ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Optimized FPGA implementation for Random Forests for Anomaly Detection

Εμμανουήλ Αραπίδης

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα Οκτώβριος 2023

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Optimized FPGA implementation for Random Forests for Anomaly Detection

Εμμανουήλ Αραπίδης

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1η Νοεμβρίου 2023.

Τριμελής Επιτροπή Εξέτασης

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξύδης
Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής
ΕΜΠ	ΕΜΠ	ΕΜΠ

Αθήνα Οκτώβριος 2023

(Υπογραφή)

.....

Εμμανουήλ Αραπίδης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Οι Βιομηχανικές Εγκαταστάσεις πάντα είναι στόχος επιθέσεων σε διάφορες μορφές όπουν έχουν επιπτώσεις σε κοινωνικό, οικονομικό και ακόμα και πολιτικό επίπεδο. Η ενσωμάτωση προηγμένων πληροφοριακών τεχνολογιών, όπως το Διαδίκτυο των Πραγμάτων, τις έχει εκτεθέσει σε ζητήματα ασφαλείας του ψηφιακού κόσμου και έχει φέρει στο προσκήνιο κυβερνοεπιθέσεις. Μια δημοφιλής στρατηγική για την προστασία των βιομηχανικών εγκαταστάσεων από τέτοιου είδους επιθέσεις είναι η ανάπτυξη αλγορίθμων και μεθόδων ανίχνευσης βασισμένων σε μηχανική μάθηση. Η πρόκληση σε αυτές τις περιπτώσεις είναι να αναπτυχθούν γρήγορες, ακόμα και πραγματικού χρόνου λύσεις, που ανιχνεύουν και ακόμα αποτρέπουν κακόβουλες επιθέσεις.Σε αυτήν την διπλωματική εργασία, προτείνουμε την επιτάχυνση σε FPGA για ένα προγνωστικό μοντέλο που ανιχνεύει κυβερνοφυσικές επιθέσεις σε μια ασφαλή μονάδα επεξεργασίας νερού (SwaT). Η βιβλιογραφία δείχνει ότι τα μοντέλα μηχανικής μάθησης Random Forest εμφανίζουν υψηλή ακρίβεια στην ανίχνευση τέτοιων επιθέσεων. Ο στόχος της διατριβής είναι η επιτάχυνση του Random Forest (RF) μέσω της χρήσης του πλαισίου High Level Synthesis (HLS) Vitis, προσανατολιζόμενο σε μια συσκευή FPGA. Προτείνουμε μια ιεραρχική στρατηγική βελτιστοποίησης που στοχεύει στην ενίσχυση της απόδοσης μέσω του συνδισμού τεχνικών βελτιστοποίησης του κώδικα πηγής και τεχνικών παραλληλισμού που είναι ενσωματωμένες στο HLS. Σε ένα πρώτο επίπεδο, ενεργοποιούμε τον παραλληλισμό εντός της εκτέλεσης ενός Random Forest. Σε ένα δεύτερο επίπεδο, εξερευνούμε τον παραλληλισμό σε πολλαπλούς Random Forests με δύο διαφορετικές αρχιτεκτονικές: (i) ένα coarse grained design που διευκολύνει τον παράλληλο χειρισμό μέσω πολλαπλών instances ενός μόνο σχεδιασμού RF και (ii) έναν σχεδιασμό throughput-optimized pipeline που βασίζεται στην παράλληλη εκτέλεση πολλαπλών Random Forests μέσω μιας ακολουθίας ομοιογενών μονάδων επεξεργασίας. Τέλος,και στα δύο επίπεδα εφαρμόζεται η τεχνική precision scaling που επιτυγχάνει επιπλέον ενίσχυση της απόδοσης μέσω της εκτέλεσης λιγότερο πολύπλοκων λειτουργιών και αποτελεσματικής χρήσης πόρων. Οι παραγόμενοι σχεδιασμοί αξιολογούνται έναντι της εκτέλεσης C++ και Python για διάφορα μεγέθη εισόδου, επιτυγχάνοντας μέγιστη επιτάχυνση $\times 20.03$.

