FPGA ACCELERATION FOR RANDOM FOREST INFERENCE

An Undergraduate Research Scholars Thesis

by

DUO WANG

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Jiang Hu

May 2022

Major: Computer Engineering

Copyright © 2022. Duo Wang.

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Duo Wang, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisors prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

Page

ABSTRACT 1

ACKNOWLEDGEMENTS 3

NOMENCLATURE 4

CHAPTERS

1. INTRODUCTION 5
   - 1.1 Problem Description 5
   - 1.2 Prior Work 7
   - 1.3 Expected Results 8
   - 1.4 Social Impacts 9
2. METHODS 11
   - 2.1 General Methodology and Setup 11
   - 2.2 Dataset Preparation 13
   - 2.3 Modified RF Algorithm 16
   - 2.4 Top Function 20
   - 2.5 C Synthesis Optimizations 24
   - 2.6 Vitis HLS and Vivado Simulation 26
3. RESULTS 29
   - 3.1 Vitis HLS Optimization Result 29
   - 3.2 HLS vs CPU Performance Result 31
4. CONCLUSION 33

REFERENCES 34

ABSTRACT

FPGA Acceleration for Random Forest Inference

Duo Wang
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Jiang Hu
Department of Computer Science and Engineering
Texas A&M University

Random forest algorithm has been used broadly in both the research field and in the industry due to its ability to tackle both categorical and numerical dataset. FPGAs also have the highest growing potential and can be applied for the acceleration of random forest inference due to its low power consumption and parallelism support. Research have shown that a compact random forest algorithm is best executed through multi-threading and pipelining, and a FPGA implementation shows significant advantages compared to GP-GPU and CPU implementations in the area. It was able to process each decision tree within the forest independently in parallel. My research is dedicated to achieving this result by benchmarking individual performance running the same RF prediction algorithm on different platforms. The HDL code running on the FPGA will be translated from the source C++ code through Vitis HLS to be synthesized onto the FPGA board. The training data and the binary files will be processed beforehand for an equal competition for all platforms. I will be using various optimization techniques including loop unrolling and data-level parallelism to fully utilize the capabilities of FPGAs. With sufficient

data and analysis, my result will show that FPGAs perform better compared to other platforms such as CPU or GP-GPU.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Jiang Hu, and my graduate mentor, Chan-Wei Hu, for their guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Finally, thanks to my parents for their encouragement and to my mother for her patience and unconditional love.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

I did not receive any fundings throughout my research.

NOMENCLATURE

FPGA	Field Programmable Gate Arrays
GP-GPU	General Purpose Graphical Processing Unit
CPU	Central Processing Unit
HDL	Hardware Description Language
RF	Random Forest
ML	Machine Learning
LUT	Look Up Table
STL	Standard Template Library
IP	Intellectual Property

1. INTRODUCTION

1.1 Problem Description

Recent years with the raise of machine learning algorithms, computer hardware architects have been experimenting different platforms for the best performing architecture when deploying a ML algorithm. In a recent paper Microsoft published in 2018 [1], because of the increasing complexity and features added on network stacks for the Microsoft Azure networking center, they were able to utilize FPGAs as network switchers in their AccelNet (Azure Accelerated Networking) project to provide both programmability and scalable performance compared to traditional ASIC design and CPU clusters. The FPGA is perfect at adapting the current gate array configuration into a more fitting design depending on the request volume and request types. In a research paper published in 2012 [2], studies have shown that although RF algorithm is best executed through threading on OS supported platforms, the FPGA implementation with pipelining shows the best results compare to GP-GPU and CPU implementation. As for right now, although the performance per cost with FGPA application is merely decent, its performance per power is beyond promising. My thesis was suggested by Dr. Jiang Hu, who recognizes the endless possibilities of combining machine learning and FPGAs. I focus on recreating the experiment in [2] with only the forementioned RF prediction algorithm with my adoption to the existing RF implementation in C++ found in [3] to check if the FPGA has the best performance in terms of speed and energy.

Getting into the details of such disruptions, one must observe the general approaching method with the help of HLS software for the implementation of RF on FPGA, and why is HLS bundled with the software development process. The approach typically consists of three steps.

The first step is to implement the training and prediction of the random forest algorithm in software, presumably in C or C++ since those are the only languages HLS software support. Although the algorithm was created in 1995, it is not until recent years that there is sufficient hardware power to support it. One would usually first study the algorithm and discover how it's able to fully utilize all the features FPGA supports. The second step involves converting the C or C++ code into Verilog through the HLS software and deploying the module onto an FPGA board. It is until now that the developers realize that they have very limited choices. If they were to implement the whole project in Verilog, which is the native language any FPGA would support, it will make the scope too complicated and unnecessary due to the existence of HLS software. This tradeoff from the earlier years on performance loss versus scope complexity often occurs and the former is always chosen. This loss of performance has not been a growing issue that hinders the development of FPGA-based software until recent years when the effect of lack of competition started to make a difference. Although the HLS was cleverly written, in a problem like implementing a random forest and the challenges that come with it including algorithm parallelization, and how to make use of the limited BRAMs and flip flops to accommodate the dynamically allocated address memory produced during the training as the tree depth and feature size grows have not yet to be solved or optimized. Moving on in this example, the third step usually involved the introduction of the concept of approximate computing. In a research paper published in 2020 [5], the study shows due to the intensive floating-point calculation the algorithm requires in the training process such as the back-propagating calculation, there is a significant tradeoff between hardware area overhead and classification accuracy. In exchange for a relatively small amount of accuracy, they were able to greatly reduce hardware resource allocation and training time, resulting in a lower power budget and shorter

overall time consumption. This may sound promising on paper, but with the lack of competition, the HLS software has become outdated in such a way that could result in a deceased of performance compared to the GPU and CPU approach of solving this problem due to the lack of recent optimization on newly updated hardware architectures, which is the opposite as the paper states in theory.

1.2 Prior Work

Random forest algorithm has been researched for years and is being used in critical fields. Compared to decision trees, RF has a lower risk of overfitting because it doesn’t strictly fit into the sample data, and it tends to average the results to reduce the chance of false predictions. Areas where RF could be useful include finance, as they were used to estimate customer risk, to detect fraud and to solve option pricing problems. While in the healthcare field, RF is being used in gene expression classification and sequence annotation, and to help doctors respond to specific medications. The study in [2] includes training the forest using compact random forest algorithm on the FPGA first, which results in a compact forest determined by maximum depth parameters.

