Optimized Kth Nearest Neighbor Implementation on an FPGA Cluster

Andrew Farran
Grant Fettig
Daniel McKee

Sponsor: Lina Sawalha, Ph.D

ELECTRICAL and COMPUTER ENGINEERING 4820

April 27th, 2018

WESTERN MICHIGAN UNIVERSITY

College of Engineering and Applied Sciences

Table of Contents

INTRODUCTION **2**

DISCUSSION **2**

Methodology 2

Client Logic 8

Server Logic 11

FPGA Logic 13

Processor Results 21

FPGA Results 24

CONCLUSIONS **26**

Reusability 26

Contributions 26

APPENDIX A **27**

Client Code 27

Server Code 30

FPGA Code 40

INTRODUCTION

A huge problem with data analysis today can be defined through the term data deluge. Data deluge, also known as information explosion, is the rapid increase of data that is being recorded every day is too massive for us to analyze in an efficient manner. Data needs to be analyzed faster, and one way to do this is with accelerated hardware. Given this problem, the goal of the group was to develop a reusable Field Programmable Gate Array (FPGA) cluster prototype that utilizes a systolic array to optimize the Kth Nearest Neighbor (KNN) machine learning algorithm. The way to achieve the largest amount of speedup for this algorithm is by increasing the number of processing units in the cluster.

DISCUSSION

Methodology

The primary goal was to distribute data from the client PC evenly to all of the server Zynq boards (Zybo) via ethernet. The Zybo was chosen for this implementation due to its Zynq-7000 system on a chip (SoC) containing a dual-core ARM processor and FPGA logic. The ARM processor in each Zybo would then send the data from the DRAM to the FPGA via DMA(direct memory access). Finally, the processor would then read the registers on the side of the FPGA and send those values back to the PC via ethernet.

The first step taken in reaching this goal was making sure the DMA could send the large amount of integers in the dataset and that the integers were correct once reaching their destination. The DMA was tested to see if its bit-width could be increased from the 32-bit width it started at. After running into problems with the DMA width at 256 bits, it was decided that 64 bits would be sufficient. The figure below shows how to configure the DMA settings along with

the block diagram for sending data from the DRAM to FPGA via DMA. Figures 1 and 2 demonstrate this.

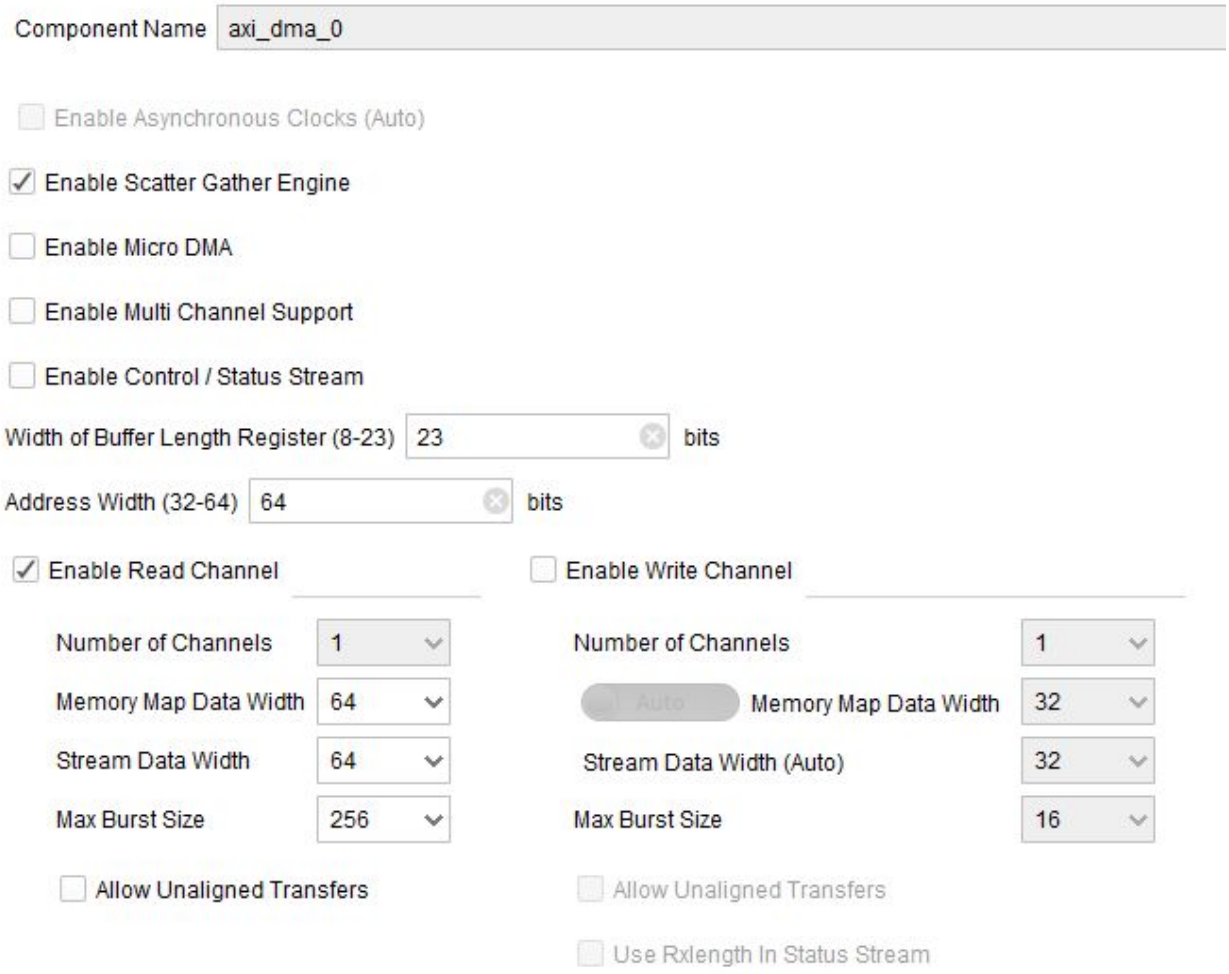


Figure 1. Configuration of the DMA settings in Vivado HLS programming environment.

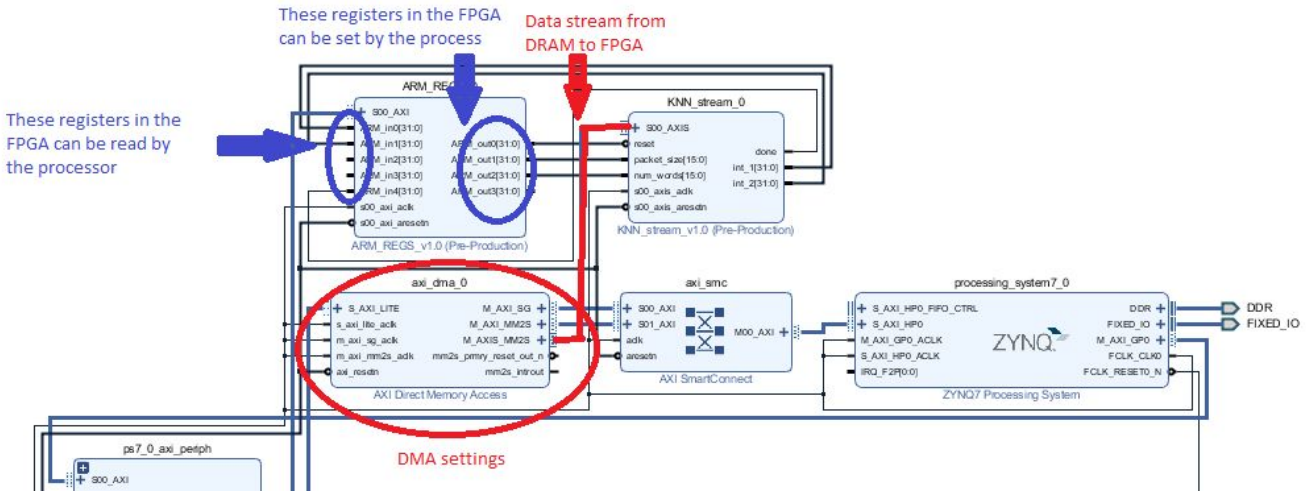


Figure 2. The block diagram with the DMA module circled in red with the registers on the side of the FPGA circled in blue.

For testing purposes, small amounts of data were initially used to make sure the DMA was functioning properly. Once successful tests were completed, the full dataset of 5 million integers was used, checking to see that the results were accurate. After the large set of data was able to be sent, the next step was getting the Zybo set up so that it could receive the large dataset in form of ethernet packets and be able to send at least one packet back to the PC. The first step was looking at a loopback mode example online. Following unsuccessful attempts to alter the code so that the communication was not in loopback mode, it was as decided to look at a different example. Following steps through a Xilinx Zynq Echo Server sample project, data packets were successfully sent to the Zybo from the PC and then immediately echoed back. This code worked by configuring the Zybo with an IP of 192.168.1.10 connecting through communications port 7. Port 7 is a TCP or UDP port that is specifically meant to be used for the echo protocol. The code configured a callback function that would be called whenever a packet

was received. This packet was put into a payload and then sent back. By storing this payload in a buffer and moving the "send_ethernet()" function to a different location, the code functioned properly. Global variables were created to determine how many packets the ethernet would accept before it would disable the callback function. After the callback function was disabled, the buffer would be sent through the DMA and the last two integers were read by the processor It was then put into p->payload, and sent back to the PC where the results were monitored using Wireshark. This test can be shown below in Figure 3.

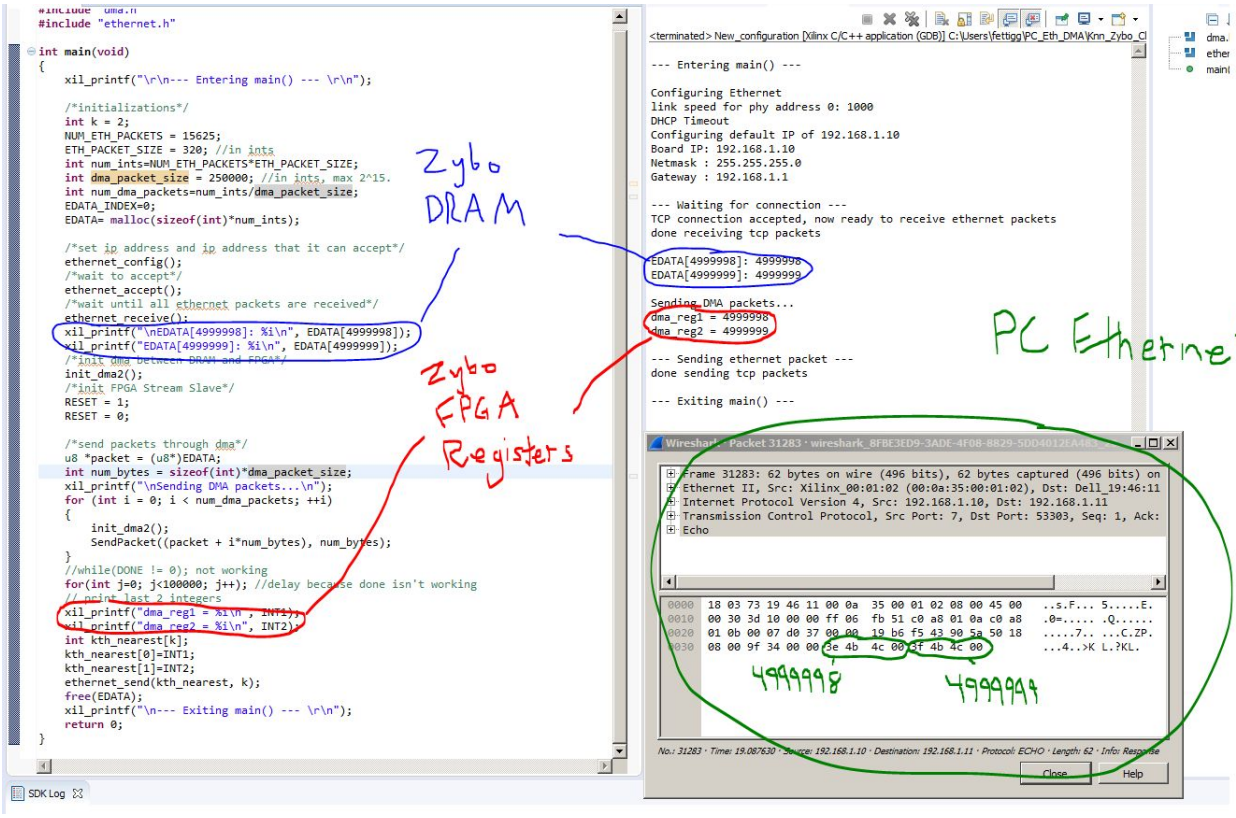


Figure 3. This shows specific lines in the Zybo code for the DMA. The bottom right window shows the data used, set by variable EDATA[], is accurate. Wireshark was used to confirm this.

The next step was incorporating an ethernet switch so that multiple Zybos could be used to receive data from only one client PC. After struggling with the Python code that was to

receive data back from the Zybo, it was discovered that using port 7 was not going to work for purpose of the project. Each Zybo was given a separate port number as well as IP address using ports 5, 11, 13 and 17 along with IPs 192.168.1.11, 192.168.1.12, 192.168.1.13, and 192.168.1.14 for the Zybos. The IP address for the PC was set to 192.168.1.10. After testing every IP and port combination individually, a single Python script was written to alternate between sending for each Zybo. After coming to the realization that the speed of the communication logic was not fast enough, it was decided that the Python code would need to be multithreaded. The entire system was then tested using the KNN algorithm written in C to be done on the processor of each individual Zybo because the algorithm in the FPGA was not yet finished at that time.

The hardware was designed top-down in terms of the block diagram shown in Figure 2, and the code was written from the bottom-up. The programmable logic is arranged as a hierarchy of modules using a systolic array design. The code and the block design were developed using Xilinx Vivado HDK.

A systolic array is a network of modular processing elements (PE). Identical PEs may be duplicated to perform the same operation in parallel, and different kinds of PEs can be placed before or after one another. This is shown in the following figure. The specifics of this design for the KNN algorithm are described in a later section.

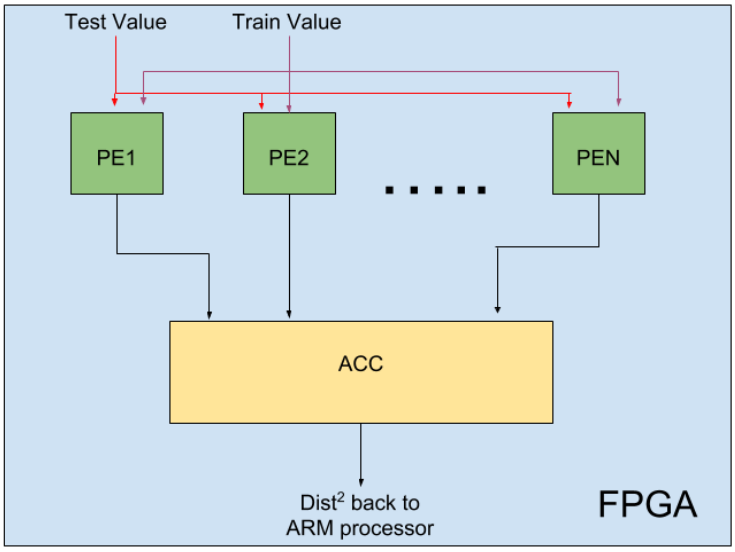


Figure 4. Multiple PEs perform operations on different sets of data in parallel before sending their results to an accumulator PE.

Going more in depth in regards to the PE and it's modular design, Figure 5 shows the flow of data through different pieces of the KNN process. The data must first be put through the ArithPE for the differences as well as the squares of those differences. The KnnPE then takes those results and sums them together which sends it to the top level array for finding the Kth nearest neighbors.

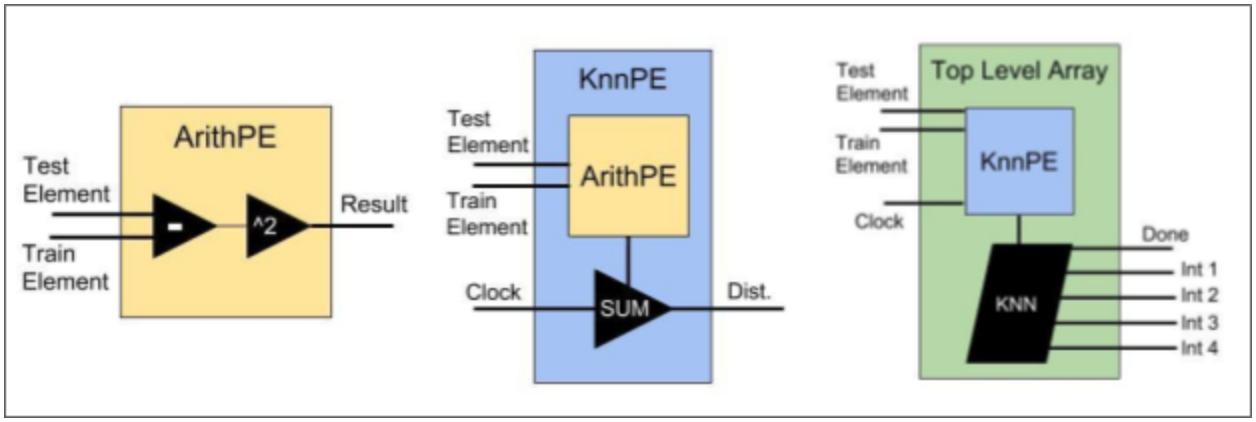


Figure 5. Modular design of the processing elements for distance and KNN computation.

To get a better understanding of the big picture, Figure 6 represents a very simplified version of the flow of logic in this implementation.

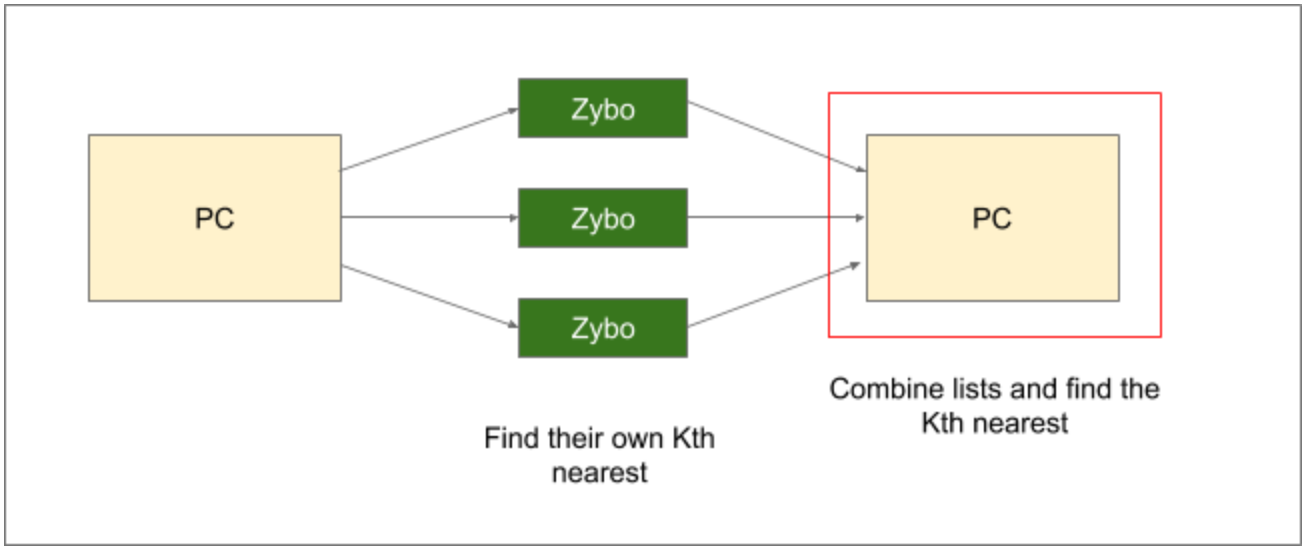


Figure 6. Overall picture of the dataflow

The dataflow begins with the PC where the Python code runs through its logic. The data that the Python code reads is sent to the ARM on the Zybo then to the FPGA for each board being used. Once the computations have been made on each board it is sent back to the PC to find the final kth nearest.

Client Logic

The Python script for this project was solely developed for the handling of the client logic on the PC. This is the starting point of the dataflow as it handles the reading of the binary file of 5 million integer containing the test and train vectors. The file being read through by this code is a file of randomly generated integers in the range [-20:20] which was created through a simple for- loop written in C language. It was not regenerated for each instance of the Python code running, rather it was generated once and became the set file that was used for testing. This made it easier to debug different pieces of the project as things were added or altered because

every successful run would provide the same results that were known to be correct. To read through this file, the Python code had to create a path to the file by using the "open()" system call. The call parameters are the name of the file, "bin_file.bin", as well as the mode, "rb". The mode needed to be set as "rb" due to the fact that when scripting with Python on Windows, the open function needs to know the file type to avoid potential data corruption. The next logical step once the file is opened is to read data from the file, but a connection to the server Zybo needs to be created first so data can be sent to a board immediately after each dimension of test vector and train vector is read. To do this, a socket needs to be created which is done using ".socket()" function with generic parameters "socket.AF_INET" and "socket.SOCK_STREAM". Once the socket itself has been initialized, a connection is made to the Zybo using the ".connect()" function with parameters matching the IP of the board as well as the port.

Now that a connection has been made, the ".read()" function is used to read the data from the file. The format of the data in the file is staggered, meaning the first 10,000 integers represent the first set of test vectors, and the next 10,000 integers represent the first set of train vectors. Because that is how the data is stored, that is also how the data is sent which is done utilizing the ".send()" function whose parameter are either the train vector payload or the test vector payload. Upon exiting the send function, meaning all the data has been sent, the Python code waits for the kth nearest neighbor computations to be completed on the FPGA. Once these are completed, the ".recv()" function receives the computed distances to do the final sort determining the top Kth nearest neighbors. Figure 7 below shows a single thread of this code as explained here. The key system call and functions are circled so as to bring them to attention.

```
def client_socket(TCP_IP, TCP_PORT, thread_offset, kth_nearest):
    #---Open binary file---
    file = open(file_name, "rb")
    #---Connect to client sockets---
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((TCP_IP, TCP_PORT))
    print("Connected to ", TCP_IP)
    #---Read file and send test vector---
    for i in range(0, pkts_per_vector):
        if i == pkts_per_vector-1:
            payload = file.read(last_pkt_size_bytes)
        else:
            payload = file.read(snd_pkt_size_bytes)
        client_socket.send(payload)
        time.sleep(1/10000000)
    #send the train vectors
    for i in range(0, train_vectors_per_board):
        file.seek(dim_bytes+i*num_zybos*dim_bytes+thread_offset*dim_bytes)
        for j in range(0, pkts_per_vector):
            if j == pkts_per_vector-1:
                payload = file.read(last_pkt_size_bytes)
            else:
                payload = file.read(snd_pkt_size_bytes)
            client_socket.send(payload)
            time.sleep(1/10000000)
    print("Done sending to ", TCP_IP)
    file.close()
    #---Receive packets---
    recpacket = client_socket.recv(rec_pkt_size_bytes)
    print("Received packet from ", TCP_IP)
    i=0
    j=3
    for x in range(0, k):
        kth_nearest.append(int.from_bytes(recpacket[i:j], byteorder='little'))
        i=i+4
        j=j+4
```

Figure 7. Single Python thread on client PC that creates a socket, connects to server Zybo over the created socket, reads from a binary file, sends the data from the file to the Zybo and receives data back once Zybo computations are completed.

As discussed briefly in text above, the Python code was multithreaded and Figure 7 represents only a single thread of this process. This means that for N number of Zybos up to four, this piece of code is running N number of instances at one time. For flexibility and ease of use regarding this code, the specific IP addresses, MAC addresses and port numbers for each board are set in conditional statements. All that needs to be changed is the integer variable describing the number of boards being used in that run. So if that variable is equal to two, only

two instances of this thread is being run and only the configuration settings for the first two boards are called.