Λέξεις κλειδιά: Ανίχνευση ανωμαλιών, Βιομηχανικές Εγκαταστάσεις, Επυτάχυνση σε FPGA, Vitis HLS Tool, Scikit Learn

Abstract

Industrial facilities have always been the target of attacks in various forms and impact on a social, economic and even political level. The integration of advanced information technologies such as the Internet of Things in such environments has exposed them to security issues of the digital world and have brought forward cyber attacks. A popular strategy to protect industrial facilities from such attacks, is to develop detection algorithms and methods based on machine learning as well as time series analysis. The challenge in this cases is to develop quick, even real-time solutions, that detect and even deter malicious attacks. In this work, we propose FPGA acceleration for a predictive model that detects cyber-physical attacks at a Secure Water Treatment facility (SwaT). Literature shows that Random Forest machine learning models exhibit high accuracy in the detection of such attacks. The goal of the thesis is to accelerate the Random-Forest(RF)-based inference through the use of High Level Synthesis framework Vitis HLS targeting an FPGA device. We propose a hierarchical optimization strategy that targets performance enhancement through the synergy of source code optimization techniques and HLS-inherent parallelization techniques. On a first level, we enable parallelism within the execution of a single Random Forest inference task. On a second level, we explore parallelism across multiple Random Forests by proposing two different architectures: (i) a coarse-grained design that facilitates parallel execution through multiple instances of a single RF design and (ii) a throughput-optimized design that is based on pipelined execution of multiple Random Forest across an array of homogeneous Processing Units. Lastly, these two levels of parallelism are coupled with a precision scaling exploration that achieves further performance enhancement through the execution of less complex operations and efficient resources utilization. The above strategy is further enclosed in an automated framework that performs an exploration over a set of examined parameters and delivers a pareto front with solutions that achieve a trade-off between performance and resources utilization. The generated designs are evaluated against C++ and python inference for various inputs sizes achieving a speedup of almost $\times 20.03$.

Key words: Anomaly Detection, Industial Facilities, Acceleration on FPGA, Vitis HLS Tool, Scikit Learn

Ευχαριστίες

Θα ήθελα, εν πρώτοις, να ευχαριστήσω τον επιβλέποντα καθηγητή κ. Δημήτρη Σούντρη για την εμπιστοσύνη που μου έδειξε για την εκπόνηση της παρούσας διπλωματικής αλλά και την μοναδική εκπαιδευτική εμπειρία που αυτή μου προσέφερε. Επιπλέον θέλω να ευχαριστήσω την υποψήφια διδάκτρορα κα. Κωνσταντίνα Κολιογεώργη, τον Επίκουρο Καθηγητή Γιώργο Λεντάρη και τον υποψήφιο διδάκτρορα Δημήτρη Δανόπουλο για τις συμβουλές τους και την καθοδήγησή τους σε όλα τα στάδια εκπόνησης της παρούσας εργασίας. Κλείνοντας, θα ήθελα να ευχαριστήσω τους φίλους μου χάρη στους οποίους τα φοιτητικά χρόνια θα μου μείνουν αξέχαστα αλλά και τους γονείς μου και τα αδέρφια μου, Φανούρη και Μαριαλένα, για την αμέριστη στήριξη που μου έδειξαν κατά τη διάρκεια των σπουδών μου.

Contents

List of Figures

List of Tables

Listings

1 Εκτενής Περίληψη

1.1 Εισαγωγή-Κίνητρο

Τα τελευταία χρόνια παρατηρείται ότι όλο και περισσότερες Βιομηχανικές Εγκαταστάσεις γίνονται στόχο επίθεσεων. Πολλά από αυτά τα συστήματα δεν είχαν σχεδιαστεί με γνώμονα την ασφάλειά. Μέχρι πρότινος οι τεχνολογίες λειτουργίας ήταν διαχωρισμένες από τις τεχνολογίες πληροφορίας. Επιπλέον απαιτούνταν εξειδικευμένες γνώσεις για τον χειρισμό και την ορθή λειτουργία αυτών. Λόγω αυτών οι Βιομηχανικές εγκαταστάσεις ήταν πιο ασφαλείς σε σχέση με σήμερα. Η συνεχως αυξανομενη ανάπτυξη και ζήτηση της τεχνολογίας Internet of Things(IoT) έχει ενισχύσει το παραπάνω φαινόμενο. Παρά τα ξεκάθαρα οφέλη, σχετικά με την παραγωγικότητα, την αποδοτικότητα και την αυτοματοποίηση, που προσφέρει η ενσωμάτωση του IoT και η διασύνδεση των συστημάτων ελλοχεύει ο κίνδυνος να εκτεθούν περισσότερες αδυναμίες.

