Instituto Superior Técnico

HW/SW Co-Design

MEEC

k-NN HW Acceleration using a Zybo Board - P1

Authors:
André Sousa nº **65313**
João Borrego nº **78990**

Group:
15

May 7, 2017

1 k-Nearest Neighbours algorithm

The k-Nearest Neighbours algorithm, or simply k-NN, is a somewhat intuitive machine learning technique used for both classification and regression problems. In this project we focused on accelerating k-NN classification, using a Zybo[1] board, combining the raw processing power of a generic ARM CPU and purposely designed hardware.

The core idea of this algorithm is to compare each testing object with the whole training set, and assign a label by majority rule of its k closest neighbours. Thus, one must first define a dissimilarity measure. The euclidean distance in M-D, with M the number of features, is a fairly common choice, and it was used in this project, due to its simplicity and effectiveness. It is defined in Equation 1.1.

$$d = \sqrt{\sum_i (y_i - x_i)^2} \tag{1.1}$$

The classification label for a given test object is decided by the majority vote of its k-nearest neighbours, i.e., the training objects that correspond to the k smaller distances. This allows for highly non-linear decision boundaries, resulting in a rather powerful classification algorithm.

The computation of distances between testing and training feature vectors is a repetitive procedure that takes most of the program execution. This looks promising for hardware acceleration, since this simple operation can be performed in custom designed hardware and hopefully result in decent speed-ups. One should mention that the outer square root is not only an expensive operation, requiring several clock cycles to be finished, but also unnecessary, since the squared euclidean distances from a test object to all training objects retain the same ordering.

For this stage of the project, we used the Iris flower data set[2], which contains four relevant measurements of 50 samples from each of three different flower species, namely *Iris setosa*, *Iris virginica* and *Iris versicolor*. We chose a 2/3 training/test split, i.e., 100 samples for training and 50 for testing.

2 Hardware Design

We designed and implemented a custom hardware IP core that receives a testing object feature vector and stores it in internal BRAM memory. This helps reduce communications and ensures maximum data re-usability. It then reads a training set object and calculates the squared euclidean distance to it. This process is repeated for all of the training objects and the output is an array of distances.

Our hardware design is able to communicate with the software running on the ARM CPU by using an AXI4 Stream Interface. It possesses a slave port, from which it is able to collect data, and a master port, to which it can send the generated output.

[1] `https://reference.digilentinc.com/reference/programmable-logic/zybo/start`

[2] `https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data`

A high level representation of the designed IP is present in Figure 1. The clock signal is not shown in order to provide a better visual understanding of the rest of the signals, yet it should be mentioned that it is provided as an input in the AXI-Stream Slave port. Additional hardware had to be added, namely a counter responsible for keeping track of how many features corresponding to a single object have been read. The counter value is used to know which position in the BRAM should be accessed to retrieve the corresponding feature of the testing example.

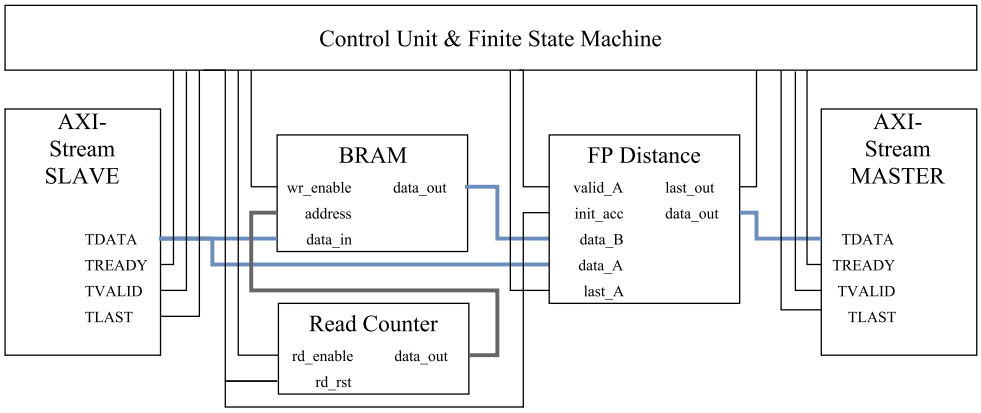

Figure 1: High level datapath representation

2.1 Finite State Machine

Our custom hardware execution is controlled by a 3-State Finite State Machine, that is illustrated in Figure 2.

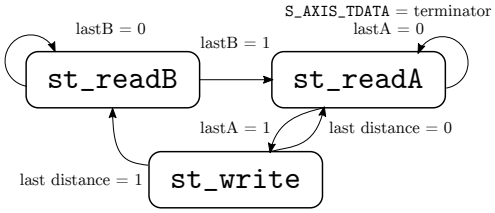

Figure 2: Custom IP core state machine simplified representation

Firstly, in state `st_read_B`, a test object is read into an internal BRAM, each feature at a time. Each feature is represented by a 32 bit single-precision floating point number. When no more input is available, which is indicated by the activation of the input signal `S_AXIS_TLAST`, the process moves on to the next state.

In state `st_read_A`, a training object is read, feature by feature. As soon as a valid float is read, the calculation of the distance starts. A single feature of the testing object and training object are fed to a distance calculator block, described in more detail in Section 2.2. This is repeated until all of the features have been read. Then, the state is altered to `st_write`.

In this final state, the result is written to the output `M_AXIS_TDATA` channel, and the master `TVALID` signal is activated.

There is an issue with this approach that has not yet been mentioned. It is impossible to know the number of training examples beforehand without explicitly communicating it to the hardware. There are several solutions, among which one of the simplest is to make use of a predefined terminator. It consists of a 32 bit binary sequence that can have any value as long as it does not correspond to a valid feature. For this reason, ideally, an invalid float specifier should be used, such as `NaN` (Not A Number). We opted by just using an arbitrary large number (1E30), since Vivado® library for using real numbers does not have native support for NaN, and it would have hindered our debugging procedures.

If in state `read_A` instead of a valid feature, a terminator is read, then an internal register is set to 1, to signal that the next training example will be the last. The synchronous update of this signal is required, in order to avoid latches. When the process moves on to state `st_write`, it can activate the signal `M_AXIS_TLAST` in the same clock cycle the final output is produced and valid. This step effectively flushes the output, by signalling the end-of-packet on the output stream of data. Then, the process is restarted again from state `st_read_B`.

2.2 FP Squared Euclidean Distance Block

A custom block for calculating the single-precision floating point squared euclidean distance operation was designed. It consists of a simple pipeline of three blocks generated using the Floating Point Operator IP provided by Xilinx. The hardware operates on the a single feature at the time, using the value of the testing sample as input B, and the value of the training sample as input A. Firstly, the subtraction between the inputs is calculated in a FP subtractor. Then, the output is fed to both inputs of a FP multiplier, effectively calculating the squared difference between A and B. Finally, the output is connected to an accumulator, which is reset each time a new training sample is loaded. The floating point IPs are generated to operate with a non-blocking behaviour, which simplifies data communication by not having the `TREADY` signal. A diagram of the distance calculator IP is shown in Figure 3. The pipelining of the results between stages is achieved by connecting the valid output signal `TVALID` of a given stage to the valid input port of the following stage.

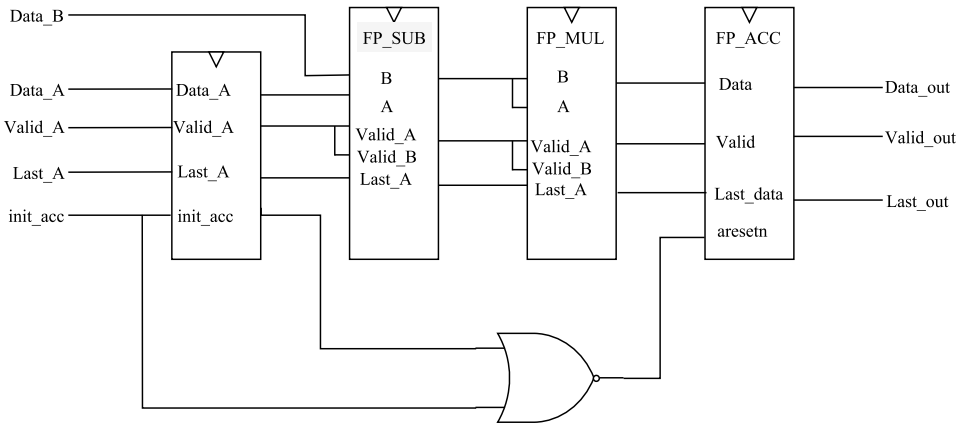

Figure 3: FP Squared Euclidean Distance Block Diagram

2.3 Simulation

The projected IP was simulated in Vivado®, by creating a test bench that generates a clock and the input signals. A detail of the output of the simulation can be seen in Figure 4.