Random forest algorithm has been researched for years and is being used in critical fields. Compared to decision trees, RF has a lower risk of overfitting because it doesn’t strictly fit into the sample data, and it tends to average the results to reduce the chance of false predictions. As implemented on other devices including GPUs and CPUs, the FPGA approach has been the least researched area. Due to the limited options, rarely has anyone been able to come up with a method to accurately measure the true performance of RFs on FPGAs compared to other platforms, which is due to the reason that HLS has not been the most optimized method of approach and was not optimized because of the lack of competition. As for the customers,

areas where RF on FPGA could have been useful include finance, as they were used to estimate customer risk, to detect fraud, and to solve option pricing problems. While in the healthcare field, RF is being used in gene expression classification and sequence annotation, and to help doctors respond to specific medications. And since prior works have been done in RF include training the forest using compact random forest algorithm on the FPGA, which results in a compact forest determined by maximum depth parameters. For the team managers on such projects, it would encourage the employees to study more and discover more possibilities with other software rather than sticking with HLS, and there would have been multiple FPGA accelerators implemented if only there would be more competitions in the market so that the developers can collaborate better at making the tool more viable in areas where a machine learning algorithm can take advantage of the unique structure of an FPGA. The study from [7] creates a decision tree classifier on the FPGA and stores all parameters in the memory while comparing different parallel architectures. The authors from [6] explores FPGA implementation to accelerate the training process of a decision tree using efficient parallel structures for key steps to determine values for parameters inside of the decision tree. But with the prior work mentioned aforehand, not all have proven that FPGA has the fastest and most efficient results. All these studies are conducted or partially conducted with the help of HLS and due to the nature of the development platform, there is yet much to explore as well.

1.3 Expected Results

The minimum goal of the research is replicate the results in [2] with my own adoption of a RF algorithm, and then to come up with a valid power and time comparison across CPU, GP-GPU and FPGA. The ambitious goal is to implement the approximate decision trees onto the FPGA board. I plan to base the research on studies from the past and investigate the challenges

FPGA faces when working with machine learning algorithms. Historically FPGAs are challenged for its small sized memory issues, which could lead to malfunction of algorithms and speed loss. The design component includes the bare minimum of a RF algorithm, Vitis HLS Linux testing platform and a FPGA to test on. The final project will have a fully functional RF module deployed on the FPGA for RF predictions. The RF algorithm will be tailored to FPGA's parallelizing nature and the lack of the ability to dynamically allocate resources as the tree grows deeper. It will have a better performance per watt compared to a CPU or GPU implementation and will be ready to use with either categorical or quantitative data. If time permits, the final project will also include the approximate computing feature by reducing a small percentage of accuracy for a better performance in terms of power and speed when doing floating point calculations and possibly the complexity of trees and forest.

1.4 Social Impacts

As mentioned above, this solution, compared to a traditional approach has multiple advantages including power efficiency, portability, and flexibility. Although this is not the cheapest solution, it still offers a better performance at the same price point. The impact with this implementation can be applied in banking, government regulations, and the medical field. When applied in the banking field, it has the potential to make personal finance easier to operate and contribute to the economy by creating job opportunities in the hardware industry. When applied in the medical field, or in the field of public health, personal safety, and the general welfare of common people, it has the potential to replace current solutions to be the fastest and the most portable real-time analyzer in the medical fields, and thus it will save more patients and come up with better medicine compared to today's rate. Furthermore, due to the increase in power

efficiency, we can conserve more nature resources and help reduce emission to protect the environment to make a global, cultural, and social impact.

2. METHODS

2.1 General Methodology and Setup

The raw C++ RF prediction algorithm for my research will be adapted from an existing repository found in [3], and the prediction portion of the code will be adapted for HLS and later used to collect timing and energy efficiency data. The raw register transfer level description will be translated from the same C++ code through Vitis HLS software, which I later optimized and tailored to the research. The data used for training and prediction is found on Kaggle [4] where the author compiled a set of binary classification data for heart disease prediction published by the University of California, Irvine.

The trained RF binary file will be generated with the training portion of the code from [3], and later used for prediction by transferring the same binary file to different platforms (CPU, GP-GPU and FPGA) for power and timing comparison. The picture as shown in figure 2.1 depicts the process of the trained forest binary file being generated with the passing arguments including thread counts, number of trees, tree type for being classification, path to the input file and the dependent variable name which is called "HeartDisease" in this example.

```
:: ./ranger --verbose --file heart.csv --depvarname HeartDisease --treetype 1 --ntree 10000 --nthreads 10  --write
Starting Ranger.
Loading input file: heart.csv.
Growing trees ..
Computing prediction error ..
Saved forest to file ranger_out.forest.

Tree type:                         Classification
Dependent variable name:           HeartDisease
Number of trees:                   10000
Sample size:                       691
Number of independent variables:   11
Mtry:                              3
Target node size:                  1
Variable importance mode:          0
Memory mode:                       0
Seed:                              0
Number of threads:                 10

Overall OOB prediction error:      0.150507

Saved confusion matrix to file ranger_out.confusion.
Finished Ranger.
```

Figure 2.1: Training completion and forest file generation example.

For the training platforms, The CPU will utilize the TAMU computing service with dual Intel(R) Xeon(R) CPU E5-2650s and 96 Gb onboard memory, the FPGA was selected from a range of Xilinx FPGA and the Zybo Z7 7010 was chosen for its compatibility and portability. As shown in figure 2.2, it has 512 MB DDR3 memory and 4400 logic slices of 4 6-input LUTs and 8 flip-flops.

Figure 2.2: The Zybo Z7 7010 FPGA.

