kNN algorithm

Start

K-NEAREST NEIGHBOURS ACCELERATION

Presented by

Vincent	Pham	z5363266
Tony	Yang	z5356286
Arisa	Thangpaibool	z5433515
Alex	Ye	z5316722

PROBLEM STATEMENT

Objectives:

- Accelerate kNN using HLS
- Recognize handwritten digits and classify them correctly
- Implement optimization strategies to improve overall performance

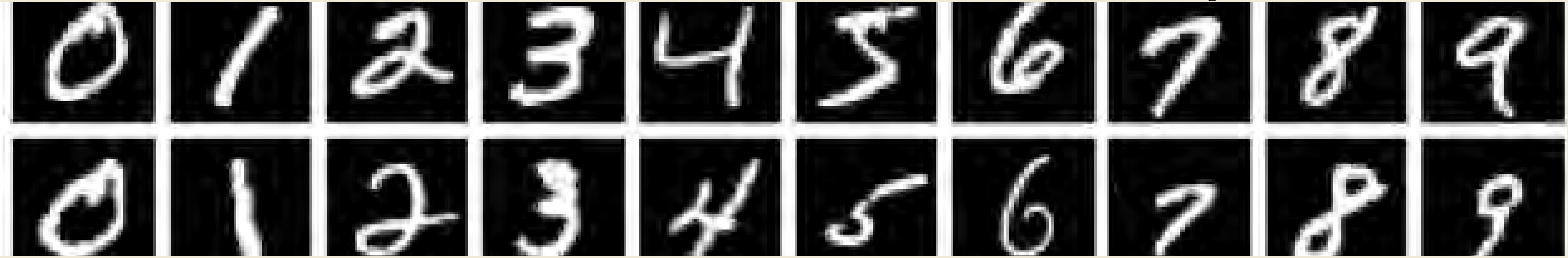
KNN ALGORITHM

- Used for both classification and regression tasks
- Lazy learning algorithm
- Input training dataset with labels
- Test sample:
  - Compute distance to all training samples
  - Sort distances
  - Assign labels

MNIST DATASET

- Handwritten digit dataset 28x28 pixel (784 bytes)
- Converted to 784 bits by making any non-zero bytes into a 1 bits, and any zero bytes to zero bits
- Used for training and testing in machine learning

kNN + MNIST
Handwritten Signature Verification



SW DESIGN

- Use 60000 training samples and 10000 testing samples
- compute_distance() -> Compare the number of differing bits
- sort() -> Mergesort and use the lowest k labels

```
root@xilinx-k26-starterkit-2021_1:/lib/firmware/xilinx# ./baseline 500
Time to load training samples = 0.035952 seconds
Time to execute test samples = 38.660167 seconds
Total taken to execute = 38.696119 seconds
Accuracy of 0.960000 using k = 5 and 500 test samples
root@xilinx-k26-starterkit-2021_1:/lib/firmware/xilinx#
```

Baseline SW evaluation

Algorithm 1 K-Nearest Neighbors (KNN)

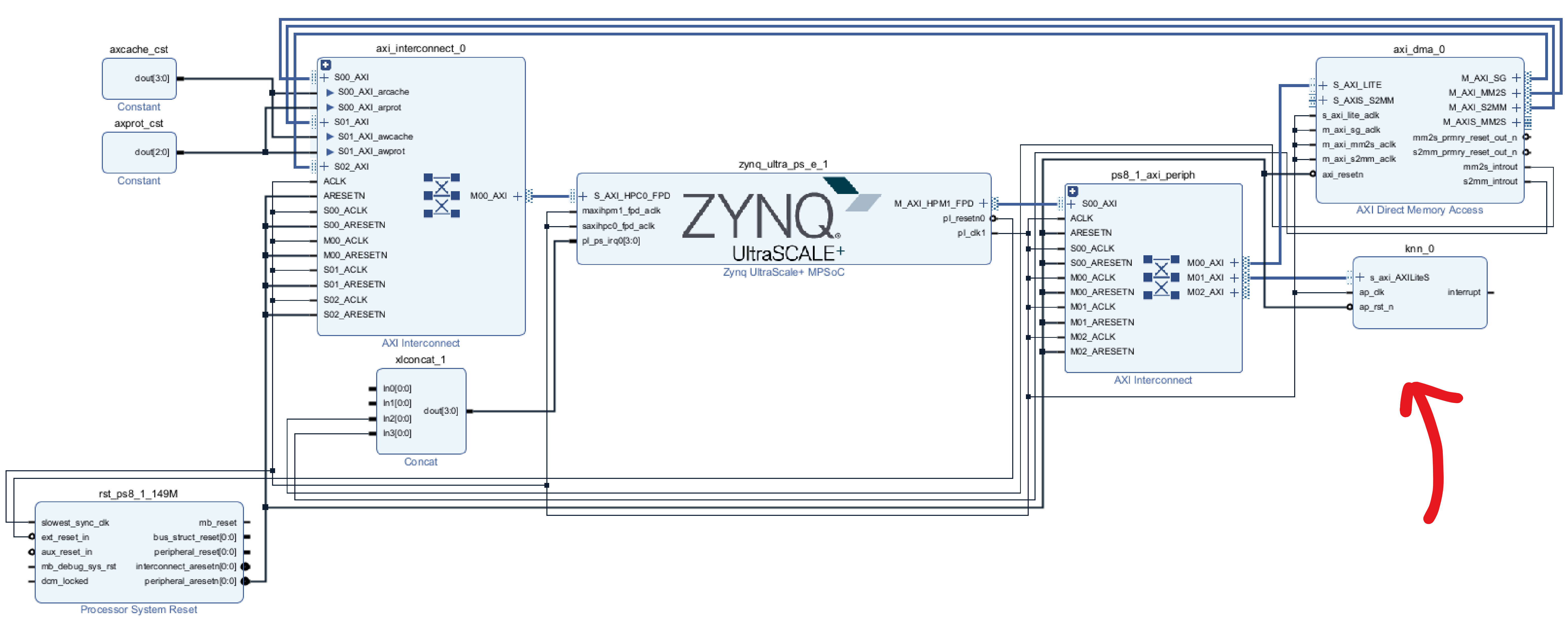
```
Input_knn:
    k ← number of nearest neighbors to consider
    train_dataset ← array of data images from the training dataset
    n ← size of train_dataset
    cls ← number of distinct classes in train_dataset
    test_input ← a single image from the test dataset

function KNN(input_knn)
    Initialize dist[n] array
    distance_loop:
    for i ← 0 to n − 1 do
        dist[i] ← COMPUTE_DISTANCE(train_dataset[i], test_input)
    end for
    Sort dist[] in increasing order
    Initialize freq[cls] array of zeroes
    frequency_loop:
    for i ← 0 to k − 1 do
        train_class ← get class of train_dataset[i]
        freq[train_class] ← freq[train_class] + 1
    end for
    max_index_loop:
    max ← freq[0]
    for i ← 1 to cls − 1 do
        if freq[i] > max then
            max ← freq[i]
        end if
    end for
    return max
end function

function COMPUTE_DISTANCE(train_input, test_input)
    return distance metric between train_input and test_input
end function
```

kNN pseudocode

HW BLOCK DIAGRAM

AXI4-LITE INTERFACE

- Traditional memory mapped address/data interface
- Lacks burst access capability
- Handshake process to transfer address, data, and control information



HW ARCHITECTURE

```c
// Get test input
char label;
fread(&(label), sizeof(label), 1, test_fp);
unsigned char buffer[SAMPLE_SIZE];
fread(buffer, sizeof(buffer), 1, test_fp);

// stop IP
*(uint32_t *)(axiBasePtr + USER_IP_ADDR_OFFSET_CTRL) = 0x0;

input_ptr[0] = label & 0b00001111;
for (int j = 20; j < 70; j++)
{
    input_ptr[j - 20 + 1] = reverse_bits(buffer[j]);
}

for (int j = 0; j < 51; j++)
{
    *(uint32_t *)(axiBasePtr + USER_IP_ADDR_OFFSET_INPUT_DATA + j * 8) = input_ptr[j];
}

// start IP
*(uint32_t *)(axiBasePtr + USER_IP_ADDR_OFFSET_CTRL) = 0x1;

while (!(*(uint32_t *)(axiBasePtr + USER_IP_ADDR_OFFSET_CTRL) & 0b1110))
    ;

if ((*(uint32_t *)(axiBasePtr + USER_IP_ADDR_OFFSET_OUTPUT_DATA) & 0b1) != 0)
{
    hits++;
}
```

Vitis HLS AXI4-LITE handshake

Memory

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
distances_0_U	knn_distances_0	0	32	2	0	5	16	1	80
distances_1_U	knn_distances_1	0	10	1	0	5	5	1	25
freq_U	knn_freq	0	64	5	0	10	32	1	320
train_0_U	knn_train_0	0	4	375	0	6000	4	1	24000
train_1_U	knn_train_1	0	8	750	0	6000	8	1	48000
train_10_U	knn_train_10	0	8	750	0	6000	8	1	48000
train_11_U	knn_train_11	0	8	750	0	6000	8	1	48000
train_12_U	knn_train_12	0	8	750	0	6000	8	1	48000
train_13_U	knn_train_13	0	8	750	0	6000	8	1	48000
train_14_U	knn_train_14	0	8	750	0	6000	8	1	48000
train_15_U	knn_train_15	0	8	750	0	6000	8	1	48000
train_16_U	knn_train_16	0	7	657	0	6000	7	1	42000
train_17_U	knn_train_17	0	8	750	0	6000	8	1	48000
train_18_U	knn_train_18	0	8	750	0	6000	8	1	48000
train_19_U	knn_train_19	0	8	750	0	6000	8	1	48000

Training samples defined in Vivado HLS and being converted into LUT

```
Accelerator successfully removed.
Accelerator loaded to slot 0
# Testing samples = 10000
Time to execute test samples = 3.124631 seconds
Accuracy of 0.918400 using k = 5
```

Vitis HLS accelerated kNN output

OPTIMISATIONS

Kira KV260 doesn’t have enough memeory we need

- Reduce the training size (60000 -> 6000)
- Reduce each training data from 28*28 bits to 400 bits
- Prestore the training data in FPGA to reduce the transferring time and decrease the usage of BRAM

```
void knn(unsigned char train[6000][51], unsigned char input[51], ap_uint<1>& output)
```

```
void knn(unsigned char input[51], ap_uint<1>& output)
```

- Use an iterative loop to replace mergesort to get the k smallest distances
    - HLS doesn’t support pointers comparasion so mergesort costs O(n) space
- Adding directives
    - Pipelining the loop getting the k smallest distances
    - Unrolling initialise loops
    - Using ROM to store the training data to save BRAM and support concurrent reads
    - Partition the arrays used in the pipelined loop to aviod II violation

ACCURACY

- We observe the accuracy of the KNN in SW and HW floating around 90%.
- Baseline SW accuracy was achieved by using 10,000 training samples which is why the accuracy is higher because it allows the KNN algorithm to compare the distances of more data points in the input space
- With the Baseline HW, the results is invalid because the AXI4-lite communication was invalid
- Overall, there is no accuracy loss from our optimizations

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

TIMING AND UTILIZATION

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

- Our optimization strategies greatly decreased the total execution time of the KNN algorithm.

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%)

DISCUSSION

What worked; what didn't; why?

Challenges

- Limited HW resources
  - Reduced the number of training samples from 60000 to 6000
- AXI4-lite stream communication failing to work
  - Import error with <string> library
- Accuracy degradation from reducing training size
  - Our SW solution which uses 60000 training samples has higher accuracy compared to the HW implementation

RECOMMENDATIONS

What you would do (differently), given more time

PS TO PL COMMUNICATION USING AXI4-LITE STREAM INTERFACE

Using the AXI4-stream to leverage its high-speed streaming data capability is more suitable for the KNN task because it is data-centric. Unlike AXI4-lite, AXI-stream can burst an unlimited amount of data without the handshake mechanism between test sample inputs

USE MULTIPLE FPGAS TO STORE MORE TRAINING SAMPLES

The accuracy of the hardware version was lowered due to the inability to store 60000 training samples. Using 10 FPGAs to store all training samples would increase the accuracy to match what is seen in the software version.

USE DIFFERENT ALGORITHMS

Try using other algorithms and see if we can get further performance improvements such as support vector machines, decision trees, artificial neural networks or logistic regression models.