Προκειμένου να διασφαλύσουμε τις βιομηχανικές εγκαταστάσεις από τέτοιου είδους επιθέσεων χρειάζεται ο ταχύς εντοπισμός αυτών και η ανάπτυξη αποτρεπτικών μηχανισμών. Σκοπός αυτής της διπλωματικής εργασίας είναι ο σχεδιασμός ενός ισχυρού προβλεπτικού μοντέλου που θα μπορεί να εντοπίσει τέτοιου είδους επιθέσεις και να το επιταχυνουμε κατα το inference. Για να πετύχουμε αυτό κάνουμε Anomaly Detection με χρήση αλγοριθμων μηχανικής μάθησης, όπως ο Random Forest ensembler.Το παραγόμενο αυτό μοντέλο επιλέξαμε να το επιταχύνουμε σε FPGA δειςε με την βοήθεια του εργαλείου Vitis HLS.

Η ανίχνευση ανωμαλιών, γνωστή και ως ανίχνευση ακραίων τιμών, αναφέρεται στο πρόβλημα της εύρεσης μοτίβων σε δεδομένα που δεν συμμορφώνονται με την αναμενόμενη συμπεριφορά. Κυρίως επικεντρωνόμαστε .Χρησιμοποιώντας ιστορικά δεδομένα εκπαιδευουμε το μοντέλο Random Forest για να κάνει τέτοιες προβλεψεις. Random Forest είναι ένας αλγόριθμος supervised learning όπου αποτελείται από μια συλλογή από Decision Trees. Κάθε Decision Tree εκπαιδευεται ανεξαρτητα απο τα υπολοιπα σε ενα υποσύνολο από το αρχικό dataset. Για να προκύψει κάποια πρόβλεψη κάθε δέντρο κάνει ανεξαρτητα την δική του και ύστερα αυτές συλλέγονται. Στην δική μας περίπτωση όπου μας ενδιαφέρει το classification, η τελική προβλεψη εξάγεται με majority voting.

Μια από τις κύριες προκλήσεις που σχετίζονται με την εφαρμογή real-time anomaly detection είναι το γρήγορο και αξιόπιστο infer-

ence. Στην εργασία μας προτείνουμε να χρησιμοποιήσουμε Field Programmable Gate Arrays (FPGA) και το toolchain Vitis-HLS. Οι δυνατότητες παραλληλίας που προσφερουν τα FPGA μας επιτρέπει να μειώσουμε αισθητά το χρόνο inference, κάτι απαραίτητο για real-time και security critical εφαρμογές σαν την δική μας. Επιπλέον, μέσω το Vitis μπορούμε να γράψουμε τον κώδικα μας σε high level γλώσσα όπως η C++ και όχι σε γλώσσα περιγραφής υλικού όπως Verilog ή VHDL. Μας δίνει την ευελιξία και ευκολία να μπορούμε να επικεντρωθούμε στην βελτίωση του design του αλγοριθμου και όχι στην υλοποίηση του σε low-level.

Στα πλαίσια αυτής της διπλωματικής εργασίας, εξετάσαμε τη δυνατότητα παραλληλοποίησης του αλγορίθμου RF για την επίτευξη επιτάχυνσης. Στο εσωτερικό ενός RF, κάθε Decision Tree είναι ανεξάρτητο από τα υπόλοιπα, και αυτό αποτελεί το πρώτο επίπεδο παραλληλίας που εκμεταλλευτήκαμε. Δημιουργήσαμε διάφορα αντίγραφα σε hardware για κάθεDT, επιτρέποντας την παράλληλη εκτέλεσή τους.

Εξερευνήσαμε επίσης υψηλότερα επίπεδα παραλληλίας, όπως αυτό της παράλληλης επεξεργασίας πολλαπλών δειγμάτων εισόδου. Προτείναμε δύο διαφορετικές αρχιτεκτονικές. Η πρώτη, Coarse Grained Multiple RF, περιλαμβάνει πολλαπλές αντίγραφες του ίδιου RF, με παράλληλους decision tree estimators σε κάθε ένα εξ΄ αυτών. Αυτό επιτρέπει την παράλληλη εκτέλεση πολλαπλών ινπυτς ταυτόχρονα. Υπάρχει σαφής συμβιβασμός μεταξύ απόδοσης και χρήσης πόρων, τον οποίο αναλύσαμε λεπτομερώς.