Server Logic

The server side of the logic is for the Zybo board which was programmed using C in Xilinx Vivado SDK. As the Python socket code takes in parameters of a specified IP and Port number, these need to also be set on the server side matching what was set on the client side. Similarly to the Python code, all of the port numbers as well as MAC and IP addresses are held in conditional statements that are only reached based on the specified number of Zybos. The beginning of the program flow starts with the server logic as the C code handles all of the Ethernet configurations. When running the main method, after configuring the Ethernet, it will then begin polling for a connection through an established socket. This is where the Python code is run. As the Python code is launched and begins running through its own logic, the C portion of the code is ready to begin accepting the incoming packets. Using the "memcpy()" system call, as the data is received, it is copied into a buffer so that the DMA can handle the set of data. Below Figure 8 shows the conditional statement that sets the IP and MAC address as well as the port number if the user is only using one board. Figure 9 shows the system call that was used for placing the data into the buffer.

```
if(Zybo==1){
    IP=11;
    port=5;
    mac_ethernet_address[5]=0x02;
}
```

Figure 8. Lines that set the IP and MAC address and port number when user is using one board.

```
err_t recv_callback(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{

    /* indicate that the packet has been received */
    tcp_recved(tpcb, p->len);
    /* copy payload into char buff */
    memcpy(VECTORS+BUF_OFFSET,p->payload, p->len);
    BUF_OFFSET=BUF_OFFSET+p->len;
    ETH_PACKETS_RECEIVED++;
    pbuf_free(p);
    return ERR_OK;
}
```

Figure 9. memcpy() system call is used to put the received data into the buffer

The complete scope of data that needs to go into the buffer is of 100 training vectors and one test vector. Each of those, however, have a dimension of 10,000 integers per vector. As the buffer itself handles only one test and train vector at a time, the buffer will only have 20,000 integers stored in it during a given instance. This means that before the second set of 20,000 integers is saved in the buffer (the second test and train vector) the first set of 20,000 integers needs to be handled and sent to the FPGA. This is where the DMA logic begins handling the substantial amount of data. Running in a for-loop, the "SendPacket()" function sends the train and test vectors to the FPGA after reordering the data. This is all done in the Zybo ARM processor. On the following page, Figure 10 shows a nested for-loop with the manipulation of data by the first circled for-loop, as well as the "SendPacket()" function being used after the DMA is initialized by the "init_dma2()" function.

```
for(j=0;j<dim_ints;j++){
    dma_packet_buffer[m]=*((int*)(VECTORS)+j);
    m=m+2;
}

for(i=0; i<num_train; i++){
    m=0;
    for(j=0;j<dim_ints;j++){
        dma_packet_buffer[m+1]=
                *((int*)(VECTORS)+dim_ints+dim_ints*i+j);
        m=m+2;
    }
    init_dma2();
    SendPacket((u8*)(dma_packet_buffer+i*dma_packet_buffer_size),
            dma_packet_buffer_size);
}
```

Figure 10. Manipulation of data for the buffer is seen in the circled for-loop and the SendPacket() function uses the DMA to send the data from the buffer to the FPGA.

Once all the data has been handled by the DMA and is in the FPGA, the code must wait for the nearest neighbor computations to be completed. Once the FPGA done-bit is set high, the DMA is used again to transfer the data back to the ARM processor through the DRAM. This is done so that it can be sent back to the PC to do the final sort finding the Kth nearest neighbors between all boards being used.

FPGA Logic

The FPGA logic, which will be broken down further below, is used to complete the Kth nearest neighbor algorithm for a subsection of the data (train vectors). It then uses a distance formula to calculate the distances between the test vector and each train vector. Each result is then sorted using an insertion sort, placing the smallest distances in ascending order into an array. For the implementation here, the distance is only the sum of the differences between each

dimension squared. Taking the square root of the result does not change the relative distance so time was not wasted implementing that step. Below, Figure 11 shows the left side of the block diagram and Figure 12 shows the right side. Together they show all of the modules that were used each containing FPGA logic to complete a specific piece of the process.

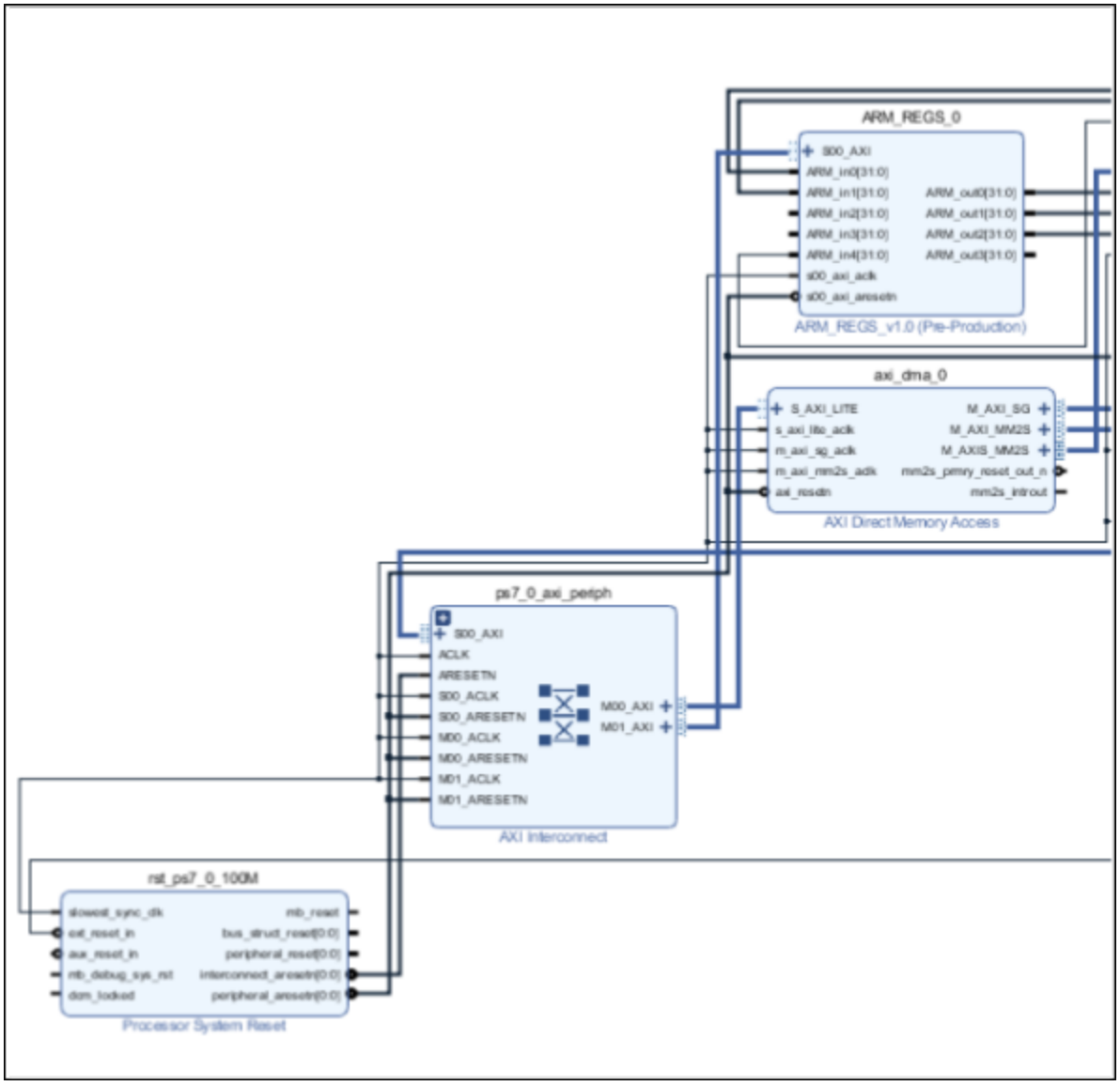


Figure 11. Left side of the block diagram showing four of the seven modules created.

The important module to mention here is the module for the DMA, axi_dma_0. The methodology for the DMA in the C code was explained above and later in the paper the specific functionality of the DMA code will be explained further. This module is what allows the transfer of data from the ARM to the FPGA through the DMA to happen successfully.

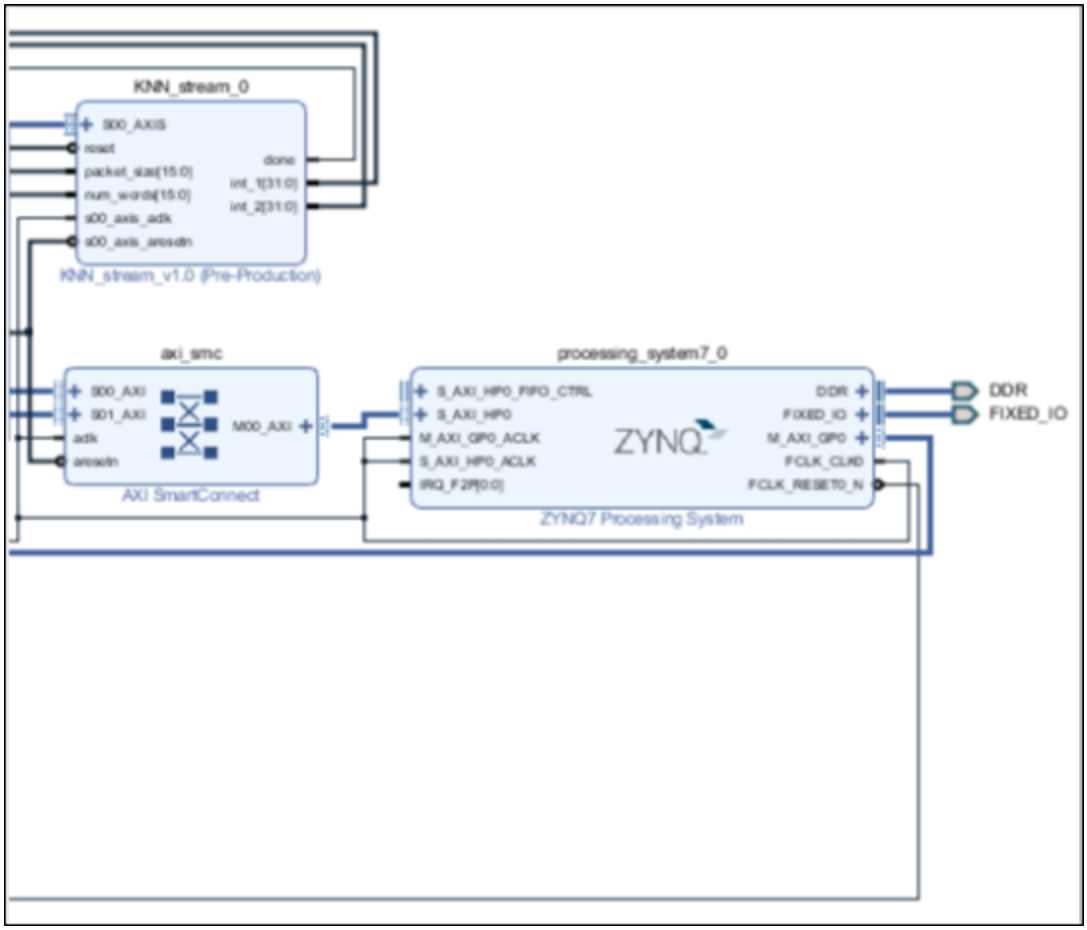


Figure 12. The right side of the block diagram showing the other three of the seven modules.