For the setup of the research, the server is connected to the 110V AC wall-in power. The server is able to produce efficient power to the board and to prepare the necessary files in binary and synthesis the C++ code to FPGA logic in order to program the board. The FPGA is connected to the server via a USB-A cable for power and in order to transfer necessary files in binary and the synthesis Verilog code in one or several bitstreams. The FPGA usually operates on a range of 1.2 to 3.3 V from the server but the actual power in watt will highly depend on the workload and different stages of the application. The server is responsible for converting C++ code into synthesized Verilog to the FPGA and the database. It contains the HLS program for C++ to Verilog conversion. The FPGA is a physical component responsible for training the ML module and doing predictions with the model. Once the server grabbed the C++ code and all the training and prediction data from the server, it starts converting the code into Verilog with Vitis HLS. After the conversion, it generates a Vivado IP which can be brought into Vivado for more behavior simulation and implementation that feeds the Verilog to the FPGA board where the machine learning algorithm is implemented. It also transfers the training data and prediction data over for predicting in the testbench file written in Verilog. Vitis HLS and Vivado will be installed on the server to support the C synthesis of the HLS code and Verilog synthesis for the FPGA. The FPGA will receive the Verilog code and specific instructions for place and route so that each flip flop can be used as either an array for input data or the intermediate values during the prediction phase. The FPGA must be always connected to the server.

2.2 Dataset Preparation

Since binary classification is a major application of random forest algorithm. I used a dataset focusing on predicting if a person would have heart disease based on features such as age and sex in [4]. It introduces one independent variable, HeartDisease, to indicate if that person has

heart disease or not for a particular set of traits. It also has 9 dependent variables which is more than enough for the scope of this research since the goal is to compare performance instead of accuracy. The data contains 35.92 kilobytes of data and is perfectly suitable to be split into 80 percent training and 20 percent prediction datasets with my algorithm. Since the algorithm in [3] only works with numerical data, I first cleaned up the raw data by converting string text in the data into integers count from 0 to the last possible different strings for each attribute in the data if there is any. And then I first created dictionaries for each of the string attributes so I can use them to store the strings into corresponding numerical values, and then I used regex patterns to match the commas in text and replace them with spaces since comma will be the delimitator for further string slicing as seen in figure 2.3.

```
import json
import re

# Get dicts
time_dict = {}
port_dict = {}
carr_dict = {}
time_count = 0
port_count = 0
carr_count = 0
with open("train_test.csv","r") as t:
    next(t)
    for row in t:
        if row.count(",") == 26:
            match = re.search(r"\".*?\"",row).group()
            fixed = match.replace(",","")
            row = row.replace(match,fixed)
        l = row.split(",")
        for i in range(len(l)):
            if i == 4:
                if l[i] not in time_dict:
                    time_dict[l[i]] = time_count
                    time_count+=1
            elif i == 8:
                if l[i] not in carr_dict:
                    carr_dict[l[i]] = carr_count
                    carr_count+=1
            elif i == 17:
                if l[i] not in port_dict:
                    port_dict[l[i]] = port_count
                    port_count+=1
            elif i == 20:
                if l[i] not in port_dict:
                    port_dict[l[i]] = port_count
                    port_count+=1
```

Figure 2.3: Input csv parsing and dictionary construction.

After reading in the training dataset csv file and constructed the aforementioned dictionaries, I then replaced the commas and the string text in different attributes with their corresponding numeric value as seen in figure 2.4.

```
with open("time_ref.txt","w") as t:
    t.write(json.dumps(time_dict))
with open("port_ref.txt","w") as t:
    t.write(json.dumps(port_dict))
with open("carr_ref.txt","w") as t:
    t.write(json.dumps(carr_dict))
print("HT Done.")

# Update file
with open("train_test.csv","r") as t:
    next(t)
    with open("training.csv","a") as u:
        for row in t:
            if row.count(",") == 26:
                match = re.search(r"\".*?\"",row).group()
                fixed = match.replace(",","")
                row = row.replace(match,fixed)
            l = row.split(',')
            res = ""
            strl = [str(int) for int in l]
            res = ",".join(strl)
            u.write(res)
```

Figure 2.4: String replacement and cleaned data generation.

For the splitting of training and prediction datasets, I wrote the algorithm that could first read in the whole csv, and then randomly select a row of data to be appended into a list of rows of size 20% of the total rows. While the data is being appended into the new list of rows, it also removes the randomly selected rows from the original file and since it became the training file with 80% of the total data and then the list with 20% of total data becomes the data for prediction. It then generates the two files with the prediction and training data as separate files as shown in figure 2.5.

```python
import random

with open("demo_total.csv","r") as t:
    header = next(t)
    content = t.readlines()
    final = []
    count = len(content)
    for i in range(int(5*0.2)):
        if i%1000 == 0:
            print(i)
        final.append(content.pop(random.randrange(count)))
        count-=1
with open("demo_pred.csv","w") as a:
    a.write(header)
    for i in final:
        a.write(i)
with open("demo_train.csv","w") as a:
    a.write(header)
    for i in content:
        a.write(i)
```

Figure 2.5: String replacement and cleaned data generation.

2.3 Modified RF Algorithm

Since the RF algorithm has been explored for the last century, there is little reason to come up with a brand new RF algorithm. From the work of [3], the researchers are able to come up with an algorithm that divides the whole algorithm into smaller sets of C++ classes. The main function first takes in user input arguments, including dependent variable, number of trees, number of threads, forest or data input file location, prediction mode and forest type. It also has the binary classification support which is perfect for the dataset in [4]. Based on the input, it creates class instance including a “Data”, “Forest”, “Tree”, “ForestClassification” and “TreeClassification” that takes in the aforementioned arguments for class construction to either store data or used for later calculation. If the training feature is selected, like shown in figure 2.6, the output is able to capture the process of taking in the required arguments and generating the binary file “ranger_out.forest” for the prediction algorithm with the generated forest.

```
Starting Ranger.
Loading input file: ../../../database/pred_data_demo.csv.
Loading forest from file ranger_out.forest.
Predicting ..

Tree type:                          Classification
Dependent variable name:            DEP_DEL15
Number of trees:                    10
Sample size:                        9
Number of independent variables:    25
Mtry:                               5
Target node size:                   1
Variable importance mode:           0
Memory mode:                        0
Seed:                               0
Number of threads:                  10

Saved predictions to file ranger_out.prediction.
Finished Ranger.
```

Figure 2.6: The training stage output with another example dataset and 10 threads, 10 trees.