Η δεύτερη προτεινόμενη αρχιτεκτονική ονομάζεται Pipelined RF. Διαπιστώσαμε ότι η διαδικασία διάσχυσης ενός DT μπορεί να κατακερματιστεί σε ανεξάρτητα στάδια. Χρησιμοποιώντας buffers, μετατρέψαμε τη διαδικασία σε pipeline. Συνδυάσαμε παράλληλα και ανεξάρτητα pipelined DTs για να δημιουργήσουμε σε υψηλό επίπεδο ένα pipelined RF. Το pipelined RF μπορεί να δέχεται νέα δείγματα εισόδου σε κάθε κύκλο, επιτρέποντας την παραλληλή διαχείριση τους και την επίτευξη χαμηλού latency. Και στις δύο αρχιτεκτονικές, εφαρμόσαμε την τεχνική precision scaling για βελτίωση της απόδοσης κατά το inference και μείωση της χρήσης πόρων.

Στο υπόλοιπο της διπλωματικής εργασίας, παρουσιάσαμε την μεθοδολογία μας, τα αποτελέσματα της εφαρμογής των διαφορετικών τεχνικών και τις συγκρίσεις με C++ και Python υλοποιήσεις του RF αλγορίθμου. Εκτελέσαμε ανάλυση δεδομένων στα ευρήματα και παρουσιάσαμε συμπεράσματα, κλείνοντας με μια ματιά στο μέλλον και τη δημιουργία

ενός εργαλείου που αυτοματοποιεί τη διαδικασία του design exploration.

1.2 Σχετική Εργασία

Η Ανιχνεύση Ανωμαλιών είναι ένα ευρέως ευρημενό ζήτημα. Πολλές διαφορετικές προσεγγίσεις έχουν δοκιμαστεί στην προσπάθεια ανάπτυξης ενός αξιόπιστου μοντέλου. Μια από αυτές αφορά την σύγκριση Support Vector Classifires(SVM) και Random Forest(RF). Οι συγγραφείς του παπερ χρησιμοποιούν 2 διαφορετικά datasets που αφορουν Intrusion Detection(ID) σε βιομηχανικές εγκαταστάσεις. Σε μια άλλη έρευνα, επιδιωξαν να αναπτύξουν ένα έξυπνο και ασφαλές μοντέλο για να ανιχνεύουν αδυναμίες σε IoT συστήματα και να προστατεύονται από cyber-attacks.Για να το καταφέρουν αυτό, σύγκριναν πολλούς διαφορετικούς αλγόρυθμους μηχανικής μάθησης. Συνολικά χρησιμοποιήθηκαν 5, οι οποίοι ήταν Logistic Regression(LR), Support Vector Machine, Decision Trees(DT), Random Forest και Artificial Neural Networks (ANN).Συμπέραναν ότι ένα απλό μοντέλο όπως το PΦ μπορεί, σε συνάρτηση πάντα με το δατασετ, να φέρει καλύτερα αποτελέσματα όσον αφορά την ακρίβεια των προβλέψεων σε συγκρίση με πολύ πιο περίπλοκα σαν το ANN.

Είναι εμφανές ότι ο RF, χάρις στη καλή ακρίβεια και τους γρήγορους χρόνους εκπαίδευσης και εκτέλεσης, είναι μια δημοφιλής επιλογή στο χώρο του anomaly detection. Πάντα υπάρχει όμως η ανάγκη για πιο ακριβή, ταχύτερές και ενεργεικά συμφέρουσες λύσεις. Λόγω αυτού, έχουν γίνει προσπάθειες για επιτάχυνση του μοντέλου σε πληθώρα hardware πλατφορμών, όπως GPU και FPGA. Μία από αυτές είναι το παπερ όπου χρησιμοποιείσαι μια παραλλαγή του κλασικού αλγόριθμου RF, η οποία είναι Compact Random Forest(CRF). Οι βασικές διαφορές τους είναι στο πλήθος των DT και στο μέγιστο δεπτη που επιτρέπεται να μεγαλώσει κάθε δέντρο. Η ανάγκη για αυτές τις αλλαγές προκύπτει από το γεγονός ότι ο PΦ αλγόριθμος είναι memory bound κάτι το οποίο δεν ενδείκνυται για hw acceleration, με περιορισμένους πόρους. Στην υλοποίηση σε FPGA προτείνεται μια pipeline αρχιτεκτονική με N διαφορετικά piplines για N δέντρα. Στην υλοποίηση CP-GPU, εστιάζουν στον μέγιστο εκμεταλλευτικό παραλληλισμό σε πολυπύρηνες μονάδες και την επίτευξη βέλτιστης επαναχρησιμοποίησης δεδομένων στις μνήμες cache.Η μελέτη καταλήγει στο συμπέρασμα ότι τα FPGA παρέχουν υψηλότερη απόδοση ανά watt και μπορούν να χειριστούν μεγαλύτερους ταξινομητές χωρίς να υπ-

ονομεύουν τις δυνατότητές τους.

