Laboratory Exercise 4

Classification of Handwritten Digits using Machine Learning: Linear Classifier

In this exercise you will learn about a machine learning approach to classifying handwritten digits. You will create software and hardware implementations of a linear classifier that accepts images of handwritten digits ranging 0 to 9 and determines which digit is present in each image.

The MNIST Handwritten Digits Database

In this exercise you will use the MNIST handwritten digits database, which is widely used by scientists around the world to train and test handwritten digit recognition machines. The database provides 28x28-pixel 8-bit grayscale images, each containing a single handwritten digit ranging 0 to 9. An example of an image from the MNIST database is shown in Figure 1.

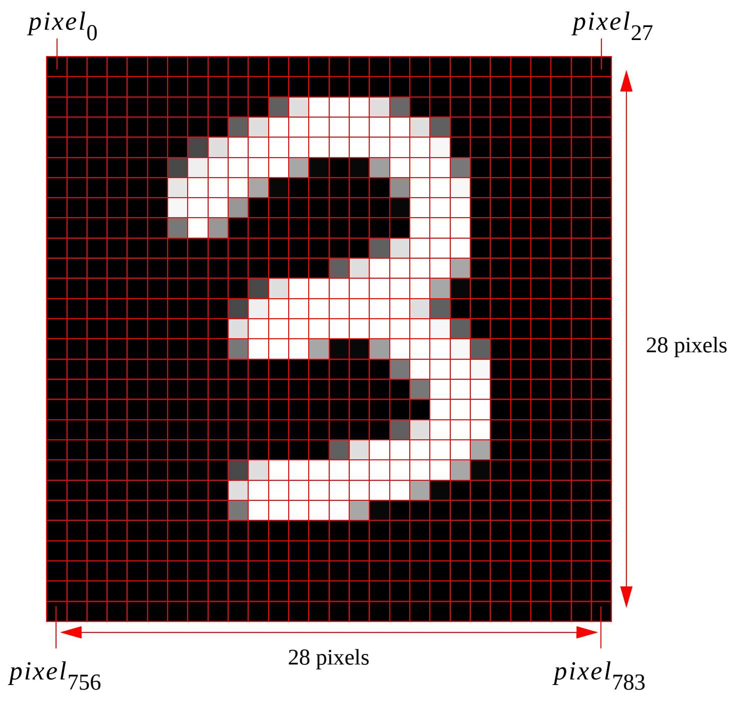


Figure 1: An example handwritten digit image from the MNIST database

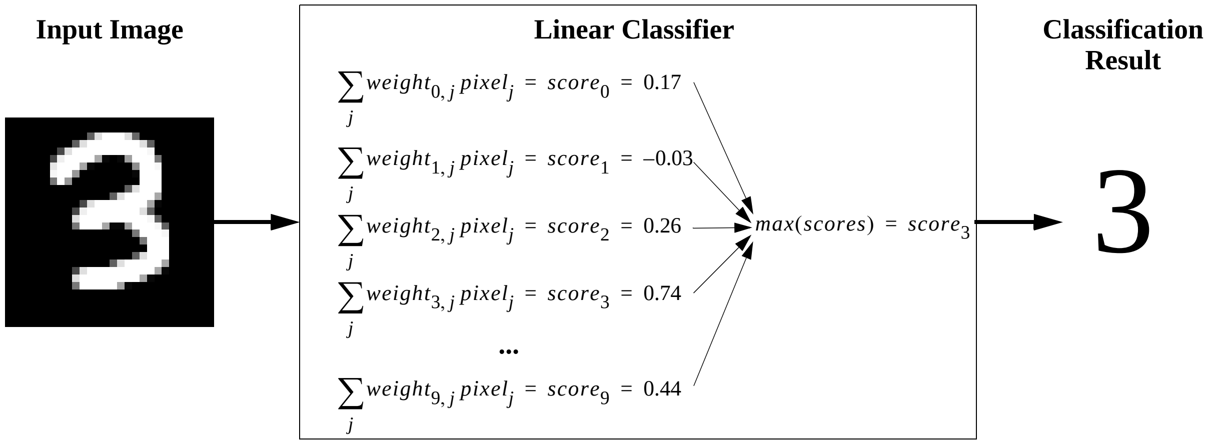
The database contains a training set of 60000 images which are used to train the classifier, and a test set of 10000 images which are used to test the classifier's accuracy. The 60000 training images have already been used to train the weights for the linear classifier. Your task is to use the provided weights to implement the linear classifier and test its classification accuracy on the 10000 test images.

The Linear Classifier

$$score = \sum_j weight_j \, pixel_j$$

Figure 2: The equation for calculating the score of a class in the linear classifier

The linear classifier evaluates the likelihood (or score) of the input image being each of the possible digits. The score is calculated using the equation shown in Figure 2, where j ranges 0 to 783 to span the pixels and their corresponding weights. Each digit has its own set of weights leading to a total of 10*784 = 7840 weights for the classifier. After calculating the scores of all the digits, the digit whose score is highest is chosen as the classification result. An example of the calculations is shown in Figure 3, where the linear classifier receives an input image containing the digit 3 and outputs the correct classification result.



Input Image
Linear Classifier
Classification Result
$\sum_j weight_{0,j} pixel_j = score_0 = 0.17$
$\sum_j weight_{1,j} pixel_j = score_1 = -0.03$
$\sum_j weight_{2,j} pixel_j = score_2 = 0.26$
$\sum_j weight_{3,j} pixel_j = score_3 = 0.74$
...
$\sum_j weight_{9,j} pixel_j = score_9 = 0.44$
$max(scores) = score_3$
3

Figure 3: The linear classifier operating on an input image containing the digit "3" and yielding the correct result

Part I

Implement a program in the C++ language that performs linear classification of MNIST handwritten digits. Your program must use the trained weights stored in files *weights_0*, *weights_1*, ... , *weights_9*, in the */design_files/weights_fp/* directory. Test your program by classifying the 10000 MNIST test images stored in the file */design_files/t10k-images-idx3-ubyte*. Compare the results of your classifier to the correct labels stored in the file */design_files/t10k-labels-idx1-ubyte* to calculate your classifier's accuracy as:

$$Accuracy = (\# \; of \; correct \; results)/10000 * 100\%$$

Compile and execute your program on a DE-Series Linux platform. Upon completion, your program should output the classification accuracy and the runtime in milliseconds.

Parsing the MNIST Database Files

The MNIST database is provided in .idx files. The file formats of the MNIST images and labels files are shown in Figure 4. Your program must open and parse these files to extract the image pixels and labels. Each pixel is 8 bits and represents a grayscale value. Each label is also 8 bits, and its value can range from 0 to 9 to indicate the correct classification of the corresponding image. The 32-bit integers stored in these files are stored in most-significant byte (MSB)-first format, which must first be converted to least significant byte (LSB)-first format before being used on the ARM processors of the DE-series boards.

```
TEST SET IMAGE FILE (t10k-images-idx3-ubyte):
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  10000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel

TEST SET LABEL FILE (t10k-labels-idx1-ubyte):
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  10000            number of items
0008     unsigned byte   ??               label
0009     unsigned byte   ??               label
........
xxxx     unsigned byte   ??               label
The labels values are 0 to 9.
```

Figure 4: File formats of the MNIST files. Source: http://yann.lecun.com/exdb/mnist/

Parsing the Weights Files

Trained weights for use in this part of the exercise are provided in the directory */design_files/weights_fp/*. There are 10 weights files named *weights_0*, *weights_1*, ... , *weights_9* that correspond to their respective digits. Each file contains 784 32-bit floating point weights that can be parsed using the code shown in Figure 5.

```c
#define FEATURE_COUNT 784

// weights must be an array of sufficient size (>= 784)
bool read_weights_file(char *filename, float *weights) {
    FILE *f = fopen(filename, "rb");
    if (f == NULL){
        printf("ERROR: could not open %s\n",filename);
        return false;
    }
    int read_elements = fread(weights, sizeof(float), FEATURE_COUNT, f);
    fclose(f);

    if (read_elements != FEATURE_COUNT){
        printf("ERROR: read incorrect number of weights from %s\n", filename);
        return false;
    }
    return true;
}
```

Figure 5: Function that parses a weights file

Part II

Implement an OpenCL kernel that performs linear classification of handwritten digits. Use the function prototype shown in Figure 6. The kernel reads the input array `images` which contains 10000 images. Each image in the array is 784 chars long, for a total of 10000 * 784 = 7840000 chars in the array. The kernel reads the input array `weights` which contains 10 sets of 784 weights, for a total of 7840 floats in the array. The kernel writes the classification results to the output array `results` which will contain 10000 chars when the kernel finishes. Each char in the `results` array will contain a value from 0 to 9 indicating the classification result.

```
__attribute__((reqd_work_group_size(10000,1,1)))
__kernel void linear_classifier(global const unsigned char * restrict images,
                                constant float * restrict weights,
                                global unsigned char * restrict results)
{
    ... Your Code ...
}
```

Figure 6: Function prototype for the linear classifier kernel

In previous exercises you manually created local memories to cache frequently used values and reduce memory access overhead. In this exercise, you will use Intel FPGA SDK for OpenCL's automatic cache generation feature to cache the linear classifier's weights. This is done by assigning the `constant` keyword to the weights array which tells the compiler that the values in the array are constant and should be cached to improve performance. By default the Intel FPGA SDK for OpenCL creates a cache that is 16384 bytes large to store the values. The cache size can be changed when compiling your kernel using the argument `-const-cache-bytes=<N>`, where <N> is the number of bytes desired. Select a cache size that is sufficiently large to minimize cache misses.

Maximize the performance of your design by unrolling your loop(s) until you run out of FPGA resources. Modify the SW implementation from Part I to invoke your OpenCL kernel. Ensure that your OpenCL kernel produces the same classification accuracy as your software implementation in Part I. Determine the runtime of your OpenCL kernel. How does its performance compare to that of the software implementation?

Part III

Open the "Area Analysis of System" page in the compilation report of your design in Part II. By examining the detailed breakdowns, you will see that the majority of resource utilization comes from implementing floating-point arithmetic operations. How many ALUTs, FFs, and DSPs does a floating-point add require? How many ALUTs, FFs, and DSPs does a floating-point multiply require? How many floating-point add operations could you fit into your FPGA, assuming you are implementing nothing else? How many floating-point multiply operations could you fit into your FPGA, assuming you are implementing nothing else? Note that you can get the total number of available ALUTs, FFs, and DSPs on the "Summary" page of the report.

Part IV

In this part of the exercise, you will modify your implementation from Part II to use fixed-point arithmetic instead of floating-point arithmetic.

Fixed-Point Arithmetic

The fixed-point datatype stores real numbers using a fixed number of digits after the decimal point. Fixed-point data consists of an arbitrary number of bits which are divided into two sets. The set of most significant bits represents the whole number to the left of the decimal point, while the set of least significant bits represents the fractional portion to the right of the decimal point. As an example, a 16-bit fixed-point number can be divided into a 12-bit set and an 4-bit set, where the most significant 12 bits represent the whole number, and the least significant 4 bits represent the fractional number. Such a fixed-point representation is called "12.4 fixed-point format". Each bit contributes 2^x to the real number value, where x is the bit's position offset from the decimal point. Figure 7 shows an example of a 16-bit value which is interpreted as a 12.4 fixed-point value and as a 8.8 fixed-point value.

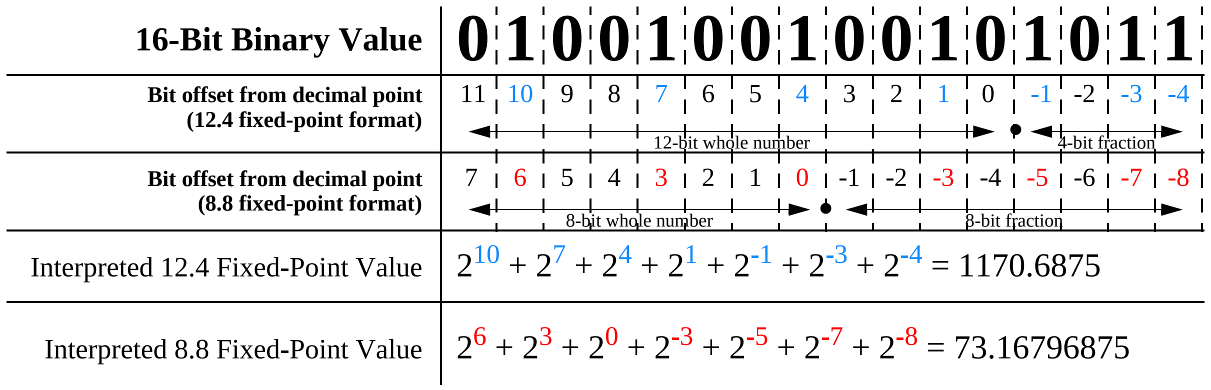


Figure 7: Interpreting a 16-bit value as 12.4 and 8.8 fixed-point values

An advantage of using fixed-point arithmetic is that the operations require fewer hardware resources to implement compared to floating-point arithmetic. This means that a fixed-point circuit can be smaller and more power-efficient than its floating-point equivalent. Alternatively, you could achieve higher performance by implementing more fixed-point units with the same hardware resources.

Fixed-point multiplication can be computed using an integer multiplier circuit, with the caveat that the circuit does not track the decimal point locations of the multiplicands and the product. It is therefore the designer's responsibility to track the decimal point locations. The product's decimal point location (offset from the LSB) is determined by adding the offsets in the multiplicands. For example, if you multiply a 12.4 fixed-point number by a 8.8 fixed-point number, the decimal point in the product will be offset from the LSB by 4 + 8 = 12 bits. In other words, the product would have a 12-bit fraction.

Fixed-point addition can be computed using an integer adder circuit, provided that two summands have the same decimal point location. When adding two fixed-point values whose decimal point locations differ, one of the summands must be shifted to match the other prior to addition. The resulting sum will have the same decimal point location as its summands.

Modifying Your Kernel to Use Fixed-Point Arithmetic

Modify your host program to parse the fixed-point weights in the directory */design_files/weights_fxp/*. These fixed-point weights are stored in 16.16 fixed-point format. Modify your kernel to accept the fixed-point weights as an `int` array, as shown in Figure 8. When multiplying the 16.16 fixed-point weights by the 8-bit pixels, the pixels can be considered to be 8.0 fixed-point numbers.

```
__attribute__((reqd_work_group_size(10000,1,1)))
__kernel void linear_classifier(global const unsigned char * restrict images,
                                constant int * restrict weights,
                                global unsigned char * restrict results)
{
    ... Your Code ...
}
```

Figure 8: Function prototype for the linear classifier kernel that uses fixed point weights

Use the `aoc` flag `-c` to obtain a compilation report for your modified kernel and examine the resource utilization of implementing fixed-point addition and multiplication. Determine the per-adder and per-multiplier resource cost and compare them to the numbers you determined in Part III. Unroll your loops to fully utilize the resources of the FPGA and maximize the kernel's performance. How many times can you unroll your loop before running out of resources, and how does that compare to your design from Part II? Compile your design and use it to classify the MNIST test images. What is the runtime of your new kernel and how does it compare to that of your kernel from Part II? Is your classification accuracy affected? Why or why not? In what situations would changing from floating-point arithmetic to fixed-point arithmetic affect (and not affect) classification accuracy?