The following five figures will show snippets of the Verilog code. More specifically, pieces regarding the top module as well as the kth nearest neighbor calculations, which show the code for the processing element diagram seen in Figure 4.

```
//input data to test and train regs
always@(posedge S_AXIS_ACLK)
begin
    // Set FIFOs and pointers to 0
    if (reset) begin
        testv <= 32'b0;
        trainv <= 32'b0;
        newdata <= 1'b0;
    end
    // If writing data from DMA...
    else if (fifo_wren) begin
        testv <= S_AXIS_TDATA[31:0];
        trainv <= S_AXIS_TDATA[63:32];
        newdata <= 1'b1;
    end
    else
        newdata <= 1'b0;
end
```

Figure 13. Setting test and train to received data from DMA

Figure 13 to the left is the piece of code that inputs data from the ARM into the registers beside the FPGA. Both the test and train vectors are set as 32-bit numbers in the first if-statement and in the else-if statement, the test and train vectors are being set to the actual value of data that was received by the ARM.

```
//when available, output data and enable KNN on negedge
always@(negedge S_AXIS_ACLK)
begin
    // Set enable LOW on initial reset
    if ( reset ) begin
        KNN_en <= 1'b0;
        int_1 <= 32'b0;
        int_2 <= 32'b0;
    end
    // If new reg data, send data to KNN
    else if(newdata) begin
        KNN_en <= 1'b1;
        int_1 <= testv;
        int_2 <= trainv;
    end
    else
        KNN_en <= 1'b0;
end
```

Figure 14. This is the logic handling whether or not the KNN module is sent data

For code shown here in Figure 14, it is seen that output data is being set and KNN is being enabled on the negative edge of the clock if reset is high. If new data is coming in, the data is sent to the KNN algorithm logic.

The ArithPE portion of the modular processing element design is shown by Figure 15.

```
module ArithPE(
    input [31:0] x,
    input [31:0] y,
    output [31:0] z
    );

    wire [31:0] diff;

    assign diff = x - y;

    assign z = diff * diff;
endmodule
```

Here, 32-bit inputs ‘x’ and ‘y’ are created to represent the test and train vectors and the output ‘z’ is created to hold the result. The variable ‘diff’ is a wire that is assigned to be the result of the difference between ‘x’ and ‘y’ and finally ‘z’, which is the result that is to be output to the next piece of the PE design, is assigned as the difference squared.

Figure 15. Part of the PE that handles the difference and squaring of the test and train

```
always @ (posedge clk) begin
    //if reached total, reset
    if (count >= total-16'b1) begin
        sum <= 0;
        count <= 0;
    end
    //update sum and count
    else begin
        sum <= sum + sqrsum[size-1];
        count <= count + 16'b1;
    end
end

//preemtively set done on negedge
always @ (negedge clk) begin
    if (count >= total-16'b1) begin
        done <= 1'b1;
    end
    else
        done <= 1'b0;
end
```

KnnPE code, represented here by Figure 16 , takes in the result from the difference squared and handles the summing of each result coming from the ArithPE. As long as count is less than the total number of train vectors, both sum and count are updated. Once count is larger than total, the final piece in this snippet sets the done-bit high.

Figure 16. The KNN processing element that handles The sum

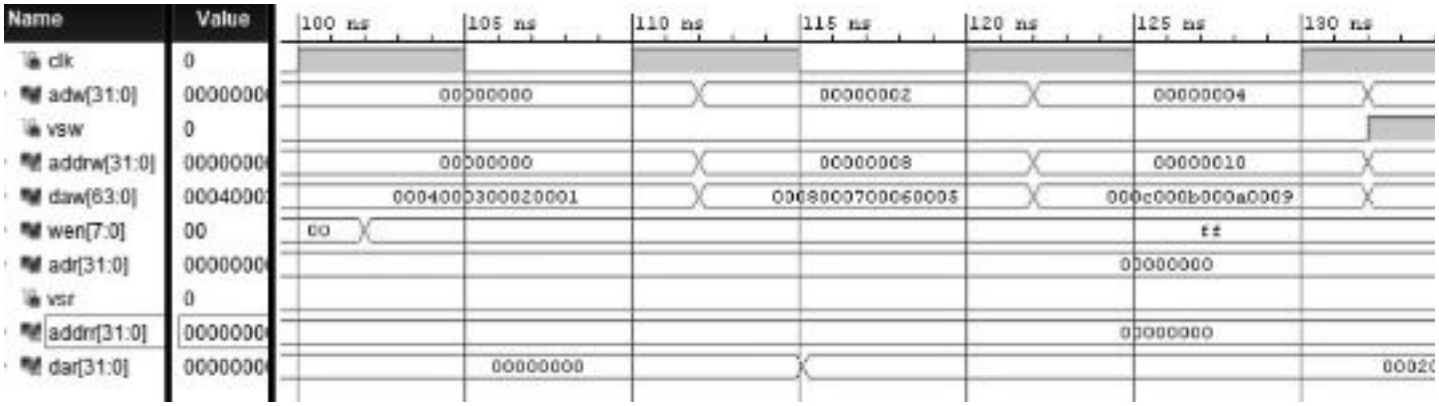
The final step in the modular design of the processing element is shown in Figure 17 which is the code for the top level array. This is where the kth nearest neighbor algorithm is used to find the smallest distances between the test and train vector as computed through Figure 15 and Figure 16. Tracking the status of the kth nearest neighbor algorithm means that particular block of code is waiting for the algorithm to run through all of the distances and finding the Kth smallest distances. Once it is finished, the done-bit is sent high and the results can be sent through the DMA back to the ARM which will end up back with the PC for the final sorting of the distances.

```
for (i=0; i<knns; i=i+1) begin
  for (j=0; j<k; j=j+1) begin
    if (near[2][k-1-j] > dist[i]) begin
      if (j>0) begin
        near[2][k-j] = near[2][k-1-j];
        value[2][k-j] = value[2][k-1-j];
      end
      near[2][k-1-j] = dist[i];
      value[2][k-1-j] = count + {8'b0,i};



//track status of knn algorithm
if ( (count+knns)<train_vectors ) begin
    count <= count + knns;
    done <= 1'b0;
end
else begin
    count <= 16'b0;
    done <= 1'b1;
```

Figure 17. KNN algorithm logic with done bit being sent high upon completion

Some issues were had synchronizing the data sent from the DMA with the KNN logic within the FPGA. As discussed previously, data was rearranged inside the DRAM before sending it to the FPGA, but this involved sending the same test data multiple times with each train vector. Another approach was attempted within the final stages of the project timeline. This was implementing BRAM directly with the FPGA and only sending the test vector once. This would create a buffer between sending data to the FPGA and reading data in the KNN logic.

For this application, the BRAM was configured to utilize 128 kB of data, or 32 k integers. The first half was designated for the test vector, and the second half acted as a buffer for all train vector data. A simple dual port design was used for the BRAM which allowed writing 64 bits to port A and reading 32 bits from port B. Writing 64 bits was chosen to match the DMA data width, and reading 32 bits allows reading one integer from the test vector, then reading one integer from the train vector. As an example, figure 18 shows the BRAM writing and reading data on the negative edge of the clock.



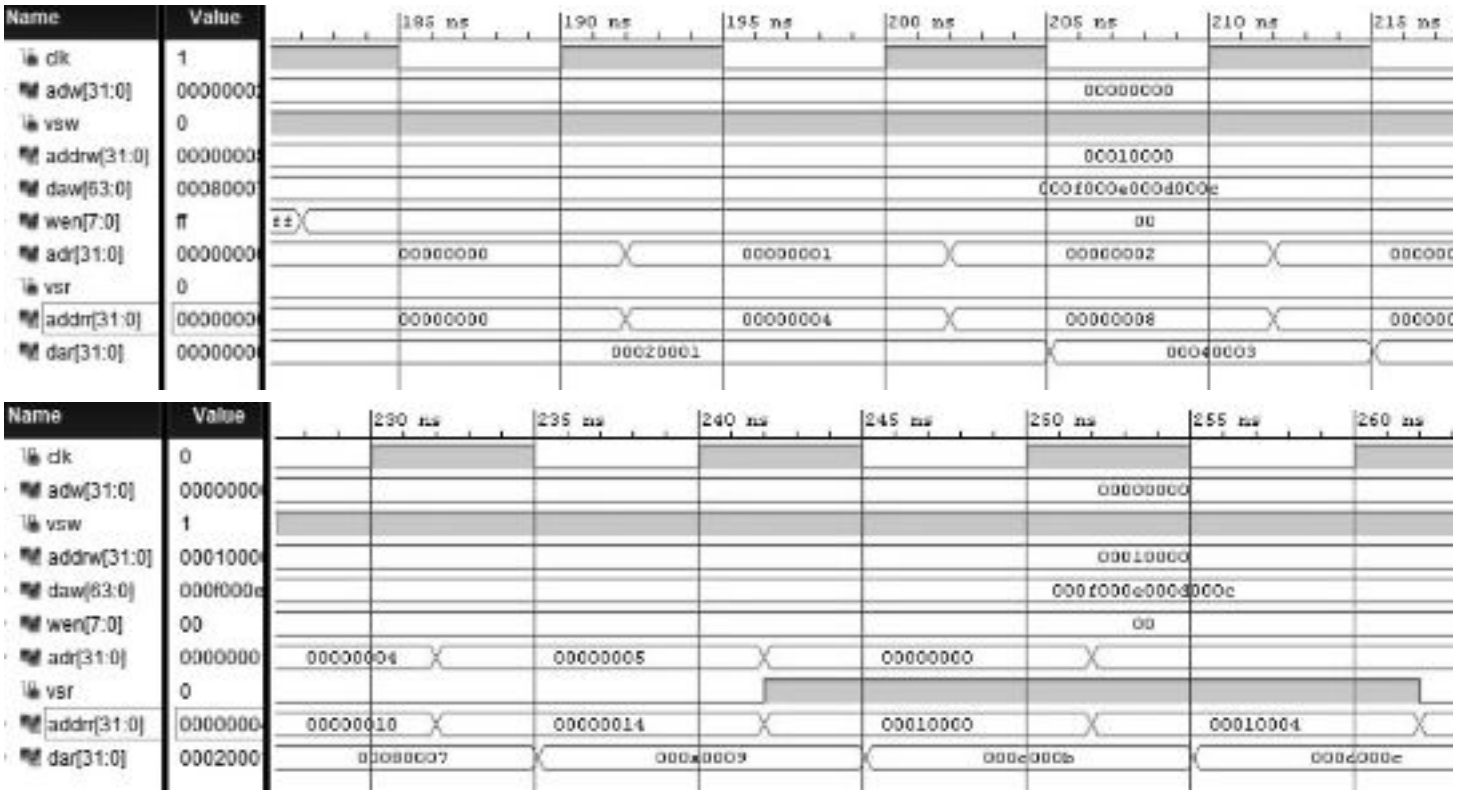


Figure 18. The write address and data are addrw and daw, and the read address and data are addrr and dar. The BRAM runs on the negative edge of the clock, and read operations have a two cycle delay before the data can be read. When write enable (wen) is 0xFF, data is being written to the respective address. Read is always enabled.

While writing and reading with the BRAM was proven to operate correctly, connecting it to the FPGA logic to manage incoming DMA data provided an extra level of difficulty for debugging. Without a hardware debugger, the four FPGA registers connected to the processor were used for debugging purposes. However, this method of debugging hardware with software was time consuming due to the bitstream generation each time a change was made.

Processor Results

The results on a single ARM processor contained on the Zybo is shown below in Figure 19. The first part of the figures below show the configuration settings for connection of PC to Zybo. This was important because it was one of the first parts that showed whether or not connection to the boards and all of the configurations were correct. If there was a board added, it could be seen here if the code was made aware that an additional board was accounted for by changing the variable handling the number of Zybos.

```
K:  4
Number of Zybos:  1
Total Train Vectors:  492
Number of Dimensions for each Vector:  10000
Ethernet Packet Size in Integers:  320
Number of Packets For Each Vector 32
Number of Train Vectors Each Board Will Receive:  492
Number of Packets Each Board Will be sent:  15776
creating thread 1
Connected to  192.168.1.11
Done sending to  192.168.1.11
Received packet from  192.168.1.11

---Results---
0 th nearest:  645242
1 th nearest:  647499
2 th nearest:  650686
3 th nearest:  651830
Total Time: 17.45236817038955
```

Figure 19. These are the nearest neighbor distance results from the on-board ARM processor as well as the time (in seconds) for the program to run with a single Zybo.

Similarly to Figure 19 above, Figure 20 below shows the configuration settings as well as the results for the kth nearest computations. The difference between the two is that Figure 20 shows the results for two Zybos instead of just a single board.