For prediction, it begins to read in the binary forest file generated from the training stage and retrieve all the required variable including number of columns, number of rows, ordered variables, number of trees and dependent variable. It then performs a cross-check to make sure if the input dependent variable matches the variable name from the binary forest input file. Since the dataset is used for binary classification to determine if a plane was delayed according to other factors of the event in figure 2.7.

```
class Data {
public:
    Data();

    void set_y(size_t col, size_t row, double value, bool &error);

    void set_x(size_t col, size_t row, double value, bool &error);

    double get_x(size_t row, size_t col);

    size_t getSnp(size_t row, size_t col, size_t col_permuted);

    bool isOrderedVariable(size_t varID);

    std::vector<std::string> variable_names;
    size_t num_rows;
    size_t num_rows_rounded;
    size_t num_cols;

    unsigned char *snp_data;
    size_t num_cols_no_snp;

    // For each varID true if ordered
    std::vector<bool> is_ordered_variable;

    // Order of 0/1/2 for ordered splitting
    std::vector<std::vector<size_t>> snp_order;
    bool order_snps;

    std::vector<double> x;
    std::vector<double> y;

};
```

Figure 2.7: The Data class which stores the majority of feature for each tree.

It then constructs a "ForestClassification" object and pushes "Tree" objects into a vector of trees which requires reading in more details from the binary file, which includes child_nodeIDs, split_varIDs, split_values, class_values and response_classIDs as seen in figure 2.8.

```
class Tree {
public:
    Tree(std::vector<std::vector<size_t>> child_nodeIDs, std::vector<size_t>
     split_varIDs, std::vector<double> split_values, std::vector<double>
     class_values, std::vector<unsigned int> response_classIDs);

    double getPrediction(size_t sampleID);

    // Terminal nodeIDs for prediction samples
    std::vector<size_t> prediction_terminal_nodeIDs;

    void tree_predict(const Data prediction_data, bool oob_prediction);

private:
    // Vector of left and right child node IDs, 0 for no child
    std::vector<std::vector<size_t>> child_nodeIDs;

    // Splitting variable for each node
    std::vector<size_t> split_varIDs;

    // Value to split at for each node, for now only binary split
    // For terminal nodes the prediction value is saved here
    std::vector<double> split_values;

    // Classes of the dependent variable and classIDs for responses
    std::vector<double> class_values;
    std::vector<unsigned int> response_classIDs;

};
```

Figure 2.8: The Data class which stores the majority of feature for each tree.

They will be used later to retrieve the prediction value by going down the individual tree and finally returns with the value on the terminal node. On CPU or GPU, the program will first have to split the process onto different threads and use these threads for speeding up the traversal by splitting the trees across all the threads. It would then aggregate the results by selecting the most voted answer from all the trees for the final prediction of the program.

```
std::vector<std::thread> threads;
threads.reserve(num_threads);
//data.get()->to_string();
for (uint i = 0; i < num_threads; ++i) {
  threads.emplace_back(&Forest::predictTreesInThread, this, i, data.get(),
   false);
}
showProgress("Predicting..", num_trees);
for (auto &thread : thread &  : threads) {
  thread.join();
}
```

Figure 2.9: The original C++ code uses the thread STL for multithreading.

Since the FPGA does not have a CPU on board, it will not be able to process any multithreading and mutex commands. On the FPGA, instead of multithreading, I converted and optimized the prediction process into FPGA parallel process, so that it would be able to utilize as many gate arrays as possible. Essentially, transferring the creation of threads with a for loop with passes the thread index into the required function. Instead of indicating the true thread ID, the index passed into the functions in the for loop become literal numerical values which get over the unsupported thread STL for HLS as seen in figure 2.9. So that, in the test bench file, I included everything up to the tree traversal and loaded everything from the OS in the testbench file since FPGA does not have an OS. The top function I included and later synthesized on the FPGA only contains the tree traversal.

2.4 Top Function

A major part of the HLS process is the synthesis of the top function of the project, which is the starting point for any simulations in Verilog. The top function I have chosen to be adopted to fit the HLS requirements is the tree traversal function which is originally a part of the Tree class from [2] as shown in figure 2.10. It is the lowest level of vector manipulation of the whole algorithm, and it is the most time consuming function and thus a perfect top function for HLS. This function is called for each thread to iterate through the attributes of prediction_data, which

is a compact Data class containing different forest and tree results stored as C++ STL vector types, and tree node values stored in vector child_nodeIDs, split_varIDs, split_values to traverse until the end of a node on a tree and return the values in form of another vector prediction_terminal_nodeIDs.

```
std::vector<size_t> predict(Data prediction_data, std
        ::vector<size_t> prediction_terminal_nodeIDs,
        std::vector<std::vector<size_t>> child_nodeIDs
        , std::vector<size_t> split_varIDs, std::
        vector<double> split_values);
```

Figure 2.10: The original C++ class function header from [2].

Since the C++ STL classes are not synthesizable, I had to convert the vectors into HLS_vector types, which are natively supported by all Xilinx models and are fully synthesizable. Another problem with a pure substitution is the port restrictions from Vitis HLS. It specifies all port length cannot exceed 4096 bits. Since the parameters of the top function will be translated into input ports where the return vector will be translated into the output port, while the original size of the vectors is relatively huge compared to the 4096 bits restriction, I have to split them into smaller sections of vectors. As seen in figure 2.11, the input types have been converted to HLS_vector and split into 4096 bits length vectors. The x vector, num_rows and is_ordered_variable are all original attributes of prediction_data and now they are able to be passed and synthesized without any problems. The length specified for the vectors are hard-coded for simulation purposes. To minimize the length of the ports, I have to use short int and float instead of their counterparts int and double. Short int and float both have 2 bytes and 4 bytes, which is not as accurate compared to 4 and 8, since the data I used doesn’t contain values that require this much of accuracy and range, this is perfectly acceptable for the purpose of the research.

```
hls::vector<short int, 227> test(hls::vector<bool,11> is_ordered_variable,
                                 hls::vector<hls::vector<short int, 128>, 2> child_nodeIDs_1,
                                 hls::vector<hls::vector<short int, 128>, 2> child_nodeIDs_2,
                                 hls::vector<hls::vector<short int, 5>, 2> child_nodeIDs_3,
                                 hls::vector<short int, 256> split_varIDs_1,
                                 hls::vector<short int, 5> split_varIDs_2,
                                 hls::vector<float, 128> split_values_1,
                                 hls::vector<float, 128> split_values_2,
                                 hls::vector<float, 5> split_values_3,
                                 hls::vector<float,128> x_hls_1,
                                 hls::vector<float,128> x_hls_2,
                                 hls::vector<float,128> x_hls_3,
                                 hls::vector<float,128> x_hls_4,
                                 hls::vector<float,128> x_hls_5,
                                 hls::vector<float,128> x_hls_6,
                                 hls::vector<float,128> x_hls_7,
                                 hls::vector<float,128> x_hls_8,
                                 hls::vector<float,128> x_hls_9,
                                 hls::vector<float,128> x_hls_10,
                                 hls::vector<float,128> x_hls_11,
                                 hls::vector<float,128> x_hls_12,
                                 hls::vector<float,128> x_hls_13,
                                 hls::vector<float,128> x_hls_14,
                                 hls::vector<float,128> x_hls_15,
                                 hls::vector<float,128> x_hls_16,
                                 hls::vector<float,128> x_hls_17,
                                 hls::vector<float,128> x_hls_18,
                                 hls::vector<float,128> x_hls_19,
                                 hls::vector<float,65> x_hls_20,
                                 int index
                                 ) {
```

Figure 2.11: The adopted top function header code.

The algorithm consists of two main parts. The first part is responsible for taking in the broken down vectors as shown in the parameters labeled by an underscore plus the index number and combine them into one with for loops. For example, as shown in figure 2.12, for loops are used to repopulate the arrays “x” from the 20 arrays it received into one and are now partitioned on the DRAMs onto the FPGA. Once all the required data has been imported and repopulated, the second step involves the tree traversal as seen in figure 2.13. It performs inferencing on the sample index of the data set and then traverses the tree like structured child_nodeIDs vector that contains features of a decision tree to determine the values used for the final prediction and then write it back to return_terminal. I hard-coded the number 18 as the for loop range since it is the maximum depth in my RF example.

```
    float x[2497];

    for (int u = 0; u < 128; ++u) {
#pragma HLS PIPELINE
        x[u+128*0] = x_hls_1[u];
        x[u+128*1] = x_hls_2[u];
        x[u+128*2] = x_hls_3[u];
        x[u+128*3] = x_hls_4[u];
        x[u+128*4] = x_hls_5[u];
        x[u+128*5] = x_hls_6[u];
        x[u+128*6] = x_hls_7[u];
        x[u+128*7] = x_hls_8[u];
        x[u+128*8] = x_hls_9[u];
        x[u+128*9] = x_hls_10[u];
        x[u+128*10] = x_hls_11[u];
        x[u+128*11] = x_hls_12[u];
        x[u+128*12] = x_hls_13[u];
        x[u+128*13] = x_hls_14[u];
        x[u+128*14] = x_hls_15[u];
        x[u+128*15] = x_hls_16[u];
        x[u+128*16] = x_hls_17[u];
        x[u+128*17] = x_hls_18[u];
        x[u+128*18] = x_hls_19[u];
    }
```

Figure 2.12: Local float array x being regenerated from 19 indices passed into the array.

```
    size_t nodeID = 0;
    // For each sample start in root, drop down the tree and return final value

        for (int j = 0; j< 18;++j) {
#pragma HLS PIPELINE off
            // Break if terminal node
                if (child_nodeIDs[0][nodeID] == 0 && child_nodeIDs[1][nodeID] == 0) {
                    break;
                }

                // Move to child
                size_t split_varID = split_varIDs[nodeID];

                double value = x[split_varID * 227 + index];

                if (is_ordered_variable[split_varID]) {
                    if (value <= split_values[nodeID]) {
                        // Move to left child
                        nodeID = child_nodeIDs[0][nodeID];
                    } else {
                        // Move to right child
                        nodeID = child_nodeIDs[1][nodeID];
                    }
                } else {
                    size_t factorID = floor(value) - 1;
                    size_t splitID = floor(split_values[nodeID]);

                    // Left if 0 found at position factorID
                    if (!(splitID & (1ULL << factorID))) {
                        // Move to left child
                        nodeID = child_nodeIDs[0][nodeID];
                    } else {
                        // Move to right child
                        nodeID = child_nodeIDs[1][nodeID];
                    }
                }
            }

        return_terminal[index] = nodeID;

    return return_terminal;
```

Figure 2.13: Decision tree traversal with a maximum depth of 18 levels.

2.5 C Synthesis Optimizations

To achieve optimal performance, in the official Vitis HLS manual [8], the authors have provided a wide variety of pragma form optimizations. The pragma “HLS PIPELINE” found in figure 2.12 under the declaration of the for loop tells HLS to unroll the loop and utilize the rich number of flip-flops in the FPGA to rearrange the loop in a pipelined fashion. With this, HLS performs a static parallel analysis to maximize throughout by manipulate instructions to fit in as many instructions as possible in one clock cycle. Since the behavior is closely resembling a typical CPU behavior, some dependences, such as true data dependences, must be avoided and will result in latency increase if not handled correctly. The original code from [3] included a greedy loop with “while (1)” that exhaust all the leaf nodes in a decision tree to reach to the final leaf node that contains the prediction as shown in figure 2.14.

```
while (1) {

  // Break if terminal node
  if (child_nodeIDs[0][nodeID] == 0 && child_nodeIDs[1][nodeID] == 0) {
    break;
  }

  // Move to child
  size_t split_varID = split_varIDs[nodeID];

  double value = prediction_data->get_x(sample_idx, split_varID);
  if (prediction_data->isOrderedVariable(split_varID)) {
    if (value <= split_values[nodeID]) {
      // Move to left child
      nodeID = child_nodeIDs[0][nodeID];
    } else {
      // Move to right child
      nodeID = child_nodeIDs[1][nodeID];
    }
  } else {
    size_t factorID = floor(value) - 1;
    size_t splitID = floor(split_values[nodeID]);
```

Figure 2.14: The “while (1)” greedy logic snippet from [3].

The same logic cannot be included for C synthesis. It is simply because the generation of LUTs with the complier needs to know exactly how many times a loop has to operate when determining how many clocks cycles this operation takes, so that a constant range must be provided for this tree traversal. This would require first determining the maximum number of iterations of this while loop for any decision trees during this process, the maximum depth of any decision trees for the dataset I used, and then converting the greedy “while (1)” condition to a range based for loop with the depth hard-coded as the range for the dataset I used. I have determined the maximum depth for my dataset is 18 and the modified code snippet is already captured in figure 2.13, as the range for the for loop replacing the while loop has the number 18 encoded. This will tell the C Synthesis complier exactly how many clocks cycle this operation takes when generating the timing analysis.

Another requirement for the pragma “HLS PIPELINE” to work without any overhead for a nested for loop is the concept of perfect vs imperfect loop. In [8], the Vitis HLS tool in default applies loop flatten to any nested loops with the “HLS PIPELINE” specified in the outer loop. Since there would be extra clock cycles introduced to get in and out of the nested inner loop. For example, as part of the repopulation step in my top function, a 2D array is repopulated with a nested for loop for later traversal as shown in figure 2.15. The loop has to be “perfect” in HLS terms such that first, the inner loop does not have a variable bound, and second, there is no logic between the outer for loop statements and the inner for loop statement. Once the two conditions are met, they will be able to be optimized by HLS.

```
    for (int i = 0; i < 2; i++) {
#pragma HLS PIPELINE
        for (int j = 0; j < 128; j++) {
            child_nodeIDs[i][j] = child_nodeIDs_1[i][j];
        }
    }
```

Figure 2.15: child_nodeIDs is being repopulated from the input parameter child_nodeIDs_1 in a nested loop.

Other automatically applied optimization pragma include array partitions as denoted by "#pragma HLS ARRAY_PARTITION variable= dim= complete/cyclic/block". Since Vitis compiler results the arrays onto the RAM on the FPGA, it could be time consuming when elements have to be accessed due to latency being larger then register communication. This bottleneck is even more apparent when the array is accessed more than one time. The "ARRAY_PARTITION" pragma, as noted in [8], has the ability to tell the compiler to map the data to smaller arrays and make them register access on the FPGA. This would require a large amount of registers on the physical part and the latency is the tradeoff.

Simulation results with different combinations of optimization will be included in the results chapter.

2.6 Vivado Behavior Simulation

Once the C simulation and C synthesis passed with the modified algorithm written in C++, the next step is to generate a Vivado IP and export the RTL to Vivado for further simulations and verification. Vitis HLS also generates a high level wrapper that contains the inputs and outputs of the final design, and it is able to automatically wire the internal logic with all the generated Verilog files. As shown in figure 2.16, the testbench file test_tb.sv I wrote is capable of creating a unit under test with the auto-generated Verilog wrapper bd_0_wrapper.v from Vitis HLS.

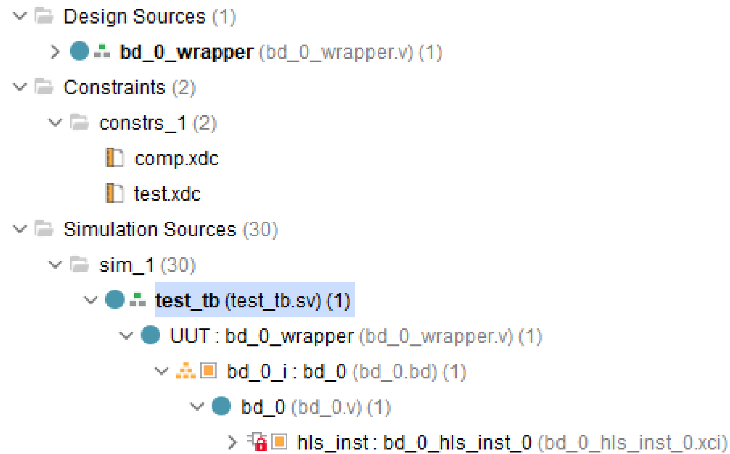


Figure 2.16: Verilog wrapper after C Synthesis and testbench hierarchy.

The testbench file is written in SystemVerilog to enable support for short int and float data type to better work with the inputs of the Verilog wrapper. The testbench file consists of two major parts. The first part is to load all the required data to be fed into the inputs of the wrapper. For example, figure 2.17 shows the process of reading the file containing one of the required parameters "is_ordered_variable" line by line from the server, and then it is able to write to the SystemVerilog variable "is_ordered_variable_flattened" in bits to send to the wrapper as its parameter.

```
is_ordered_variable_scan = $fscanf(is_ordered_variable_data, "%d", int_data);
if (!$feof(is_ordered_variable_data)) begin
    is_ordered_variable_flattened[(is_ordered_variable_index*8) +: 8] = int_data;
    is_ordered_variable_index <= is_ordered_variable_index + 1;
end
```

Figure 2.17: Loading from file in SystemVerilog.

The second part is to construct a Unit under test as shown in figure 2.18. It sends all of the required parameters to the wrapper with a few extra input registers. Register "ap_clk" specifies the clock frequency for the simulation. Register "ap_rst" specifies the reset signal which is always set to false for the purpose of this simulation. Register "ap_ctrl_start" is always

set to 1 to ensure the control signals are always on. Register "ap_return" is used for final output and is has to be set with a register from the testbench file of the same size for verification.

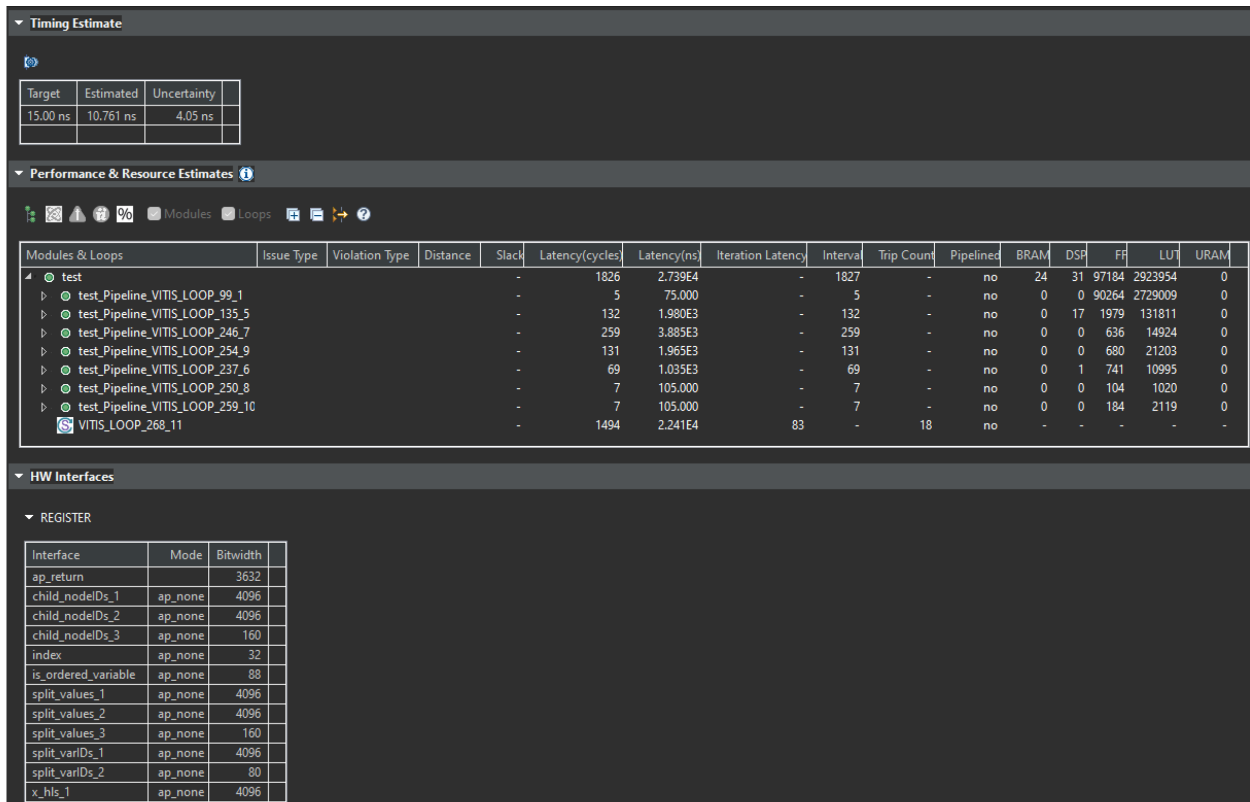
```
// Pass to test
bd_0_wrapper UUT (
    .child_nodeIDs_1(child_nodeIDs_hls_1_flattened),
    .child_nodeIDs_2(child_nodeIDs_hls_2_flattened),
    .child_nodeIDs_3(child_nodeIDs_hls_3_flattened),
    .is_ordered_variable(is_ordered_variable_flattened),
    .split_values_1(split_values_1_flattened),
    .split_values_2(split_values_2_flattened),
    .split_values_3(split_values_3_flattened),
    .split_varIDs_1(split_varIDs_1_flattened),
    .split_varIDs_2(split_varIDs_2_flattened),
    .x_hls_1(x_hls_1_flattened),
    .x_hls_2(x_hls_2_flattened),
    .x_hls_3(x_hls_3_flattened),
    .x_hls_4(x_hls_4_flattened),
    .x_hls_5(x_hls_5_flattened),
    .x_hls_6(x_hls_6_flattened),
    .x_hls_7(x_hls_7_flattened),
    .x_hls_8(x_hls_8_flattened),
    .x_hls_9(x_hls_9_flattened),
    .x_hls_10(x_hls_10_flattened),
    .x_hls_11(x_hls_11_flattened),
    .x_hls_12(x_hls_12_flattened),
    .x_hls_13(x_hls_13_flattened),
    .x_hls_14(x_hls_14_flattened),
    .x_hls_15(x_hls_15_flattened),
    .x_hls_16(x_hls_16_flattened),
    .x_hls_17(x_hls_17_flattened),
    .x_hls_18(x_hls_18_flattened),
    .x_hls_19(x_hls_19_flattened),
    .x_hls_20(x_hls_20_flattened),
    .ap_return(ap_return),
    .index(index),
    .ap_clk(ap_clk),
    .ap_rst(ap_rst),
    .ap_ctrl_start(ap_ctrl_start)
);
```

Figure 2.18: Passing required parameter to wrapper to create the unit under test.

3. RESULTS

3.1 Determining Optimal Clock Period

The process of determining the most optimal clock period involves taking trails and running the C synthesis to see if there is any timing violations with the target clock period. This means that Vitis HLS is able to schedule all the required logic in 1 clock cycle within the provided range of period. The timing report is automatically generated by Vitis HLS. For example, figure 3.1 shows the timing information with a target period of 15 ns, while the final latency in cycles and ns can be found on the first line in the table under “Performance & Resource Estimates” for the top function which is named test.



Target	Estimated	Uncertainty
15.00 ns	10.761 ns	4.05 ns

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
test				-	1826	2.739E4	-	1827	-	no	24	31	97184	2923954	0
test_Pipeline_VITIS_LOOP_99_1				-	5	75.000	-	5	-	no	0	0	90264	2729009	0
test_Pipeline_VITIS_LOOP_135_5				-	132	1.980E3	-	132	-	no	0	17	1979	131811	0
test_Pipeline_VITIS_LOOP_246_7				-	259	3.885E3	-	259	-	no	0	0	636	14924	0
test_Pipeline_VITIS_LOOP_254_9				-	131	1.965E3	-	131	-	no	0	0	680	21203	0
test_Pipeline_VITIS_LOOP_237_6				-	69	1.035E3	-	69	-	no	0	1	741	10995	0
test_Pipeline_VITIS_LOOP_250_8				-	7	105.000	-	7	-	no	0	0	104	1020	0
test_Pipeline_VITIS_LOOP_259_10				-	7	105.000	-	7	-	no	0	0	184	2119	0
VITIS_LOOP_268_11				-	1494	2.241E4	83	-	18	no	-	-	-	-	-

Interface	Mode	Bitwidth
ap_return		3632
child_nodeIDs_1	ap_none	4096
child_nodeIDs_2	ap_none	4096
child_nodeIDs_3	ap_none	160
index	ap_none	32
is_ordered_variable	ap_none	88
split_values_1	ap_none	4096
split_values_2	ap_none	4096
split_values_3	ap_none	160
split_varIDs_1	ap_none	4096
split_varIDs_2	ap_none	80
x_hls_1	ap_none	4096

Figure 3.1: Timing analysis report generated by Vitis HLS (snippet).

Taking trails and running the synthesis, the table 3.1 reflects the process and the most optimal clock period turns out to be 11.3 ns for my current design. Equation 3.1 shows how to calculate total latency in nanoseconds.

$$Total\ latency\ in\ ns\ = Total\ latency\ in\ cycles\ *\ Target\ period\ in\ ns \qquad (Eq\ 3.1)$$

Table 3.1: Target period with the respective total latency in cycles and ns.

Target period (ns)	Total latency (cycles)	Total Latency (ns)
10 (Timing violation)	N/A	N/A
15	1826	27390
13	1844	23970
11.5	1847	21240
11.3	1847	20870

3.2 Vitis HLS Optimization Comparison Results

With the optimization methods mentioned in subsection 2.5, the table 3.2 shows how different optimization methods affect the total latency in ns. The clock period used for to generate the following results is 11.3 ns as mentioned in subsection 3.1. An example of using breakdown for loops in array regeneration can be seen in figure 3.2, where individual for loops are used to regenerate array x. Whereas a single for loop is used to regenerate the entire array can be seen from figure 2.12, where the same logic is written under with a single for loop.

Table 3.2: Vitis HLS optimization effect on final timing.

Breakdown for loop in array regeneration	For loop pipelining in array regeneration	Latency (cycles)	Latency (ns)
Yes	Yes	3129	35360
Yes	No	4681	52898
No	Yes	1847	20870
No	No	2972	33580

```
    for (int u = 0; u < 128; ++u) {
#pragma HLS PIPELINE
        x[u+128*0] = x_hls_1[u];
    }
    for (int u = 0; u < 128; ++u) {
#pragma HLS PIPELINE
        x[u+128*1] = x_hls_2[u];
    }
    for (int u = 0; u < 128; ++u) {
#pragma HLS PIPELINE
        x[u+128*2] = x_hls_3[u];
    }
    for (int u = 0; u < 128; ++u) {
#pragma HLS PIPELINE
        x[u+128*3] = x_hls_4[u];
    }
    for (int u = 0; u < 128; ++u) {
#pragma HLS PIPELINE
        x[u+128*4] = x_hls_5[u];
    }
    for (int u = 0; u < 128; ++u) {
#pragma HLS PIPELINE
        x[u+128*5] = x_hls_6[u];
    }
```

Figure 3.2: A single for loop is broken down into individual ones for the same logic (snippet).

3.3 HLS vs CPU Performance Result

The HLS timing has been computed above, and the CPU time can be calculated and reported using the chrono library. With a simple subtraction, the duration can be calculated in equation 3.2 and the code is found in figure 3.3.

$$Duration\ (ns)\ =\ time_after_function_call\ (ns)\ -\ time_before_function_call\ (ns) \qquad (Eq\ 3.2)$$

```
auto start = std::chrono::high_resolution_clock::now();
prediction_terminal_nodeIDs_hls = test(...); // Pass parameters here

auto stop = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(stop - start);
```

Figure 3.3: C++ logic for calculate CPU time in ns.

After taking the average over 1000 runs with the same code running, an average latency can be computed and the speedup over the HLS latency can be calculated with the equation 3.3. The data collected is shown in table 3.3.

$$Speedup\ =\ Average\ CPU\ latency\ (ns)\ /\ Best\ Vitis\ HLS\ total\ latency\ (ns) \qquad (Eq\ 3.3)$$

Table 3.3: Vitis HLS timing results vs CPU timing result.

Average CPU latency (ns)	Best Vitis HLS total latency (ns)	Speedup
65196.7	20870	3.1239

4. CONCLUSION

From the subsection 3.3, since the speedup is relatively small compared to the desired speedup, there is not enough evidence to say the FPGA implementation has a significant advantage over the CPU implementation. The result can also be affected by hardware restrictions that the CPU is not optimized for small, single core operations, and the FPGA device, Zybo Z7 7010, does not have enough flip-flops and LUTs for further on-board testing. Although the advantage of this implementation over CPU is not prominent, the process of combining different optimization techniques greatly shows the potential for HLS focused optimization on FPGAs. For example, the reason why breaking down the loops has worse performance in table 3.2 is because there is no dependence exists between the statements in the loop, breaking the loops into separate individual loops will only add in the overhead and thus increasing the overall latency. If the dependency exists between the statements, the hardware scheduler may take a longer time to process the pipeline and the overall performance may be worse due to the pipeline overhead. Since the scheduler has to compensate for the dependency when the pipeline pragma is under effect, it must delay the pipeline by putting in stalls and increase the total latency. The experiment from table 3.1 is also interesting. It shows the physical limits of FPGA devices and the main reason FPGAs can't be used to replace the existing CPU based machines. But at the same time, it also opens up other possible optimization techniques focusing on minimizing the clock frequency on the critical path, and it also shows the impact of LUT size and flip-flop numbers on FPGAs, such that a larger FPGA board usually would take less optimization efforts to achieve the same timing requirements.

REFERENCES

[1] D. Firestone, “Azure accelerated networking: Smartnics in the public cloud,” Microsoft, 2018. [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2018/03/Azure_SmartNIC_NSDI_2018.pdf.

[2] B. Van Essen, “Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?,” IEEE Xplore, 2012. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6239820.

[3] M. N. Wright, “ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R,” Journal of Statistical Software, Mar-2017. [Online]. Available: https://arxiv.org/pdf/1508.04409.pdf.

[4] Fedesoriano, “Heart failure prediction dataset,” Kaggle, 10-Sep-2021. [Online]. Available: https://www.kaggle.com/datasets/fedesoriano/heart-failure-prediction/metadata.

[5] M. Barbareschi, “Advancing synthesis of decision tree-based multiple classifier systems: an approximate computing case study,” Springer, 12-Apr-2021. [Online]. Available: https://link.springer.com/content/pdf/10.1007/s10115-021-01565-5.pdf.

[7] X. Lin, “Random Forest Architectures on FPGA for multiple applications,” Association for Computing Machinery, May-2017. [Online]. Available: https://dl.acm.org/doi/epdf/10.1145/3060403.3060416.

[8] “Vitis high-level synthesis user guide - xilinx.eetrend.com,” Xilinx, 22-Oct-2021. [Online]. Available: http://xilinx.eetrend.com/files/2021-11/wen_zhang_/100555486-227863-ug1399-vitis-hls.pdf.