.
- *digitrec_test.cpp*: a test bench (only useful for simulation) that helps verify your code and perform experiments with various handwritten input digits.
- *Makefile*: a makefile for you to easily compile the code into an executable named `digitrec_k-nn_tb` and execute the program to check results (enter `make digitrec-sw`).
- *run.tcl*: the template project Tcl script that allows you to run Vivado HLS synthesis in command line (`vivado_hls -f run.tcl` or `make digitrec-hw`). For this assignment, it is sufficient to run the tool with this simple command.
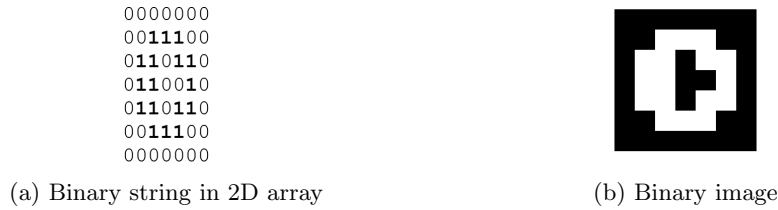
```
0000000
0011100
0110110
0110010
0110110
0011100
0000000
```

(a) Binary string in 2D array

(b) Binary image

Figure 3.1: Training instance for digit 0

```
0000000
0001000
0011100
0000100
0001000
0001000
0010000
```

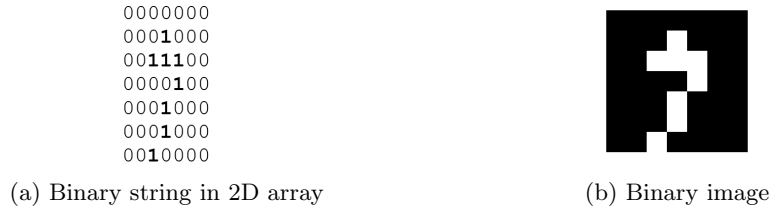(a) Binary string in 2D array

(b) Binary image

Figure 3.2: Training instance for digit 7

Before starting your assignment, please copy and unzip the zip file to your home directory. Please be sure to source the class setup script using the following command before compiling your source code: `source /classes/ece6775/setup-ece6775.sh`

3 Design Overview

You are given 10 training sets, each of which contains 1800 49-bit training instances for a different digit (0-9). Each hexadecimal string in $training\_set\_\#$ represents a 49-bit training instance for digit $\#$. The 49 bits of each instance encodes a 7x7 matrix of binary values (i.e., a bitmap). For example, $e3664d8e00_{16}$ in $training\_set\_0$ is a training instance for digit 0 and translates into the binary 2D matrix in Figure 3.1a, whose 1 bits outline the digit 0. A binary image of the array is shown in Figure 3.1b. $41c0820410_{16}$ in $training\_set\_7$ is a training instance for digit 7 and translates into the binary 2D matrix in Figure 3.2a, whose 1 bits approximately outline the digit 7. The corresponding binary image is shown in Figure 3.2b. As you can see, the resolution of the digit is limited by the number of bits (49 bits in our assignment) used to represent it. Typically, increasing the number of bits per instance would improve the resolution and possibly the accuracy of recognition.

We would like to devise an algorithm that takes in a binary string representing a handwritten digit (i.e. the testing instance) and classify it to a particular digit (0-9) by first identifying k training instances that are closest to the testing instance (i.e., the nearest neighbors), and then determining the result based on the most common digit represented by these nearest neighbors. Even values of k can sometimes lead to ties and in these cases, they can be resolved arbitrarily, e.g., by choosing the first digit in the list.

You are encouraged to read through [1] to familiarize yourself with the basic concepts of the k-NN algorithm. **In this assignment, we define the distance between two instances as the number of corresponding bits that are different in the two binary strings, i.e., the Hamming distance.** For example, 1011_2 and 0111_2 differ in the two most significant bits and therefore have a distance of 2. 1011_2 and 1010_2 differ only in the least significant bit and have a distance of 1. As a result, 1011_2 is closer to 1010_2 than to 0111_2.

4 Guidelines and Hints

4.1 Coding and Debugging

Your first task is to complete the digit recognition algorithm based on the code skeleton provided in *digitrec.cpp*. In particular, you are expected to fill in the code for the following functions:

- *update_knn*: Given the testing instance and a (new) training instance, this function maintains/updates an array of k minimum distances per training set.

- *knn_vote*: Among $10 \times k$ minimum distance values, this function finds the k nearest neighbors and determines the final output based on the most common digit represented by these nearest neighbors.