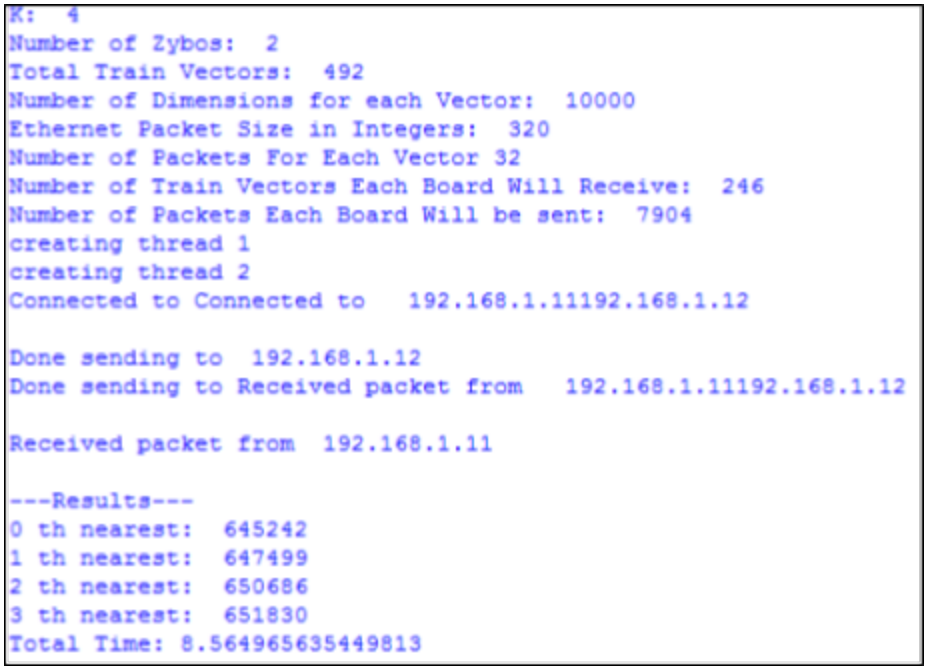


Figure 20. These are the nearest neighbor distance results from the on-board ARM processor as well as the time (in seconds) for the program to run with two Zybos.

When comparing the above figures to each other, it is seen that the time it takes for the program to execute utilizing the ARM processors on two Zybo boards for the same dataset shows the execution time is about 2.04 times faster. The tables in Figure 21 show the average running time in seconds for tests run on one through four Zybos using the ARM to do the computations in seconds.

Trial	1 zybo_processor
1	16.77
2	17.45
3	17.066
4	16.64
5	16.96
AVG Time	16.9772

Trial	2 zybo_processor
1	8.27
2	8.284
3	8.56
4	8.34
5	8.21
AVG Time	8.3328

Trial	3 zybo_processor
1	5.56
2	5.73
3	5.34
4	6.01
5	5.78
AVG Time	5.684

Trial	4 zybo_processor
1	4.204
2	4.24
3	4.31
4	4.21
5	4.25
AVG Time	4.2428

Figure 21. Execution time in seconds for the utilization of one, two three and four Zybo ARM processors.

As more Zybos are used, the timing results follow trends described by the law of diminishing returns. The execution time decrease from using one Zybo to using two Zybos was over two times faster, but the timing seen from three Zybos over two as well as four Zybos over

three is not as large. The execution time with three Zybos is seen as being 1.466 times faster than using two and the execution time with four Zybos is seen as being 1.34 times faster than using three. These tests were done to use as a comparison against the execution time of the kth nearest neighbor computations being done on the FPGA. The comparisons should show that using N number of FPGAs provides a shorter execution time over N number of ARM processors.

FPGA Results

Figure 22 below shows the execution time of the kth nearest neighbor computations being done with the developed FPGA logic. The average times were found for use of one, two, three and four FPGAs in terms of seconds.

Trial	1 zybo_FPGA
1	16.224
2	16.5144
3	16.1985
4	16.67
5	16.31
AVG Time	16.38338

Trial	2 zybo_FPGA
1	8.18
2	8.17
3	8.21
4	8.13
5	8.19
AVG Time	8.176

Trial	3 zybo_FPGA
1	5.46
2	5.45
3	5.41
4	5.39
5	5.5
AVG Time	5.442

Trial	4 zybo_FPGA
1	4.14
2	4.13
3	4.16
4	4.1
5	4.11
AVG Time	4.128

Figure 22. Average execution timings of the kth nearest neighbor computations (in seconds) on the FPGA logic.

In comparison to the ARM processors on the Zybo, the FPGA timing is faster, but just slightly. It averages about 0.2 seconds faster for the FPGA runs over the ARM processor runs. The major problem that provides a speedup that is as small as this has to do with the size of the FPGA. The ideal situation would have been to store the either the test or train vector, with all integers limited by their dimension, in a buffer on the FPGA. This was attempted but was unsuccessful. Further testing was done to implement a similar design using BRAM.

CONCLUSIONS

Reusability

One of the main goals of this project was to design this FPGA cluster to be reusable and flexible, meaning a future group or user could take any parallel algorithm and use it on the hardware accelerator. To do this, all that would need to be changed is the module in the block design that handles the algorithm. In the case of this implementation, it would be the module for the kth nearest neighbor algorithm seen in Figure 2. From there, as also mentioned earlier, the pieces of code that handle the configuration of the IP and MAC address as well as the port numbers are handled through conditional statements. So if only two boards were being used, the user would not have to comment out the setup for boards three and four.

Contributions

In the end, the group was able to develop a cluster prototype for a parallelized kth nearest neighbor systolic array implementation coded in Verilog. The client code, which was written in Python, was multithreaded to accommodate up to four Zybos. The Ethernet communications were created on the server side, having been written in C, which configures the Ethernet so the Zybo can send and receive data to and from the PC.

APPENDIX A

Client Code

```
import socket
import struct
import time
import math
import threading

#---GLOBAL VARIABLES---

#---User initialiazations---
file_name="bin_file.bin"
k=4
num_zybos=1
tot_num_train=100
dim_ints=10000
pkt_size_ints=320

#---Other variables needed---
train_vectors_per_board=int(tot_num_train/num_zybos)
pkts_per_vector=math.ceil(dim_ints/pkt_size_ints)
last_pkt_size=int(((dim_ints/pkt_size_ints)%1)*pkt_size_ints)
pkts_per_board=int(pkts_per_vector*(train_vectors_per_board+1))
snd_pkt_size_bytes=int(pkt_size_ints*4)
last_pkt_size_bytes=int(last_pkt_size*4)
rec_pkt_size_bytes=int(k*4)
dim_bytes=int(dim_ints*4)
#---Client sockets---
TCP_IP1 = '192.168.1.11'
TCP_PORT1=5
TCP_IP2 = '192.168.1.12'
TCP_PORT2=11
TCP_IP3 = '192.168.1.13'
TCP_PORT3=13
TCP_IP4 = '192.168.1.14'
TCP_PORT4=17
#---Offsets so every 3rd vector is sent to a board
offset1=0
offset2=1
offset3=2
offset4=3
#---Print individual boards kth nearest and overall kth nearest
kth_nearest=[]
#---Print parameters to make sure they match with Zybos---
print("K: ", k)
print("Number of Zybos: ", num_zybos)
print("Total Train Vectors: ", tot_num_train)
print("Number of Dimensions for each Vector: ", dim_ints)
print("Ethernet Packet Size in Integers: ", pkt_size_ints)
print("Number of Packets For Each Vector", pkts_per_vector)
print("Number of Train Vectors Each Board Will Receive: ",train_vectors_per_board)
print("Number of Packets Each Board Will be sent: ", pkts_per_board)



def run_program():
    #---Receives the individual boards kth nearest---
    kth_nearest1=[]
```

```
    kth_nearest2=[]
    kth_nearest3=[]
    kth_nearest4=[]
    start_time = time.clock()

    #---Threads to send packets and receive packets
    if num_zybos>0:
        print("creating thread 1")
        t1=threading.Thread(target=client_socket, args=(TCP_IP1, TCP_PORT1, offset1,
kth_nearest1))
    if num_zybos>1:
        print("creating thread 2")
        t2=threading.Thread(target=client_socket, args=(TCP_IP2, TCP_PORT2, offset2,
kth_nearest2))
    if num_zybos>2:
        print("creating thread 3")
        t3=threading.Thread(target=client_socket, args=(TCP_IP3, TCP_PORT3, offset3,
kth_nearest3))
    if num_zybos>3:
        print("creating thread 4")
        t4=threading.Thread(target=client_socket, args=(TCP_IP4, TCP_PORT4, offset4,
kth_nearest4))

    if num_zybos>0:
        t1.start()
    if num_zybos>1:
        t2.start()
    if num_zybos>2:
        t3.start()
    if num_zybos>3:
        t4.start()
    if num_zybos>0:
        t1.join()
    if num_zybos>1:
        t2.join()
    if num_zybos>2:
        t3.join()
    if num_zybos>3:
        t4.join()

    if num_zybos==1:
        kth_nearest=kth_nearest1
    if num_zybos==2:
        kth_nearest=sorted(kth_nearest1+kth_nearest2)
    if num_zybos==3:
        kth_nearest=sorted(kth_nearest1+kth_nearest2+kth_nearest3)
    if num_zybos==4:
        kth_nearest=sorted(kth_nearest1+kth_nearest2+kth_nearest3+kth_nearest4)

    print("\n---Results---")
    for x in range(0, k):
        print(x, "th nearest: ", kth_nearest[x])
    end_time = time.clock()
    print("Total Time:",(end_time-start_time))

def client_socket(TCP_IP, TCP_PORT, thread_offset, kth_nearest):
    #---Open binary file---
    file = open(file_name, "rb")
    #---Connect to client sockets---
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((TCP_IP, TCP_PORT))
```

```
    print("Connected to ", TCP_IP)
    #---Read file and send test vector---
    for i in range(0, pkts_per_vector):
        if i == pkts_per_vector-1:
            payload = file.read(last_pkt_size_bytes)
        else:
            payload = file.read(snd_pkt_size_bytes)
        client_socket.send(payload)
        time.sleep(1/10000000)
    #send the train vectors
    for i in range(0, train_vectors_per_board):
        file.seek(dim_bytes+i*num_zybos*dim_bytes+thread_offset*dim_bytes)
        for j in range(0, pkts_per_vector):
            if j == pkts_per_vector-1:
                payload = file.read(last_pkt_size_bytes)
            else:
                payload = file.read(snd_pkt_size_bytes)
            client_socket.send(payload)
            time.sleep(1/10000000)
    print("Done sending to ", TCP_IP)
    file.close()
    #---Receive packets---
    recpacket = client_socket.recv(rec_pkt_size_bytes)
    print("Received packet from ", TCP_IP)
    i=0
    j=3
    for x in range(0, k):
        kth_nearest.append(int.from_bytes(recpacket[i:j], byteorder='little'))
        i=i+4
        j=j+4

run_program()
```

Server Code

DMA.h

```
/*
 * dma.h
 *
 *  Created on: Oct 25, 2017
 *      Author: mcowley
 */

#ifndef SRC_DMA_H_
#define SRC_DMA_H_

#include "xaxidma.h"
#include "xparameters.h"
#include "xdebug.h"

/************************** Function Prototypes *****************************/
#if defined(XPAR_UARTNS550_0_BASEADDR)
static void Uart550_Setup(void);
#endif

void init_progam();

int init_dma2();
int RxSetup(XAxiDma * AxiDmaInstPtr);
int TxSetup(XAxiDma * AxiDmaInstPtr);
int SendPacket(u8 *TxPacket, int length);
int CheckData(void);
u8* getPacket();


#define BASE (int*)0x43C00000
#define INT1 *(BASE)
#define INT2 *(BASE + 1)
//assign done = (index*PACKET_WIDTH >= num_words);
#define INT3 *(BASE + 2) //prev. INDEX
#define INT4 *(BASE + 3) //prev. PACKET_WIDTH
#define DONE *(BASE + 4)
#define RESET *(BASE + 16)
#define PACKET_SIZE_REG *(BASE + 17)
#define TRAIN_VECTORS *(BASE + 18)
#define DEBUG *(BASE + 19)

#endif /* SRC_DMA_H_ */
```

ethernet_config.c

```
#include "ethernet.h"

/* missing declaration in lwIP */
void lwip_init();
#if LWIP_DHCP==1
extern volatile int dhcp_timoutcntr;
err_t dhcp_start(struct netif *netif);
#endif



void
print_ip(char *msg, struct ip_addr *ip)
{
	print(msg);
	xil_printf("%d.%d.%d.%d\n\r", ip4_addr1(ip), ip4_addr2(ip),
		ip4_addr3(ip), ip4_addr4(ip));
}

void
print_ip_settings(struct ip_addr *ip, struct ip_addr *mask, struct ip_addr *gw)
{

	print_ip("Board IP: ", ip);
	print_ip("Netmask : ", mask);
	print_ip("Gateway : ", gw);
}

#if defined (__arm__) && !defined (ARMR5)
#if XPAR_GIGE_PCS_PMA_SGMII_CORE_PRESENT == 1 ||
XPAR_GIGE_PCS_PMA_1000BASEX_CORE_PRESENT == 1
int ProgramSi5324(void);
int ProgramSfpPhy(void);
#endif
#endif

#ifdef XPS_BOARD_ZCU102
#ifdef XPAR_XIICPS_0_DEVICE_ID
int IicPhyReset(void);
#endif
#endif

int ethernet_config()
{
	xil_printf("\n--- Configuring Ethernet ---\n");
	struct ip_addr ipaddr, netmask, gw;
```

```
	/* the mac address of the board. this should be unique per board */
	unsigned char mac_ethernet_address[] =
	{ 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };

	echo_netif = &server_netif;
#if defined (__arm__) && !defined (ARMR5)
#if XPAR_GIGE_PCS_PMA_SGMII_CORE_PRESENT == 1 ||
XPAR_GIGE_PCS_PMA_1000BASEX_CORE_PRESENT == 1
	ProgramSi5324();
	ProgramSfpPhy();
#endif
#endif

/* Define this board specific macro in order perform PHY reset on ZCU102 */
#ifdef XPS_BOARD_ZCU102
	IicPhyReset();
#endif

	init_platform();

#if LWIP_DHCP==1
    ipaddr.addr = 0;
	gw.addr = 0;
	netmask.addr = 0;
#else
	//DOES NOT GET HERE
	/* initliaze IP addresses to be used */
	IP4_ADDR(&ipaddr,  192, 168,   1, 10);
	IP4_ADDR(&netmask, 255, 255, 255,  0);
	IP4_ADDR(&gw,      192, 168,   1,  1);
#endif

	lwip_init();

	/* Add network interface to the netif_list, and set it as default */
	if (!xemac_add(echo_netif, &ipaddr, &netmask,
                                    &gw, mac_ethernet_address,
                                    PLATFORM_EMAC_BASEADDR)) {
		xil_printf("Error adding N/W interface\n\r");
		return -1;
	}
	netif_set_default(echo_netif);

	/* now enable interrupts */
	platform_enable_interrupts();

	/* specify that the network if is up */
	netif_set_up(echo_netif);

#if (LWIP_DHCP==1)
```

```
	/* Create a new DHCP client for this interface.
	 * Note: you must call dhcp_fine_tmr() and dhcp_coarse_tmr() at
	 * the predefined regular intervals after starting the client.
	 */
	dhcp_start(echo_netif);
	dhcp_timoutcntr = 24;

	while(((echo_netif->ip_addr.addr) == 0) && (dhcp_timoutcntr > 0))
xemacif_input(echo_netif);

	if (dhcp_timoutcntr <= 0) {
		if ((echo_netif->ip_addr.addr) == 0) {

			IP4_ADDR(&(echo_netif->ip_addr),  192, 168,   1, 11); //Change
boards Ip address here.
			IP4_ADDR(&(echo_netif->netmask), 255, 255, 255,  0);
			IP4_ADDR(&(echo_netif->gw),      192, 168,   1,  1);
		}
	}

	ipaddr.addr = echo_netif->ip_addr.addr;
	gw.addr = echo_netif->gw.addr;
	netmask.addr = echo_netif->netmask.addr;
#endif

	print_ip_settings(&ipaddr, &netmask, &gw);

	/* start the application (web server, rxtest, txtest, etc..) */
	start_application();

	/* never reached */
	//cleanup_platform();
	return 0;
}
#include "lwip/dhcp.h"
```

ethernet.c

```
#include "ethernet.h"

extern volatile int TcpFastTmrFlag;
extern volatile int TcpSlowTmrFlag;

void ethernet_accept(){
	xil_printf("\n--- Waiting for connection ---\n");

	ACCEPTED_CONNECTION=0;
	/* wait for socket.connect() from python script */
	while (!ACCEPTED_CONNECTION) {
		if (TcpFastTmrFlag) {
```

```
                        tcp_fasttmr();
                        TcpFastTmrFlag = 0;
                }
                if (TcpSlowTmrFlag) {
                        tcp_slowtmr();
                        TcpSlowTmrFlag = 0;
                }
                xemacif_input(echo_netif);
        }
        xil_printf("TCP connection accepted\n");
}

void ethernet_send(int * kth_nearest, int k){
        xil_printf("\n--- Sending Ethernet packet ---\n");

        int i;
        struct pbuf *p;
        for(i=0; i<k; i++)
        {
                p->payload=malloc(sizeof(kth_nearest[i]));
                *(int *)p->payload=kth_nearest[i];
                p->len=sizeof(kth_nearest[i]);
                err_t err = tcp_write(TCP_PCB, p->payload, p->len, 1);
                //xil_printf("write err: %d\n", err);
                pbuf_free(p);
        }
        tcp_output(TCP_PCB); //send packet
        xil_printf("Done sending TCP packet\n");
}

void ethernet_receive(){
        ETH_PACKETS_RECEIVED=0;
                xil_printf("\nReceiving Ethernet Packets...\n");
                /* receive and process packets */
                while (NUM_ETH_PACKETS!=ETH_PACKETS_RECEIVED) {
                        if (TcpFastTmrFlag) {
                                tcp_fasttmr();
                                TcpFastTmrFlag = 0;
                        }
                        if (TcpSlowTmrFlag) {
                                tcp_slowtmr();
                                TcpSlowTmrFlag = 0;
                        }
                        xemacif_input(echo_netif);
                }
                xil_printf("Done Receiving Ethernet Packets\n\n");
}

err_t recv_callback(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err)
{
```

```
	/* indicate that the packet has been received */
	tcp_recved(tpcb, p->len);
	/* copy payload into char buff */
	memcpy(VECTORS+BUF_OFFSET,p->payload, p->len);
	BUF_OFFSET=BUF_OFFSET+p->len;
	ETH_PACKETS_RECEIVED++;
	pbuf_free(p);
	return ERR_OK;
}

err_t accept_callback(void *arg, struct tcp_pcb *newpcb, err_t err)
{
	static int connection = 1;

	/* set the receive callback for this connection */
	tcp_recv(newpcb, recv_callback);
	/* just use an integer number indicating the connection id as the
	   callback argument */
	tcp_arg(newpcb, (void*)(UINTPTR)connection);
	/* increment for subsequent accepted connections */
	connection++;
	TCP_PCB=newpcb;
	ACCEPTED_CONNECTION=1;
	return ERR_OK;
}


int start_application()
{
	err_t err;
	//.11 5 02, .12 11 03, .13 13 04

	struct tcp_pcb *pcb;
	/* create new TCP PCB structure */
	pcb = tcp_new();
	if (!pcb) {
		xil_printf("Error creating PCB. Out of Memory\n\r");
		return -1;
	}

	err = tcp_bind(pcb, IP_ADDR_ANY, port);
	if (err != ERR_OK) {
		xil_printf("Unable to bind to port %d: err = %d\n\r", port, err);
		return -2;
	}

	/* we do not need any arguments to callback functions */
	tcp_arg(pcb, NULL);
```

```
	/* listen for connections */
	pcb = tcp_listen(pcb);
	if (!pcb) {
		xil_printf("Out of memory while tcp_listen\n\r");
		return -3;
	}

	/* specify callback to use for incoming connections */
	tcp_accept(pcb, accept_callback);

	return 0;
}
```

main.c

```
#include "dma.h"
#include "ethernet.h"
void merge(int * arr, int l, int m, int r);
void mergeSort(int * arr, int l, int r);

int main(void){
	printf("\r\n--- Entering main() --- \r\n");

	Xil_DCacheDisable();
	Xil_L2CacheDisable();

	/*---User initializations*/
	int k = 4;
	int num_zybos=1;
	int tot_num_train = 100;
	int dim_ints = 10000;
	int pkt_size_ints=320;

	/*---Choose which IP address, port, and mac address to make this board*/
	int Zybo = 1;
	mac_ethernet_address=malloc(6);
	if(Zybo==1){
		IP=11;
		port=5;
		mac_ethernet_address[5]=0x02;
	}
	else if(Zybo==2){
		IP=12;
		port=11;
		mac_ethernet_address[5]=0x03;
	}
	else if(Zybo==3){
		IP=13;
		port=13;
		mac_ethernet_address[5]=0x04;
	}
	else{
		IP=14;
		port=17;
		mac_ethernet_address[5]=0x05;
	}
```

```
	/*Other Variables Needed*/
	int num_train=tot_num_train/num_zybos;
	int * kth_nearest = malloc(sizeof(int)*k);
	//to do ceil without function
	int remainder=(dim_ints/pkt_size_ints)%1;
	int pkts_per_vector=dim_ints/pkt_size_ints-remainder+1;
	NUM_ETH_PACKETS = pkts_per_vector*(num_train+1);
	ETH_PACKETS_RECEIVED=0;
	BUF_OFFSET=0;
	VECTORS=malloc(sizeof(int)*(num_train+1)*dim_ints);
	int dma_packet_buffer_size=sizeof(int)*dim_ints*2;
	int * dma_packet_buffer  = malloc(dma_packet_buffer_size);
	int * distances = malloc(sizeof(int)*num_train);
	int i, j;

	/*---Print parameters to make sure they match the python script---*/
	printf("K: %i\n",k);
	printf("Number of Zybos: %i\n",num_zybos);
	printf("Total Train Vectors: %i\n", tot_num_train);
	printf("Number of Dimensions For Each Vector: %i\n",dim_ints);
	printf("Ethernet Packet Size in Integers: %i\n",pkt_size_ints);
	printf("Number of Packets For Each Vector: %i\n", pkts_per_vector);
	printf("Number of Train Vectors Each Board Will Receive: %i\n",num_train);
	printf("Number of Packets Each Board Will Be Sent: %i\n", NUM_ETH_PACKETS);


	/*set ip address and ip address that it can accept*/
	ethernet_config();
	/*wait to accept*/
	ethernet_accept();
	/*receive ethernet packets*/
	ethernet_receive();

	/*
	for(i=0; i<num_train; i++){
		for(j=0;j<dim_ints;j++){
			if(*((int*)(VECTORS)+j)>20 || *((int*)(VECTORS)+j)<-20)
printf("test[%i] is wrong: %i\n", j, *((int*)(VECTORS)+j));
			if(*((int*)(VECTORS)+dim_ints+dim_ints*i+j)>20 ||
*((int*)(VECTORS)+dim_ints+dim_ints*i+j)<-20) printf("train[%i] is wrong: %i\n", j,
*((int*)(VECTORS)+dim_ints+dim_ints*i+j));
		}
	}*/
	PACKET_SIZE_REG = dim_ints;
	TRAIN_VECTORS = num_train;
	DEBUG = 0;
	RESET = 0xFFFFFFFF;
	RESET = 0x00000000;
	printf("Calculating Distances with FPGA...\n");
	u8 *packet = (u8*)dma_packet_buffer;//VECTORS;

	// BRAM data send implementation
//	int num_bytes		= sizeof(int)*dim_ints;
//	int num_dma_packets = tot_num_train+1;
//
//	for (int i = 0; i < num_dma_packets; ++i)
//	{
//		init_dma2();
//		SendPacket((packet + i*num_bytes), num_bytes);
//	}
```

```
	int m;
	m=0;
	for(j=0;j<dim_ints;j++){
		dma_packet_buffer[m]=*((int*)(VECTORS)+j);
		m=m+2;
	}

	for(i=0; i<num_train; i++){
		m=0;
		for(j=0;j<dim_ints;j++){
			dma_packet_buffer[m+1]=*((int*)(VECTORS)+dim_ints+dim_ints*i+j);
			//if(dma_packet_buffer[m+1]!= i*dim_ints+j+dim_ints)
printf("train[%i]: %i\n", i*dim_ints+j,dma_packet_buffer[m+1]);
			m=m+2;
		}
		init_dma2();
		printf("%d:\t\t%d\t%d\t%d\t%d\n", i, INT1, INT2, INT3, INT4);
		SendPacket((packet), dma_packet_buffer_size);
		sleep(1);
		//test sending data and reading back with FIFO and very small arrays (10
elements)
	}

	printf("\n");
	free(dma_packet_buffer);
	printf("Calculating Distances with Processor...\n");
		for(i=0; i<num_train; i++){
			distances[i]=0;
			for(j=0;j<dim_ints;j++){

distances[i]=distances[i]+((*((int*)(VECTORS)+j)-*((int*)(VECTORS)+dim_ints+dim_ints*i
+j))*(*((int*)(VECTORS)+j)-*((int*)(VECTORS)+dim_ints+dim_ints*i+j)));
			}
		}
		mergeSort(distances, 0, num_train-1);

	free(VECTORS);
	while(!DONE);
	printf("\nDone calculating distances with FPGA\n");

	printf("\n--- This boards results: ---\n");
	if(k>0){
		printf("1st nearest: %i\n",INT1);
		kth_nearest[0]=INT1;
	}
	if(k>1){
		printf("2nd nearest: %i\n",INT2);
		kth_nearest[1]=INT2;
	}
	if(k>2){
		printf("3rd nearest: %i\n",INT3);
		kth_nearest[2]=INT3;
	}
	if(k>3){
		printf("4th nearest: %i\n\n",INT4);
		kth_nearest[3]=INT4;
	}
	RESET = 0xFFFFFFFF;
	printf("\nDone calculating distances with Processor\n\n");
	for(i=0;i<k;i++) printf("k nearest %i from processor: %i\n",i,distances[i]);
	/*send data back to the PC*/
```

```
	ethernet_send(kth_nearest, k);
	xil_printf("\n--- Exiting main() --- \r\n");
	return 0;
}

void mergeSort(int * arr, int l, int r){
	if (l < r){
		int m = l+(r-l)/2;
		mergeSort(arr, l, m);
		mergeSort(arr, m+1, r);
		merge(arr, l, m, r);
	}
}

void merge(int arr[], int l, int m, int r){
	int i, j, k;
	int n1 = m - l + 1;
	int n2 =  r - m;
	int L[n1], R[n2];

	for (i = 0; i < n1; i++) L[i] = arr[l + i];
	for (j = 0; j < n2; j++) R[j] = arr[m + 1+ j];

	i = 0;
	j = 0;
	k = l;
	while (i < n1 && j < n2){
		if (L[i] <= R[j]){
			arr[k] = L[i];
			i++;
		}
		else{
			arr[k] = R[j];
			j++;
		}
		k++;
	}

	while (i < n1){
		arr[k] = L[i];
		i++;
		k++;
	}

	while (j < n2){
		arr[k] = R[j];
		j++;
		k++;
	}
}
```

FPGA Code

KNN_stream_v1_0.v

```
	module KNN_stream_v1_0 #
	(
		// Users to add parameters here
	 parameter i = 2,    //int datainut at a time
	 parameter d = 10000,    //dimensions
//	   parameter t = 507,    //training vectors
	 parameter k = 4,    //nearest neighbors
	 parameter s = 1,    //APEs per KPE
	 parameter p = d/s,  //iterations per KPE
	 parameter z = i/s - 1,  //KPEs
	 parameter o = 4,    //int output at once
	 // User parameters ends
		// Do not modify the parameters beyond this line


		// Parameters of Axi Slave Bus Interface S00_AXIS
		parameter integer C_S00_AXIS_TDATA_WIDTH	= 64,
	 parameter integer NUMBER_OF_INPUT_WORDS = 5000
	)
	(
		// Users to add ports here
		input wire reset,
		input wire [31:0] packet_size,
	 input wire [31:0] train_vectors,
	 input wire [31:0] debug,

		output wire done,
	 output wire [31:0] int_1,
	 output wire [31:0] int_2,
	 output wire [31:0] int_3,
	 output wire [31:0] int_4,

	 // BRAM ports
	 output wire [31:0] addra,
	 output wire [63:0] dwr,
	 output wire [7:0] wea,
	 output wire [31:0] addrb,
	 input wire [31:0] drd,
		// User ports ends
		// Do not modify the ports beyond this line

		// Ports of Axi Slave Bus Interface S00_AXIS
		input wire  s00_axis_aclk,
		input wire  s00_axis_aresetn,
		output wire  s00_axis_tready,
		input wire [C_S00_AXIS_TDATA_WIDTH-1 : 0] s00_axis_tdata,
		input wire [(C_S00_AXIS_TDATA_WIDTH/8)-1 : 0] s00_axis_tstrb,
		input wire  s00_axis_tlast,
		input wire  s00_axis_tvalid
	);

	assign dwr = s00_axis_tdata; //send DMA data to BRAM

	localparam PACKET_WIDTH = C_S00_AXIS_TDATA_WIDTH >> 5;

	// Add user logic here
```

```
       wire knn_en, index, cycles;
       wire [31:0] latency;
    wire [(32*o - 1):0] dataout;
    //wire [(16*o - 1):0] valout;
    wire [(32*i - 1):0] datain;
    wire [31:0] testd;
    wire [31:0] traind;
    wire clk;
    reg [2:0] clkcnt = 2'b0;

    wire [31:0] valout;
    wire [127:0] echo;
//    wire [31:0] techo;

    //use 2bit counter to make systolic clock 4 times slower
    always @ (posedge s00_axis_aclk) begin
        if (!s00_axis_aresetn)
            clkcnt <= 3'b0;
        else if (clk || (clkcnt > 0))
            clkcnt <= clkcnt + 3'b01;
    end

    //stop knn array when done or not enabled
    assign clk = (clkcnt > 0) ? 1'b1 : knn_en & (~done) & s00_axis_aclk;
    //move data from DMA regs to KNN arrays
    assign datain = {testd, traind};

    assign int_1 = echo[31:0];//dataout[31:0];
    assign int_2 = echo[63:32];//dataout[63:32];
    assign int_3 = echo[95:64];//dataout[95:64];
    assign int_4 = echo[127:96];//valout;//dataout[127:96];

//    initial x <= 32'b0;
//    always @ (posedge s00_axis_aclk) x <= x+32'b1;

    //Dan's KNN Systollic Array
    TopLevelArray #(.k(k),
                    .knn_size(s),
//                    .train_vectors(t),
                    .dimensions(d),
                    .total(p),
                    .datain_size(i),
                    .knns(i-1),
                    .dataout_size(o) )
                tla(.clk(clk),
                    .datain(datain),
                    .train_vectors(train_vectors),
                    .debug(debug),
                    .echo(echo),
                    .done(done),
                    .dataout(dataout),
                    .valout(valout) );

       my_S00_AXIS # (
             .C_S_AXIS_TDATA_WIDTH(C_S00_AXIS_TDATA_WIDTH),
         .NUMBER_OF_INPUT_WORDS(NUMBER_OF_INPUT_WORDS)
       ) sum_stream_v1_0_S00_AXIS_inst (
             .S_AXIS_ACLK(s00_axis_aclk),
             .S_AXIS_ARESETN(s00_axis_aresetn),
             .S_AXIS_TREADY(s00_axis_tready),
             .S_AXIS_TDATA(s00_axis_tdata),
```

```
              .S_AXIS_TSTRB(s00_axis_tstrb),
              .S_AXIS_TLAST(s00_axis_tlast),
              .S_AXIS_TVALID(s00_axis_tvalid),

         .addra(addra),
         .dwr(),
         .wea(wea),
         .addrb(addrb),
         .drd(drd),

              .reset(reset),
//            .NUMBER_OF_INPUT_WORDS(packet_size),
//            .debug(debug),
//            .echo(techo),
              .KNN_en(knn_en),
              .int_1(testd),
         .int_2(traind)
       );

endmodule
```

KNN_stream_v1_0_S00_AXIS.v

```
`timescale 1 ns / 1 ps

       module my_S00_AXIS #
       (
              // Users to add parameters here

              // User parameters ends
              // Do not modify the parameters beyond this line

              // AXI4Stream sink: Data Width
              parameter integer C_S_AXIS_TDATA_WIDTH  = 64,
              parameter integer NUMBER_OF_INPUT_WORDS = 5000
       )
       (
              // Users to add ports here
              input wire reset,
//            input wire [31:0] NUMBER_OF_INPUT_WORDS,
//            input wire [31:0] debug,
//        output reg [31:0] echo,
              output reg KNN_en,
              // output reg [31:0] sum,
              //output reg [31:0] test_index,
         output reg [31:0] int_1,
         output reg [31:0] int_2,
         output reg [31:0] int_3,
         output reg [31:0] int_4,

         // BRAM ports
         output wire [31:0] addra,
         output wire [63:0] dwr,
         output reg [7:0] wea,
         output wire [31:0] addrb,
         input wire [31:0] drd,
              // User ports ends
              // Do not modify the ports beyond this line
```

```
            // AXI4Stream sink: Clock
            input wire  S_AXIS_ACLK,
            // AXI4Stream sink: Reset
            input wire  S_AXIS_ARESETN,
            // Ready to accept data in
            output wire  S_AXIS_TREADY,
            // Data in
            input wire [C_S_AXIS_TDATA_WIDTH-1 : 0] S_AXIS_TDATA,
            // Byte qualifier
            input wire [(C_S_AXIS_TDATA_WIDTH/8)-1 : 0] S_AXIS_TSTRB,
            // Indicates boundary of last packet
            input wire  S_AXIS_TLAST,
            // Data is in valid
            input wire  S_AXIS_TVALID
      );

      reg vsw;   //select test/train vector on write, managed in first always block
    reg vsr;   //select test/train vector on read
    reg [15:0] inr;   //index to read from

      // Define the states of state machine
      // The control state machine oversees the writing of input streaming data to
the FIFO,
      // and outputs the streaming data from the FIFO
      localparam [1:0] IDLE = 2'b00,        // This is the initial/idle state

                       WRITE_FIFO  = 2'b01,  // In this state FIFO is written with the
                                             // input stream data S_AXIS_TDATA

                   ERROR = 2'b10;             // this state is reached when the input
does not
                                                     // match expectations.
      wire   axis_tready;
      // State variable
      reg [1:0] mst_exec_state;

      reg [15:0] index;
      reg rst;

      wire fifo_wren;

      // sink has accepted all the streaming data and stored in FIFO
      reg writes_done;
      // I/O Connections assignments

      assign S_AXIS_TREADY      = axis_tready;
      // Control state machine implementation
      always @(posedge S_AXIS_ACLK)
      begin
        if (!S_AXIS_ARESETN || reset)
        // Synchronous reset (active low)
          begin
            mst_exec_state <= IDLE;
         rst <= 1'b1;

         vsw <= 1'b0;  //write to test vector in BRAM
          end
        else
          case (mst_exec_state)
            IDLE:
```

```
              // The sink starts accepting tdata when
              // there tvalid is asserted to mark the
              // presence of valid streaming data
                if (S_AXIS_TVALID && (index <= NUMBER_OF_INPUT_WORDS-1))
                  begin
                    mst_exec_state <= WRITE_FIFO;
                    rst <= 1'b0;
                  end
                else
                  begin
                    mst_exec_state <= IDLE;
                    rst <= 1'b0;
                  end
             WRITE_FIFO:
              // When the sink has accepted all the streaming input data,
              // the interface swiches functionality to a streaming master
              if (writes_done) begin
                  mst_exec_state <= IDLE;
                  rst <= 1'b1;

                  vsw <= 1'b1;   //write to train vector in BRAM
            end
               else
                 begin
                    mst_exec_state <= WRITE_FIFO;
                 end
           ERROR: begin
              // an error has occured
              mst_exec_state <= ERROR;
           end
            endcase
       end
       // AXI Streaming Sink
       //
       // The example design sink is always ready to accept the S_AXIS_TDATA  until
       // the FIFO is not filled with NUMBER_OF_INPUT_WORDS number of input words.
       assign axis_tready = ((mst_exec_state == WRITE_FIFO) && (index <=
NUMBER_OF_INPUT_WORDS-1));

       always@(posedge S_AXIS_ACLK)
       begin
         if(!S_AXIS_ARESETN || reset || rst)
           begin
             index <= 0;
             writes_done <= 1'b0;
           end
         else begin
              //KNN_en <= 1'b0;
           if (index <= NUMBER_OF_INPUT_WORDS-1)
             begin
               if (fifo_wren)
                 begin
                   // write pointer is incremented after every write to the FIFO
                   // when FIFO write signal is enabled.
                   index <= index + 1;
                   writes_done <= 1'b0;
                   // sum <= sum + S_AXIS_TDATA[31:0] + S_AXIS_TDATA[63:32];
                   //test_index <= test_index + 32'd1;
                   //KNN_en <= 1'b1;
                 end
               if ((index == NUMBER_OF_INPUT_WORDS-1)|| S_AXIS_TLAST)
```

```
                  begin
                    // reads_done is asserted when NUMBER_OF_INPUT_WORDS numbers of
streaming data
                    // has been written to the FIFO which is also marked by
S_AXIS_TLAST(kept for optional usage).
                    writes_done <= 1'b1;
                  end
             end
      end
       end

       // FIFO write enable generation
       assign fifo_wren = S_AXIS_TVALID && axis_tready;


            // Streaming input data is stored in FIFO

            always @( negedge S_AXIS_ACLK )
            begin
              if (fifo_wren)// && S_AXIS_TSTRB[byte_index])
                begin
                 wea <= 8'hFF;
                end
            else
              wea <= 8'h00;
            end

// Add user logic here

       // BRAM addressing
       assign addra = {vsw, index[12:0], 3'b0};
       assign addrb = {vsr, inr[13:0], 2'b0};
       reg [1:0] rdst;

       // Update read state and knn_en
       always @ (posedge S_AXIS_ACLK) begin
       if (reset) begin
              rdst <= 2'b0;
              KNN_en <= 1'b0;
       end
       else if (inr[15:1] == index) begin
           KNN_en <= 1'b0;
       end
       else begin
           if (rdst == 2'b10)
               int_1 <= drd;
           else if (rdst == 2'b11) begin
               int_2 <= drd;
               KNN_en <= 1'b1;
           end
           else
               KNN_en <= 1'b0;

           rdst <= rdst + 2'b01;
       end
       end

       // Update read address based on read state
       always @ (negedge S_AXIS_ACLK) begin
          if (reset) begin
              inr <= 0;
```

```
            vsr <= 0;
        end
        else begin
            if (rdst == 2'b01)
                vsr <= 1'b1;
            else
                vsr <= 1'b0;

            if (rdst == 2'b11)
                inr <= (inr+1)%(NUMBER_OF_INPUT_WORDS<<1);
        end
        end

endmodule
```

TopLevelArray.v

```
/* Change parameters to fit application
This top-level module performs the knn algorithm on a set of input
train vectors and train vector.  Datain is a combination of test and
train dimensions.  The most significant value is the test dim, the
remaining are the train dims counting up towards the least significant.
Each dimension is a 32bit integer.  3 lists are used to store the knn
values.  The first two are cycled such that one can be adjusted while
the other is sequentially output. The third is used to insertion sort
and update the origional lists.  Both distance(near) and id(value) are
kept in a list for functional verification.
*/

module TopLevelArray
#(
parameter k = 8,                //nearest neighbors
parameter knn_size = 1,         //data in size for knn modules, must be 1 (1 test dim
each cycle)
//parameter train_vectors = 6,    //vectors to compare
parameter dimensions = 8,       //dimensions in each vector
parameter total = dimensions / knn_size,//total number of dim for each vector / input
data for each knn
parameter datain_size = 4,      //values input at a time
parameter knns = datain_size - 1,//knnPEs generated
parameter dataout_size = 1     //values output at a time, output one every cycle
)(
input clk,
input [(32*datain_size - 1):0] datain,
input [31:0] train_vectors,
input [31:0] debug,
output [127:0] echo,
output reg done,
output [(32*dataout_size - 1):0] dataout,
output reg [31:0] valout//[(16*dataout_size - 1):0] valout //should nt be reg
);

    wire [(knn_size * 32 - 1):0] test_data;
    wire [(knn_size * 32 - 1):0] train_data [0:(knns-1)];

    //3 of the near and value arrays are made, first 2 to rotate each calculation
cycle, last as a temp
    (*DONT_TOUCH="true"*)reg [31:0] near [0:2][0:(k-1)];   //array for k nearest
distances
    (*DONT_TOUCH="true"*)reg [31:0] nout [0:(dataout_size-1)];   //array to output
```

```
    (*DONT_TOUCH="true"*)reg [15:0] value [0:2][0:(k-1)];  //array for k nearest
counts
    (*DONT_TOUCH="true"*)reg [15:0] vout [0:(dataout_size-1)];   //inex to output

    wire pedone [0:(knns-1)];       //done bits for each knnPE
    wire [31:0] dist [0:(knns-1)];  //distance from each knn

    reg [7:0] h, i, j, w, x, y;       //counters
    reg [15:0] count;            //count up train vectors(*MARK_DEBUG="true"*)
    (*DONT_TOUCH="true"*)reg ksel;                   //select knn lists to insert and
display

    //assign valout = count;
    //assign valout = dist[0];

    genvar n;
    generate
        for (n=0; n<knns; n=n+1) begin
            //create knnPEs
            KnnPE #(.size(knn_size),.total(total))
knn(.clk(clk),.test_data(test_data),.train_data(train_data[n]),.debug(debug),.echo(ech
o),.done(pedone[n]),.dist(dist[n]));
        end

        //split test and train data from input
        assign test_data = datain[(32*datain_size-1):(32*(datain_size-1))];
        for (n=0; n<knns; n=n+1) begin
            assign train_data[n] = datain[((knns-n)*32-1):(32*(knns-n-1))];
        end

        //join distance and id for output
        for (n=0; n<dataout_size; n=n+1) begin
            assign dataout[(32*n+31):(32*n)] = nout[n];
//            assign valout[(16*n+15):(16*n)] = vout[n];
        end
    endgenerate

    initial begin
        //initialize regs
        for (i=0; i<k; i=i+1) begin
            near[0][i] <= 32'hffffffff;
            value[0][i] <= 16'hffff;
            near[1][i] <= 32'hffffffff;
            value[1][i] <= 16'hffff;
            near[2][i] <= 32'hffffffff;
            value[2][i] <= 16'hffff;
        end
        count <= 16'b0;
        done <= 1'b0;
        y <= 8'b0;
        ksel = 1'b0;
        valout <= 0; //delete this
    end

    always @ (posedge clk) begin
        //update knn values in temp list
        if (pedone[0]) begin
            if (count==1)  //delete these 2 lines
                valout <= dist[0];
            if (done) begin //reset values
                for (h=0; h<k; h=h+1) begin
```

```
                    near[2][h] = 32'hffffffff;
                    value[2][h] = 16'hffff;
                end
            end
            for (i=0; i<knns; i=i+1) begin
                for (j=0; j<k; j=j+1) begin
                    if (near[2][k-1-j] > dist[i]) begin
                        if (j>0) begin
                            near[2][k-j] = near[2][k-1-j];
                            value[2][k-j] = value[2][k-1-j];
                        end
                        near[2][k-1-j] = dist[i];
                        value[2][k-1-j] = count + {8'b0,i};
                    end
                end
            end
        end

    end

    always @ (negedge clk) begin
        if (pedone[0]) begin
            //logic to update nearest neighbor array
            for (w=0; w<k; w=w+1) begin
                    near[ksel][w] <= near[2][w];
                    value[ksel][w] <= value[2][w];
            end

            //track status of knn algorithm
            if ( (count+knns)<train_vectors ) begin
                count <= count + knns;
                done <= 1'b0;
            end
            else begin
                count <= 16'b0;
                done <= 1'b1;
                ksel <= ~ksel;  //switch knn lists
            end
        end

        //output dataout_size data at a time when done
        if ( y>0 || (pedone[0] && (count+knns)>=train_vectors) ) begin
            for (x=0; x<dataout_size; x=x+1) begin
                nout[x] <= near[(y==0)? 2 : ~ksel][y+x];
                vout[x] <= value[(y==0)? 2 : ~ksel][y+x];
            end
            y <= (y + dataout_size)%k; //use to count data output
        end
        else begin
            for (x=0; x<dataout_size; x=x+1) begin
                nout[x] <= 32'bX;
                vout[x] <= 16'bX;
            end
            y <= 8'b0;
        end
    end

endmodule
```

KnnPE.v

```
//Each KnnPE module handles the complete comparison of two vectors.
//The dimension data of each vector are input {size} elements at a time.
//This is repeated {total} times until the calculation is complete.

module KnnPE
#(
parameter size=1,       //number of comparisons completed each cycle
parameter total=10000   //number of cycles before done
)(
input clk,
input [(size * 32 - 1):0] test_data,
input [(size * 32 - 1):0] train_data,
input [31:0] debug,
output [127:0] echo,
output reg done,
output [31:0] dist  //sum of all comparisons
);

    wire [31:0] test [0:(size-1)];  //split input test data
    wire [31:0] train [0:(size-1)]; //split input train data
    reg [31:0] sum;                 //running sum of comparisons
    reg [15:0] count;               //count cycles up to total
    wire [31:0] sqrout [0:(size-1)];//comparison out from each APE
    wire [31:0] sqrsum [0:(size-1)];//cumulative sum of all outputs

    reg [31:0] echor [0:3];
    assign echo = {echor[3], echor[2], echor[1], echor[0]};

    genvar n;
    generate    //connect nets as per above definitions
        for (n=0; n<size; n=n+1) begin
            assign test[n] = test_data[(n*32+31):(n*32)];
            assign train[n] = train_data[(n*32+31):(n*32)];
            ArithPE pe(test[n], train[n], sqrout[n]);

            if (n == 0) begin
                assign sqrsum[0] = sqrout[0];
            end
            else begin
                assign sqrsum[n] = sqrsum[n-1] + sqrout[n];
            end
        end
    endgenerate

    //update output in real time
    assign dist = sum + sqrsum[size-1];

    initial begin
        sum <= 0;
        count <= 0;
    end

    always @ (posedge clk) begin
        //if reached total, reset
        if (count >= total) begin
            sum <= sqrsum[size-1];
            count <= 1;
        end
        //update sum and count
```

```
        else begin
            sum <= sum + sqrsum[size-1];
            count <= count + 16'b1;
        end

        case (count)
            (debug): begin
                echor[0] <= test[0];
                echor[1] <= train[0];
            end
            (debug+1): begin
                echor[2] <= test[0];
                echor[3] <= train[0];
            end
//            16'b10: echor[2] <= train[0];
//            16'b11: echor[3] <= train[0];
//            default: begin
//                echor[0] <= echor[0];
//                echor[1] <= echor[1];
//                echor[2] <= echor[2];
//                echor[3] <= echor[3];
//            end
        endcase
    end

    //preemtively set done on negedge
    always @ (negedge clk) begin
        if (count >= total-16'b1) begin
            done <= 1'b1;
        end
        else
            done <= 1'b0;
    end

endmodule
```

ArithPE.v

```
//This module takes two 32 bit values, and returns the difference squared 20ns after x
& y are given
module ArithPE(
    input [31:0] x,
    input [31:0] y,
    output [31:0] z
    );

    wire [31:0] diff;

    assign diff = x - y;

    assign z = diff * diff;
endmodule
```