ECE5775 High-Level Digital Design Automation, Fall 2017
School of Electrical Computer Engineering, Cornell University

Lab 2: Digit Recognition System (Part 1)

Due Wednesday, September 27, 11:59pm

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 AWS FPGA instance.

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 AWS FPGA instance.

2 Materials

You are given a zip file named *lab2.zip* on *ecelinux* under */classes/ece5775/labs*. It contains two directories: **lab2** and **harness**.

- **lab2**: contains all the codes and data of this lab.
- **harness**: contains the wrapper code of OpenCL APIs and top-level makefile. Students are **not required** to understand the content of this directory.

The structure of directory **lab2** is described as following:

- **src**:
  - **host**: contains all the codes for host.
    - `check_result.cpp`: contains function to print out the errors and calculate the overall error rate.
    - `check_result.h`: the header file of check_result.cpp.
    - `digit_recognition.cpp`: the main file, defines the implementation flow of this application
    - `testing_data.h`: declares two constant arrays including the testing data and corresponding expected labels.
    - `training_data.h`: combines all the training data sets (i.e., data/testing set.dat) into a constant array.
    - `typedefs.h`: the constant definitions and typedefs for the host.
    - `utils.cpp`: contains the functions to print usage and parse the command line arguments
    - `utils.h`: the header file of

  - **ocl**: contains the actual kernel file.
    - `digitrec.cpp`: **an incomplete** file that defines the kernel function Digitrec.
- **data**:
  - `training_set_#.dat`: training set for digit #, where # $= 0, 1, 2, ..., 9$.
  - `testing_set.dat`: a set of testing instances with corresponding expected labels.
  - `expected.dat`: the expected labels of the testing set.
- **makefile**: the makefile to compile the this application.
- **run1.sh**: this script runs the emulation with k value set to 3 by default.
- **run5.sh**: this script runs the application for 5 times with the k value of KNN from 1 to 5.

Before starting your assignment, please copy and unzip the zip file to the correct director on your AWS instance, e.g. ∼ */src/project_data/* .

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.

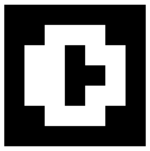
We would like to devise an algorithm that takes in a binary string representing a handwritten digit (i.e. the testing instance) and classify it to a particular digit (0-9) by first identifying k training instances that are closest to the testing instance (i.e., the nearest neighbors), and then determining the result based on the most common digit represented by these nearest neighbors.

In the latter part of the lab, we would like to research the influence of different choices of k values, and try to analyze the impact and tradeoff of the design choices.

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.** 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.

```
0000000
0011100
0110110
0110010
0110110
0011100
0000000
```

(a) Binary string in 2D array

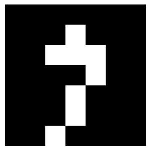


(b) Binary image

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

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 *src/ocl/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.620–638 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 *out.dat* file to debug your code.

Be aware that a bash script *run1.sh* is provided for you to run the software emulation, the k value is set to 3 by default, and you can specify the k value in the script. You can run the script with *./run1.sh*

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 5, and collect the performance and area numbers of the synthesized design for each specific k.**

The actual k value can be specified in *run5.sh*. You can run the software emulation in batch with *./run5.sh*. This script will also automatically collect the 5 estimation reports and combine them in a single file under *results* folder. You are going to look into the stats to analyze the impact of different k values.

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 (v2016.2).

- Vivado Design Suite User Guide, High-Level Synthesis, UG902 (v2016.2) [2]
  - *set_directive_unroll* p.481
  - *set_directive_array_partition* p.447

  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.146, p.151).

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

(not updated yet, will be updated after the previous parts are settled)

- Please write your report in a **single-column and single-space format with a 10pt font size. Page limit is 2. Please include your name and NetID on your report.**
- 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. Likewise there should be a summary, describing the results, and highlight the important points.
- There should be a section describing how you implement the *update_knn* and *knn_vote* functions.
- 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.[2]
- 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., with smallest latency) with the baseline design. For this comparison, you only need to set *k* to 3.
- Showing small snippets of your code to show the use of pragmas or to demonstrate the restructuring of the code is fine, but please avoid listing every single line of the code every optimization. It is possible to be succinct and thorough at the same time.
- 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.

[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]Check out the synthesis report under *knn.prj/solution1/syn/report/*.

6 Acknowledgement

This design originated from an ECE-5775 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

[2] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis UG902 (v2016.1)*, Available at http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf