Pulsar Search with Random Forest on FPGAs

Arunkumar M V

September 6, 2022

Introduction

This presentation outlines the implementation of random forest classifier (RFC) in hardware intended for FPGAs.

High-level project description

The project produces hardware designs of RFCs in the following manner:

- Training of the RFC is done using a python application - *(Software stage)*.
- Tree structures are then extracted and used to create verilog designs using chisel3-based parameterized circuit generators - *(Hardware stage)*.
- Generated design may then be tested using an in-house verilator-based simulator or used for synthesis using external FPGA toolchains - *(Run stage)*

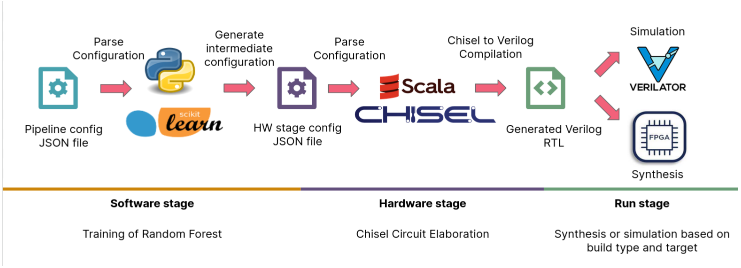
Apporach

The following slides present our approach. The project is divided into the following modules/stages:

- *Software stage*
- *Hardware stage*
- *Run stage*

Build pipeline

The aforementioned stages are linked together to form the build pipeline. The stages and overall pipeline have been implemented with Scala.


Parse Configuration
Generate intermediate configuration
Parse Configuration
Chisel to Verilog Compilation
Simulation
VERILATOR
Pipeline config JSON file
scikit learn
HW stage config JSON file
Scala
CHISEL
Generated Verilog RTL
FPGA
Synthesis
Software stage
Training of Random Forest
Hardware stage
Chisel Circuit Elaboration
Run stage
Synthesis or simulation based on build type and target

Configuring the build pipeline

- A JSON-based configuration system is used to configure the build pipeline
- Configuration file contains various parameters to be used by the different pipeline stages
- Enables the pipeline to be versatile and user-friendly

Example input configuration

```
{
    "dataset": "datasets/HTRU_2.csv",
    "input_headers": ["Mean␣of␣the␣integrated␣profile", "
        Standard␣deviation␣of␣the␣integrated␣profile", "
        Excess␣kurtosis␣of␣the␣integrated␣profile", "
        Skewness␣of␣the␣integrated␣profile", "Mean␣of␣the␣DM
        -SNR␣curve", "Standard␣deviation␣of␣the␣DM-SNR␣curve
        ", "Excess␣kurtosis␣of␣the␣DM-SNR␣curve", "Skewness␣
        of␣the␣DM-SNR␣curve"],
    "target_header": "Result",
    "train_split_size": 0.7,
    "n_estimators": 100,
    "max_leaf_nodes": 3,
    "build_type": "test",
    "build_target": "sim",
    "fixed_point_width": 32,
    "fixed_point_bp": 16,
    "opt_majority_voter": "area"
}
```

Build types and targets

The build type and target determine the final products that the build pipeline generates. These are given as input configuration parameters in the JSON file.

Build types

The build type decides which type of design is generated during the build.

Possible build types

- `test`
  - Generates a “test harness” module that wraps around a random forest classifier module
  - This module contains the test data and expected results and automatically tests the random forest classifier module
  - This build is meant for development and verification purposes
- `production`
  - Generates a random forest classifier module with appropriate communication support
  - This build is meant for deploying to production

NOTE: Currently only test builds are supported. Support for production builds will be added in the future

Build targets

The build target decides what is done with the generated verilog design during the build.

Possible build targets

- `simulation`
  - May be used to simulate the generated verilog design
  - Builds a verilator-based simulator model and runs the simulator to verify the behaviour of the module
- `synthesis`
  - May be used when the generated modules are to be synthesised by an external FPGA tool chain
  - Basically causes the build pipeline to exit after verilog generation from chisel

Software stage

The software stage performs the training of the RFC. It performs the following steps:

1. Checks for / creates a python virtual environment
2. Installs the required python libraries
3. Reads the input pipeline configuration JSON file
4. Loads and splits the dataset into training and testing data
5. Performs the fitting/training of the random forest classifier
6. Outputs the decision trees data structures and other configurations to the hardware stage as a JSON file