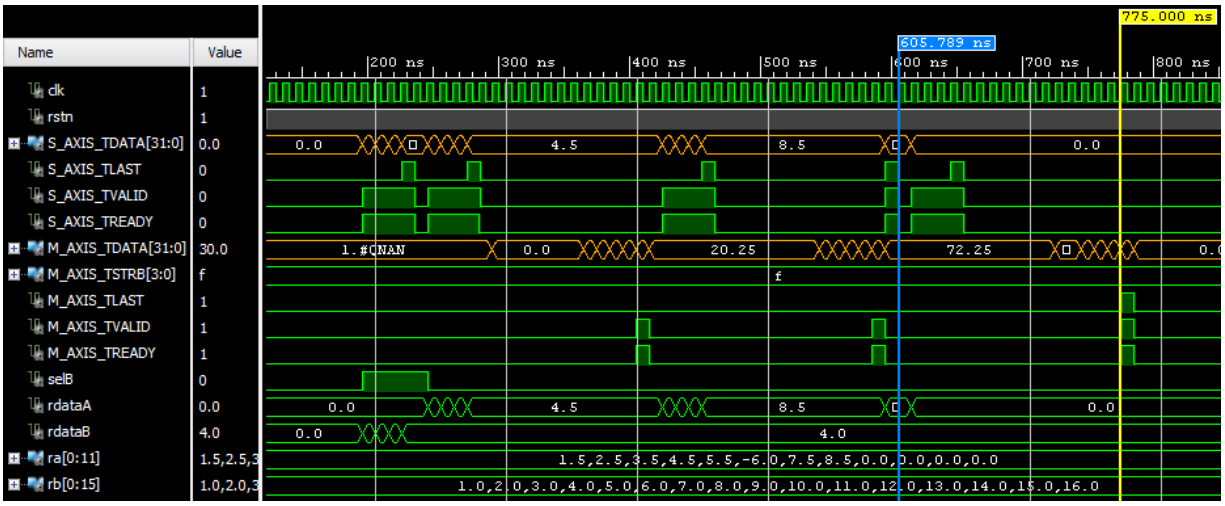

Figure 4: IP Simulation

Firstly, a testing sample and a training sample consisting of 4 features are sent over via `S_AXIS_TDATA`. The end of a computation of the distance to each of the training objects can be seen when `M_AXIS_TVALID` is active. This test only has 3 training samples. Notice the blue marker, after the computation of the distance to the first 2 training samples, and before sending the third and final one, a terminator is sent, thereby notifying the IP that the next result it produces should activate both `M_AXIS_TVALID` and `M_AXIS_TLAST`. The yellow cursor shows the correct output distance of 30 between a testing sample with feature vector $[1.0, 2.0, 3.0, 4.0]^{\mathrm{T}}$ and the last training sample with $[2.5, 2.5, 3.5, 4.5]^{\mathrm{T}}$

2.4 Overall Design

After packaging our custom made IP, a new project was created in order to connect the ZYNQ processing system to our IP. This is achieved by using an AXI4-Stream FIFO. It allows to read or write data packets to or from our IP, handling the AXI4-Stream interface signals. An extra AXI4-Interconnect block is necessary to connect the FIFO to the procesing system. The top view of our IP can be seen in Figure 5.

Timing Report

A brief summary of the timing report can be seen in Table 1

These results show that the system can operate at the desired frequency of 100 MHz.

Utilisation Report

A brief summary of the utilisation report is present in Table 2.

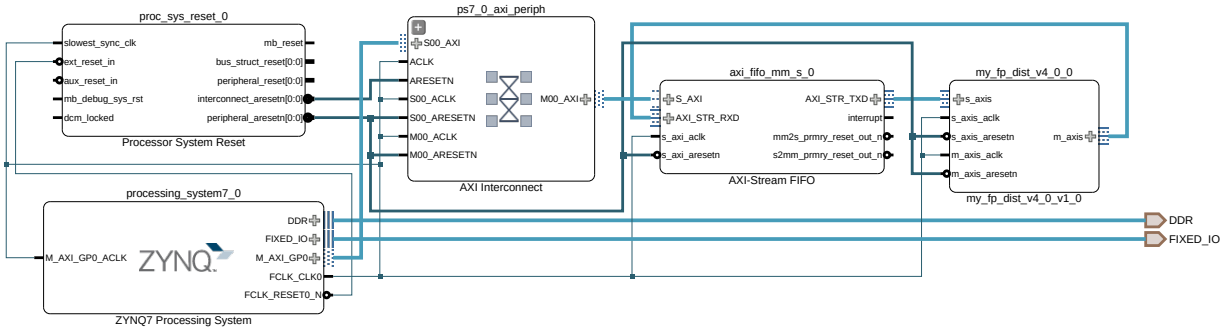

Figure 5: Top view of the system

Worst Negative Slack	Worst Hold Slack
0.775 ns	0.010 ns

Table 1: Timing report summary

Site Type	Used	Utilisation(%)
Slice LUTs	1990	11.31
as Logic	1798	10.32
as Memory	192	3.20
Slice Registers	1821	5.17
as Flip Flop	1821	5.17
as Latch	0	0
Block RAM Tile	4	6.67
DSPs	6	7.50

Table 2: Utilisation report summary

This hardware implementation is barely scratching the surface of what is possible to implement using this board, as one can see by the low utilisation percentages. It can also be seen that fortunately no latches were generated as a consequence of poor hardware design.

3 Software

The software performs two major tasks:

- Send data set objects and receive the distance vector from the hardware;
- Sort the distances and determine the output label.

After the training/test split, the dataset must undergo a preprocessing step, that converts the label from a string to an integer from 0 to the number of classes - 1 and splits the data in four

binary files, two for the training and testing samples feature vectors, encoded as single-precision floats, and another two for the class labels, encoded as integers. Upon launching the program on the Zybo board, these binary files are copied to the DDR memory, accessible by known addresses, that must not conflict with either the program instructions or its heap and stack memory. All of the dataset properties (e.g. number of training examples, dimensionality of feature vectors, etc.) should be configured in the header file `data.h`. This, added to having a variable k number of nearest neighbours, allows for a somewhat generic implementation of the k-NN classifier.

The structure of the code that handles the communication between software and hardware is present in Algorithm 1.

Algorithm 1: k-NN Classifier - Kernel

```
1  Initialise Fifo;
2  foreach testing object tst do
3  |  sendToFifo(tst);
4  |  foreach training object trn do
5  |  |  if trn is last then
6  |  |  |  sendToFifo(terminator);
7  |  |  end
8  |  |  sendToFifo(trn);
9  |  end
10 |  reveiveFromFifo(distances);
11 end
```

Each outer loop iteration generates a line of the distance matrix, corresponding to the distances of a test sample to all training examples. After collecting all of the distances, each line is sorted using a modified selection sort, that only obtains the k smallest entries. From this point on, we need only verify which classes the closes training samples belong to, and assign a label to each testing object by majority rule of its closest neighbours.

The functions that interact with the `XL1Fifo` object for reading from and writing to the FIFO were written and provided by the course professor. These in turn make use of other libraries provided by Xilinx.

The program is uploaded to the Zybo's OCM, in order for instructions to be loaded faster. This could be achieved since the final code was rather small, and made use of lighter libraries, e.g. using calls to `xil_printf` as opposed to `C`'s regular standard I/O operation `printf`. The heap and stack are both placed in DDR memory. There were some issues with insufficient sizes of the heap and stack that caused our program to halt execution and stop producing output. This was fixed by extending the mentioned sizes.

After assigning a label to each testing object, our program prints out the classification results for each of the testing objects, and some timing values for performance measurement.

4 Results

Our program has the same output as the software version, as expected. It manages to correctly classify 48 testing samples out of 50, thus having an accuracy of 96%, for $k = 3$.

The performance of our program is compared to a baseline pure software version, that shares the same structure, yet calculates the squared euclidean distances directly on the CPU. Two time measurements were defined: kernel run time and total run time. The former comprises only the calculation of the distances matrix, without counting the time of setting up variables nor the sorting and class label decision process. The total run time refers to the time it takes from the very start of the program until the final output is written to DDR memory.

It is expected that the version using hardware will in fact be slower than the pure software one, since the calculation of the euclidean distance is a simple procedure, and given that the communication with the hardware introduces a significant overhead.

Table 3 shows the execution run time results obtained for both versions as well as the respective speedups.

Software		**SW+HW**	
Kernel	Total	Kernel	Total
1.71	2.33	7.46	8.08

Kernel	Total
0.23	0.29

Table 3: Kernel and total run time executions (ms), averaged 10 runs (left); kernel and total speedup (right)

It can be seen that the hardware version is in fact quite slower and thus speedup is sub-unitary. As expected, the performance increase gained by having dedicated hardware to calculate the distances does not make up for the overhead introduced by the communication. In the next stage of the project we will attempt to fix this by resorting to direct memory access, which allows the IP to fetch specific input streams and write directly to DDR memory. Several other improvements can be implemented, namely the determination of the k smallest distances on the hardware itself. This simultaneously performs the selection of the k nearest neighbours, and reduces the number of communications greatly.

Furthermore, one could introduce parallelisation, since each testing example can be classified individually. Even though performance was underwhelming, this phase of the project provided important insight into software/hardware co-design and interaction, exposing the major bottleneck in accelerating a given application: communication.