Note that the skeleton code takes advantage of arbitrary precision integer type *ap_uint*. **A useful reference of arbitrary precision integer data types can be found on p.538–558 of the user guide [2] (please pay special attention to the bitwise and bit selection operations).**

How you choose to implement the algorithm may affect the resulting accuracy of your design as reported by the test bench. **We expect that your design would achieve an error of less than 10% on the provided testing set.** You may use the console output or the generated output files, e.g. `digitrec_3-nn_sw_result.txt`, to debug your code.

4.2 Design Exploration

The second part of the assignment is to explore the impact of the k value on your digit recognition design. Specifically, **you are expected to experiment with the k values ranging from 1 through 3, and collect the performance and area numbers of the synthesized design for each specific k.**

- The actual k value can be specified in *Makefile* for software testing or in *run.tcl* for synthesis. Similar to the CORDIC assignment, you can run simulation and synthesis in batch with *run.tcl*. This script will also automatically collect important stats (i.e., accuracy, performance, and resource usage) from the Vivado HLS reports and generate a *knn_result.csv* file under the *result* folder.

- In this assignment, you will use a fixed $10ns$ clock period targeting a specific Xilinx Zynq FPGA device. Clock period and target device have been specified in the *run* Tcl script.

4.3 Design Optimization

The third part of the assignment is to optimize the design with HLS pragmas or directives. In particular, we will focus on exploring the effect of the following optimizations in our design and apply them appropriately to **minimize the latency of the synthesized design (Hint: the final optimized latency should be roughly one order of magnitude lower than the baseline you obtained from experimentation described in Section 4.2).**

- **loop unrolling** unfolds a loop by creating multiple copies of its loop body and reducing its trip count accordingly. This technique is often used to achieve shorter latency when loop iterations can be executed in parallel.

- **array partitioning** partitions an array into smaller arrays, which may result in an RTL with multiple small memories or multiple registers instead of one large memory. This effectively increases the amount of read and write ports for the storage.

Please refer to the following user guide for details on how to apply these optimization using Vivado HLS (v2019.1).

- Vivado Design Suite User Guide, High-Level Synthesis, UG902 (v2019.1) [2]
  - *set_directive_unroll* p.461
  - *set_directive_array_partition* p.429

  You may insert pragmas or set directives to apply these optimizations. You can find code snippets with inserted pragmas throughout the user guide (e.g. p.142, p.134).

Other than the added pragmas/directives, your program should look the same with baseline.

In this experiment, **please avoid unrolling the outermost loop** (i.e., the one that iterate 1800 times) so your design would not require excessive chip area. Also for the sake of simplicity, please try to **only use fixed-bound** *for* **loop(s) in your program.** Note that *while* loops are synthesizable but may lead to a variable-latency design that would complicate your reporting. [1]

4.4 Report

- Please write your report in a **single-column and single-space format with a 10pt font size. Page limit is ONE, including necessary figures and tables.**

- The report should start with an overview of the document. This should inform the reader what the report is about, and highlight the major results. In other words, this is similar to an abstract in a technical document.

- There should be a section comparing different k values with a table that summarizes the key stats including the error rate (accuracy), area in terms of resource utilization (number of BRAMs, DSP48s, LUTs, and FFs), and and performance in latency in number of clock cycles. You can find the information from the synthesis report under `knn.prj/solution1/syn/report`.

- There should be a section describing how you would add the HLS pragmas/directives to minimize the latency of the synthesized design. Please contrast the performance and area of your most optimized design (i.e., the one with the smallest latency) with the baseline design. **For this comparison, please set k to 3.**

- The report should only show screenshots from the tool when they demonstrate some significant idea. If you do use screenshots, make sure they are readable (e.g., not blurry). In general, you are expected to create your own figures. While more time consuming, it allows you to show the exact results, figures, and ideas you wish to present.

5 Deliverables

Please submit your assignment on CMS. You are expected to submit your report and your code and scripts (and only these files, not the project files generated by the tool) in a zipped file named **digitrec.zip** that contains the following contents:

- *report.pdf*: the project report in pdf.

- A folder named *solution*: the set of source files and scripts required to reproduce your experiments (including the pragams/directives for design optimization). Note that only these files should be submitted. Please run *make clean* to remove all the automatically generated output files.

6 Acknowledgement

This design originated from an ECE-6775 project originally implemented by Ackerley Tng and Edgar Munoz.

References

[1] Kun, Jeremy. *K-Nearest-Neighbors and Handwritten Digit Classification*. Math Programming. Available at `https://jeremykun.com/2012/08/26/k-nearest-neighbors-and-handwritten-digit-classification`

---

[1]You will need to enable C-RTL co-simulation to get the actual cycle count for a design with data-dependent loop bounds.

[2] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis UG902 (v2019.1).* Available at `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf`