COMP4601 Project report
K-Nearest Neighbours Acceleration

Team Members

1. Vincent Pham z5363266
2. Tony Yang z5356286
3. Arisa Thangpaibool z5433515
4. Alex Ye z5316722

Introduction

Fundamental areas in machine learning include handwritten digit identification and signature verification, with applications ranging from digital forensics to security systems. The objective of this project is to produce accurate and efficient handwritten verification by using the MNIST dataset and the k-Nearest Neighbours (kNN) algorithm. The main goals are to identify handwritten digits, accelerate the kNN algorithm with High-Level Synthesis (HLS), and enhance system performance.

Handwritten signature verification is a crucial activity that is often difficult to process data collection and labeling as well as the limited availability of datasets because of privacy concerns. For tasks concerning signature verification, the kNN method is a good choice due to its straightforwardness in usage and efficiency, especially when dealing with smaller datasets. It operates by computing the distance between the input vector and all training vectors, subsequently classifying the input vector based on the majority label of its nearest neighbors.

The MNIST dataset, a widely recognized benchmark for the image processing and machine learning, comprises 60,000 training images and 10,000 testing images of handwritten digits. Each image is a 28x28 pixel representation of digits ranging from 0 to 9, making the dataset a critical resource for developing and evaluating digit recognition systems. The dataset's extensive use in academia and industry ensures that innovations built upon it are both relevant and impactful. For this project, we convert these images to 784 bits, simplifying the data by representing non-zero bytes as 1 and zero bytes as 0. This binary representation facilitates efficient processing and storage.

The project seeks to reduce computation time, enabling faster and more efficient processing of handwritten digit and signature data. By optimizing data types, and leveraging parallel processing capabilities, we aim to achieve high throughput and low latency.

System Design

Software Design

Firstly, some preprocessing is needed to transfer the MNIST dataset for use in C/C++. Originally, each data sample consisted of 28x28 bytes of data representing a 28x28 pixels image of the character. However, to reduce the complexity of the distance algorithm as well as reduce the amount of memory used by the dataset, we can simply just store the data as 28x28 bits where any non-zero bytes are merely converted to 1 bit. The samples are simply stored one after another in a binary file as shown in the Python code below:

```python
from mnist import MNIST

mndata = MNIST('samples')

images, labels = mndata.load_training()

file1 = open("train.bin", "wb")
for idx in range(len(images)):
    file1.write(labels[idx].to_bytes(1, 'big'))
    byte = 0
    for i, x in enumerate(images[idx]):
        if x != 0:
            byte += 1
        if i % 8 == 7:
            file1.write(byte.to_bytes(1, 'big'))
            byte = 0
        byte = byte << 1
file1.close()
```

Figure 1: Code snippet of preprocessing

Next is to write the software implementation of the KNN algorithm on the MNIST data set. Below is the pseudocode for the baseline implementation used, however, as will be outlined in the optimizations section, some components are changed, such as the removal of the sorting algorithm.

The actual software implementation follows the pseudocode very closely. Some changes that are made are the number of arguments passed in. All training and testing data is passed in and the KNN algorithm is essentially run for every single testing sample.

```
int knn(int k, sample_s *test_samples, int test_size, sample_s train_samples[TRAIN_SIZE])
```

There is an outer loop in the KNN function that runs the pseudocode for *test_size* times, once for every test sample. This function returns the number of correctly predicted labels. This is later changed so that only one test sample is passed every time the function is called and the function

would return a single bit indicating if a successful prediction is made. Another note to make is that we used the merge sort algorithm to sort the distances.

Algorithm 1 K-Nearest Neighbors (KNN)

```
1: Input_knn:
2:     k ← number of nearest neighbors to consider
3:     train_dataset ← array of data images from the training dataset
4:     n ← size of train_dataset
5:     cls ← number of distinct classes in train_dataset
6:     test_input ← a single image from the test dataset
7:
8: function KNN(input_knn)
9:     Initialize dist[n] array
10:    distance_loop:
11:    for i ← 0 to n − 1 do
12:        dist[i] ← COMPUTE_DISTANCE(train_dataset[i], test_input)
13:    end for
14:    Sort dist[] in increasing order
15:    Initialize freq[cls] array of zeroes
16:    frequency_loop:
17:    for i ← 0 to k − 1 do
18:        train_class ← get class of train_dataset[i]
19:        freq[train_class] ← freq[train_class] + 1
20:    end for
21:    max_index_loop:
22:    max ← freq[0]
23:    for i ← 1 to cls − 1 do
24:        if freq[i] > max then
25:            max ← freq[i]
26:        end if
27:    end for
28:    return max
29: end function
30:
31: function COMPUTE_DISTANCE(train_input, test_input)
32:    return distance metric between train_input and test_input
33: end function
```

Figure 2: KNN pseudocode

To compute the distance between each sample we essentially just count the number of bits that are different between the two samples. In the baseline code, we can do this by performing an XOR between every byte of the two samples, and then using a predefined lookup to count the number of bytes after the XOR:

```
//For fast pop count
unsigned short int bitcount[] = {
  0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
  1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, . . .
```

Figure 3: Fast pop count array

Porting to hardware

To fit memory constraints, we further reduce the amount of data we use per sample by only selecting bytes 20 to 69 (50 bytes), which will reduce the size of each sample, but also result in a

loss in accuracy. This is seen in the results section where all hardware versions use the truncated data version whilst the software baseline makes use of all 98 bytes of data.
Another change made was that to satisfy the memory constraints of hardware, we were only able to use 6,000 training samples as opposed to the 60,000 original samples included in the original dataset. This does result in decreased accuracy.
As we have chosen to use AXI4-LITE for this project, our KNN function has the following arguments and pragmas below:

```
void knn(sample_t & input, ap_uint<1>& output)
{
    #pragma HLS INTERFACE s_axilite port=output
    #pragma HLS INTERFACE s_axilite port=input
    #pragma HLS INTERFACE s_axilite port=return
```

Figure 4: KNN arguments and pragmas for AXI4-LITE

The input is a 404-bit data type representing the test sample inputted. The lowest 4 bits are used for the label. The output is a single bit with 1 indicating that the algorithm successfully predicted the testing sample's label. The testbench will pass all the testing samples one at a time into this function. Additionally, all 6,000 training samples used are predefined in a header file such that we can bypass the latency associated with passing the training sample from DRAM into BRAM via AXI4-LITE.

Hardware Design

After generating the RTL from the SW code in Vivado HLS, we used the provided FIR base Design project to create the block diagram and generate the bitstream and pl.dtbo files to run on the KV260 board. These files are then transferred onto the board via USB and the HW implementation is verified using Vitis HLS by reading from the testing samples stored on the board, inputting the test samples to the kNN algorithm input address and reading from the output address of the kNN. The kNN IP highlighted in green is connected to the master of the AXI Interconnect and the design's clock and reset signals. Our designs use the AXI4-lite interface.

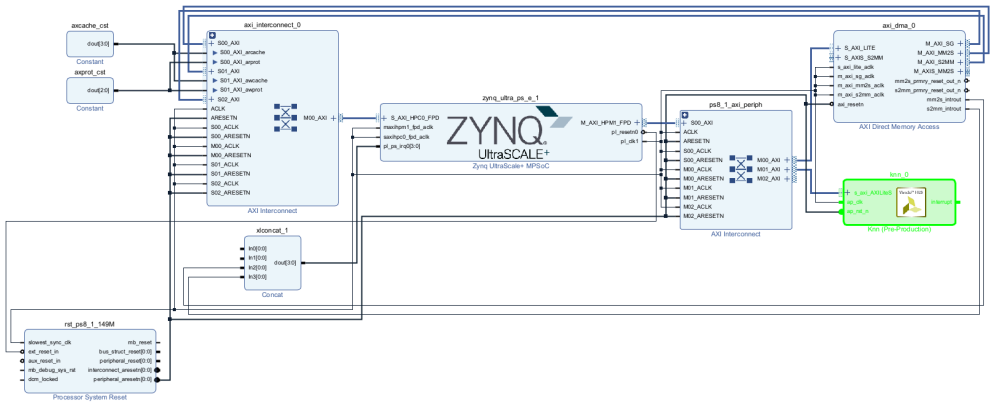

Figure 5: Block diagram of the HW architecture

We initially attempted to implement the AXI4-stream interface. However, this failed because we encountered an import error with the <string> library used by the axiDMAProxy modules in our Vitis HLS program. Consequently, we could not verify the results of HW. Instead we successfully implemented our kNN algorithm using the AXI4-lite interface. The AXI4-lite interface uses a traditional memory mapped address and data interface with a lack of burst access capability. Additionally, its handshake process to transfer address, data and control information are detrimental for our kNN task which is heavily data-centric **(Figure 6)**. Certainly, an AXI4-stream interface would improve the execution time of our kNN algorithm greatly, because of its high-speed streaming data capability without the overhead of a handshake mechanism. Despite these challenges, we demonstrate the effectiveness of HW acceleration using the AXI4-lite interface, greatly reducing execution time from 520.33 unaccelerated to 1.56 accelerated on 5000 test samples and with no loss in accuracy. Future work should consider the application of AXI4-lite stream to leverage its high-speed streaming capabilities.

```
72          // Get test input
73          char label;
74          fread(&(label), sizeof(label), 1, test_fp);
75          unsigned char buffer[SAMPLE_SIZE];
76          fread(buffer, sizeof(buffer), 1, test_fp);
77
78          // stop IP
79          *(uint32_t *)(axiBasePtr + USER_IP_ADDR_OFFSET_CTRL) = 0x0;
80
81          input_ptr[0] = label & 0b00001111;
82          for (int j = 20; j < 70; j++)
83          {
84              input_ptr[j - 20 + 1] = reverse_bits(buffer[j]);
85          }
86
87          for (int j = 0; j < 51; j++)
88          {
89              *(uint32_t *)(axiBasePtr + USER_IP_ADDR_OFFSET_INPUT_DATA + j * 8) = input_ptr[j];
90          }
91
92          // start IP
93          *(uint32_t *)(axiBasePtr + USER_IP_ADDR_OFFSET_CTRL) = 0x1;
94
95          while (!(*(uint32_t *)(axiBasePtr + USER_IP_ADDR_OFFSET_CTRL) & 0b1110))
96              ;
97
98          if ((*(uint32_t *)(axiBasePtr + USER_IP_ADDR_OFFSET_OUTPUT_DATA) & 0b1) != 0)
99          {
100             hits++;
101         }
```

Figure 6: The code snippet demonstrates getting the test input and reading it into address with a handshake protocol (lines 78 to 93) where the IP is first stopped.

Optimisation

Figure 7 presents the C++ implementation of the **K-Nearest Neighbors (KNN)** function. The function contains six loops: **data_loop**, **label_ini_loop**, **distance_loop**, **freq_init_loop**, **neighbour_loop**, and **bitcount_loop**. Below is a detailed analysis of each loop with considerations for optimization:

1. **data_loop:**
   The data_loop can be fully unrolled without any concern for increased utilization. This loop constructs the test_data based on the given input array. In a hardware context, this is analogous to a simple wire connection, allowing us to unroll this loop essentially "for free."
2. **label_ini_loop:**
   The label_ini_loop is responsible for initializing the array to a specific value. Due to the small size of K in our implementation, we can unroll this loop. Unrolling this loop involves altering the reset state of the registers, which would inevitably increase utilization. This decision is a trade-off between time and space; unrolling or not unrolling has minimal impact on the solution. Given our emphasis on performance, we decided to unroll the loop.
3. **distance_loop:**
   The distance_loop cannot be fully unrolled for two reasons: insufficient memory and inherent data dependencies within the loop. The inner loop's function is to find the K smallest numbers encountered so far, so the current iteration will have an impact on the next iteration. Therefore, the only viable optimization method is pipelining. By partitioning the curr_dist array, we can achieve an interval of 1.
4. **freq_init_loop:**
   Similar to the label_ini_loop, unrolling the freq_init_loop has minimal impact on performance and utilization. However, due to our preference for optimizing performance, we choose to unroll this loop.
5. **neighbour_loop:**
   There is no possibility of optimizing the neighbours_loop. Unrolling is not feasible due to data dependencies, and pipelining is hindered by concurrency issues. While human designers might solve this through forwarding, our attempts to pipeline the neighbours_loop indicated reduced accuracy, suggesting that the High-Level Synthesis tool is insufficiently sophisticated to handle this hazard. This was not an issue in distance_loop because the distances array was swapped in the same order.
6. **bitcount_loop:**
   The bitcount_loop is essentially inside of the distance_loop, so it must be fully unrolled as a result of the pipelining of the distance_loop. Fortunately, this unrolling doesn’t have much impact on utilization, as it can be implemented using XOR gates and an adder tree to produce the result.

```cpp
void knn(unsigned char input[51], ap_uint<1>& output) {
    #pragma HLS INTERFACE s_axilite port=output
    #pragma HLS INTERFACE s_axilite port=input
    #pragma HLS INTERFACE s_axilite port=return

    ap_uint<400> test_data;
    data_loop:
    for (int i = 50; i >= 1; i--) {
        ap_uint<8> temp = input[i];
        test_data += input[i];
        if (i != 1) {
            test_data = test_data << 8;
        }
    }

    label_t test_label = input[0];

    // Array that stores the k closest distances, will be sorted
    unsigned short int distances[K][2];
    label_ini_loop:
    for (int i = 0; i < K; i++) {
        distances[i][0] = MAX_DIST;
        distances[i][1] = -1; // Uninitialised
    }

    distance_loop:
    for (int i = 0; i < TRAIN_SIZE; i++) {
        data_t data = 0;
        for (int kkk = 20; kkk < 70; kkk++) {
            data.range((kkk-20) * CHAR_SIZE, (kkk + 1 - 20) * CHAR_SIZE - 1) = train[kkk+1 - 20][i];
        }

        unsigned short int curr_dist = calculate_distance(test_data, data);
        label_t curr_label = train[0][i];
        for (int j = 0; j < K; j++) {
            // Keep swapping curr distance and label until the end of the array

            if (curr_dist < distances[j][0]) {
                unsigned short int temp1 = curr_dist;
                curr_dist = distances[j][0];
                distances[j][0] = temp1;
                label_t temp2 = curr_label;
                curr_label = distances[j][1];
                distances[j][1] = temp2;
            }
        }
    }

    // Frequency of most frequent character
    int max_freq = 0;
    // Most frequent character
    int most_freq = -1;
    // Array to store frequencies of characters
    int freq[NUM_CHARS];
    freq_init_loop:
    for (int i = 0; i < NUM_CHARS; i++)
        freq[i] = 0;

    // Filter through k nearest neighbours
    neighbours_loop:
    for (int i = 0; i < K; i++) {
        int label = distances[i][1];
        freq[label]++;
        if (freq[label] > max_freq) {
            max_freq = freq[label];
            most_freq = label;
        }
    }

    if ((int)test_label == most_freq) output = 1;
    else output = 0;
}

// Implementation counts the number of differring bits between the two samples.
unsigned short int calculate_distance(data_t a, data_t b) {
    unsigned short int distance = 0;
    bitcount_loop:
    for (int i = 0; i < DATA_SIZE; i++) {
        if (a[i] != b[i]) distance++;
    }
    return distance;
}
```

Figure 7: knn cpp code

Results

For evaluation, we collected the execution time, accuracy, and utilization of three versions of our kNN algorithm. The baseline version is based on our SW system design. When transferring to HW we identified key issues with its implementation which resulted in inconclusive and incorrect results. One such issue is that the training size was too large and needed to be partitioned. Thus, we predefine the training samples in our succeeding solutions to mitigate this issue.

Model	Total execution time (seconds) Loading training / Loading test		
	Number of test samples		
	1000	**5000**	**10000**
Baseline SW	0.038 / 77.09	0.036 / 387.21	-
Baseline HW	0.037 / 25.137*	0.037 / 125.7*	-
Optimized with predefined training samples SW	0 / 102.08	0 / 520.33	-
Optimized with predefined training samples HW	0 / 31.21	0 / 156.04	-
Optimized with predefined training samples and pragmas SW	0 / 113.49	0 / 553.81	0 / 1106.72
Optimized with predefined training samples HW	**0 / 0.312**	**0 / 1.56**	**0 / 3.124**

*Baseline was bugged causing inaccurate results

Table 1: Execution time results

Model	Accuracy		
	Number of test samples		
	1000	**5000**	**10000**
Baseline SW	**96.1%**	**94.96%**	-
Baseline HW	100%*	100%*	-
Optimized with predefined training samples SW	90.3%	88.98%	-
Optimized with predefined training samples HW	90.3%	88.98%	-
Optimized with predefined training samples and pragmas SW	90.3%	88.98%	-
Optimized with predefined training samples and pragmas HW	**90.3%**	**88.98%**	**91.84%**

*Baseline was bugged causing inaccurate results

Table 2: Accuracy results

Model	Max Latency (cycles)	BRAM_18K	DSP48E	FF	LUT	URAM
Baseline	2514034	204 (70%)	0 (0%)	2076 (~0%)	1567 (1%)	0 (0%)
Optimized with predefined training samples	3126135	0 (0%)	0 (0%)	3150 (1%)	47568 (40%)	0 (0%)
Optimized with predefined training samples and pragmas	6085	25 (8%)	0 (0%)	5046 (2%)	49350 (42%)	0 (0%)

Table 3: Utilization results

The third iteration with predefined training samples and pragmas use the directives in Figure 8

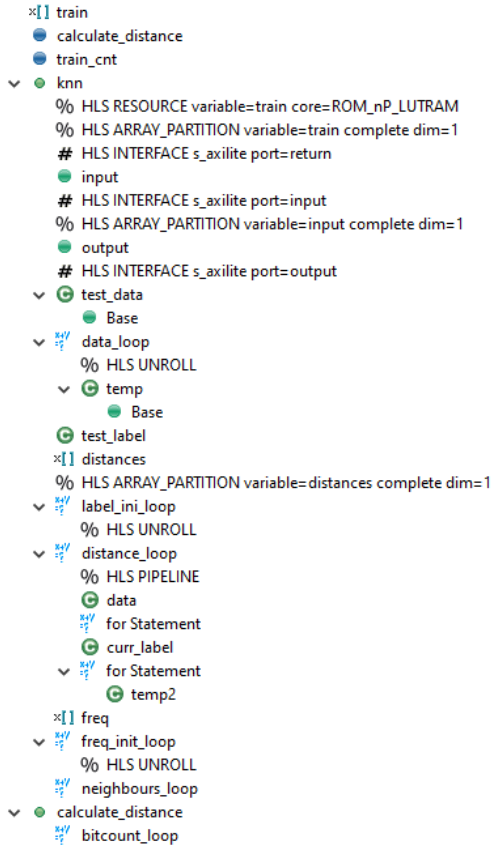

Figure 8: Directives for optimization of predefined training samples

Discussion

In the results section, we observe that the performance of the KNN algorithm has increased by 500 times, reducing the clock cycles from 3 million to 6 thousand. However, the total execution time shows an improvement of only 100 times. This discrepancy is due to the fact that only a portion of the program has been accelerated.

During the execution process, besides the time spent on the KNN algorithm, time is also consumed by file reading operations via system calls. Using **Amdahl's Law**, we can determine that the KNN algorithm accounts for approximately 99.198% of the total execution time originally. In a theoretical scenario where the acceleration of KNN is infinite, the maximum overall performance improvement would be 124.75 times faster.

Therefore, to achieve a better overall performance, it is essential to optimize not only the KNN algorithm but also the other components of the program. This holistic approach is necessary to overcome the limitations imposed by the non-accelerated parts of the system.