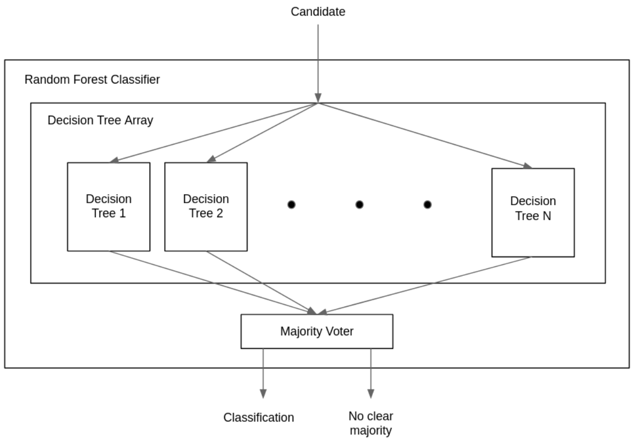
Hardware stage

The hardware stage generates the verilog design based on input configuration. It performs the following steps:

1. Reads the configuration JSON file generated by the software stage
2. Supplies the required chisel generator module with necessary parameters
3. Performs the circuit elaboration/compilation
4. Outputs the run stage configuration as an intermediate Scala object

Hardware architecture

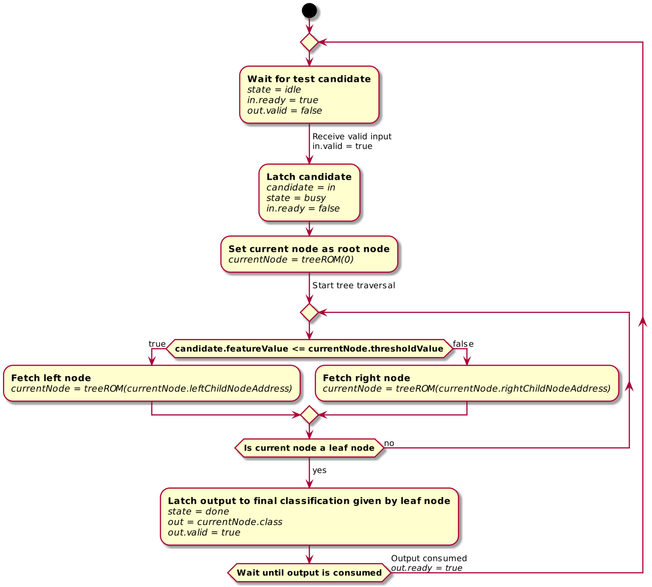
The figure below depicts the overall architecture of the RFC implemented in Chisel. The following slides delve into different modules present in this architecture.



Decision tree module

This module stores a decision tree in an internal ROM and performs tree traversal when a candidate arrives for testing.

Flowchart of the decision tree module



Structure of a node in the decision tree ROM

Is Leaf Node?	Feature/Class Index	Threshold	Right Child Node Address	Left Child Node Address
1 bit	max(feature/ class index width) bits	fixed point width bits	$log_2(N)$ bits	$log_2(N)$ bits

Decision tree array module

This module consists of multiple decision tree modules that form the random forest. It handles the passing of test candidates to the various trees.

Majority voter module

This modules takes as inputs form the individual trees and makes the final decision regarding the classification of the data point.

- The class getting the maximum votes (from each of the decision trees) is declared as the final class of the test candidate.
- In case a majority is not achieved (two or more classes getting same number of votes) a separate ”no clear majority” flag is asserted.

Random Forest Classifier module

This modules ties the decision tree array module to the majority voter module. The input to this module are the test candidates and the output are their corresponding classes.

Run stage

The run stage takes the generated verilog and performs the necessary actions based on the build target. It either builds and runs the verilator-based simulator or outputs the generated verilog file for further syntheis with external FPGA toolchains.

Observations

The project was tested with the HTRU 2 dataset and some key observations were made. These are presented in the following slides

Discrepancy between software and hardware majority voting

It was noted that there are a relatively small number of discrepancies between the hardware and software predictions. This is because software prediction algorithms uses a confidence based system for finding the final classification while hardware uses a voting based system. *This has even lead to cases where hardware gives correct prediction relative to the target while the software is wrong.*

Pros

- Prediction method in hardware is much simpler as it uses integer comparisons as opposed to floating point based operations in software.

Cons

- Hardware has higher chance of encountering "No Clear Majority" cases than software.

Decision trees and other configurations being baked into hardware design

It can be observed that the generated design is not generic and the RFC model is baked into it.

Pros:

- The hardware architecture is leaner as it does not need to deal with memory arbitration or external communication for acquiring the decision trees.

Cons:

- If any changes are to be made to the model it would require the design to be recompiled from chisel, resynthesised and loaded on to the FPGA. This is unfavorable for cases where online learning may be required.

Salient features

Some salient features of the implementation:

- Input JSON configuration make the project more versatile and user-friendly
- Implementation of circuits using highly parameterized chisel generators make the design highly scalable (any number of decision trees of any size)
- Supports trees of varying depth
- Simpler implementation as the ROMs are baked into the design
- Automatic code generative architecture

Project documentation

- Paper can serve as primary documentation.
- Developer oriented documentation on build system and configuration system have been added to the codebase.
- The code has also been documented with scaladoc and comments.

Thank you!