Βασισμένοι στα προηγούμενα, αποφασίσαμε να πραγματοποιήσουμε Ανίχνευση Ανωμαλιών χρησιμοποιώντας Random Forest σε ένα FPGA. Καταρχάς, δεν θα επιβάλουμε κανέναν περιορισμό στο μέγεθος του RF κατά την εκπαίδευση. Επιπλέον, θα συγκρίνουμε δύο διαφορετικές αρχιτεκτονικές στο FPGA. Μια αρχιτεκτονική είναι προσέγγιση pipeline, παρόμοια με αυτή που προτάθηκε στο άρθρο, ενώ η άλλη εμπλέκει μια πιο παράλληλη υλοποίηση.

Χάρη στις αυξημένες δυνατότητες των σύγχρονων FPGAs, θα στοχεύσουμε στο να επιτύχουμε τον μέγιστο δυνατό αριθμό παράλληλων και ανεξάρτητων περιπτώσεων RF που μπορούν να χωρέσουν στον πίνακα. Όλες οι περιπτώσεις θα είναι πανομοιότυπα αντίγραφα μεταξύ τους. Εσωτερικά, τα δέντρα θα επεξεργαστούν κάθε δείγμα παράλληλα σε και τις δύο προτεινόμενες αρχιτεκτονικές. Στόχος μας είναι να συγκρίνουμε ποια από τις δύο προσεγγίσεις, σωληνωτή ή παράλληλη, θα μας παρέχει υψηλότερη απόδοση και να εξετάσουμε την ισορροπία μεταξύ πόρων και απόδοσης.

1.3 Random Forest Αλγόριθμος

Ο αλγόριθμος RF προτάθηκε πρώτη φορά από τον Leo Breiman. Συνδίασε πολλές τεχνικές και πρότεινε καινούργιες. Αποτελεί μια παραλλαγή του CART αλγορίθμου στον οποίο εφαρμόστηκε βαγγινγ και τυχαία επιλογή χαρακτηριστικών . Ο CART αλγόριθμος είναι δύσκολο να μην κάνει overfit τα δεδομένα και συμπεριφερεται μη αποδοτικά σε ουτλιερς. Ο Breiman πρότεινε να χρησιμοποιηθει η τεχνική bagging ή αλλιώς bootstrap aggregating. Στο Bagging πολλοί αδύναμοι [λεαρνερς] εκπαιδεύονται παράλληλα ο καθένας σε διαφορετικό σύνολο δεδομένων, το οποίο προκύπτει με τυχαία ανακατομή του αρχικού. Το σύνολο των δεδομένω όπου εκπαιδεύεται ο κάθενας learner(classifier ή regressor) παράγεται τραβόντας N δείγματα με επανατοποθέτηση, όπου N είναι το μέγεθος του αρχικού. Ο στόχος μας είναι εκπαιδεύοντας κάθε δέντρο,στη δική μας περίπτωση, με διαφορετικό σύνολο δεδομένω να γινει το μοντέλο πιο ανθεκτικό σε νέα δείγματα. Τα δέντρα που παράγονται από την παραπάνω διαδικασία στη συνέχεια αποφασίζουν συλλογικά για τη πρόβλεψη σε κάθε νέα είσοδο. Στη περίπτωση του classification η απόφαση παίρνεται με ψηφοφορία ενώ του regression με το μέσο όρο απο όλες τις προβλέψεις των δέντρων.

Στα περισσότερα σύνολα δεδομένων, υπάρχουν κάποια χαρακτηριστικά που φαίνεται να υπερτερουν των άλλων στη λήψη των αποφάσεων

κατα την διχοτομήση των κόμβωβ και κατα επακόλουθο στη κατασκευή των δέντρων. Ακόμα και με bootstrap φαίνεται ότι τα παραγόμενο δέντρα δεν καταφερνουν να διαφοροποιηθούν αρκετα μεταξυ τους για να αντιμετωπιστεί το overfitting. Ο Breiman προσπαθησε να εντάξει περισσότερη τυχαιότητα στο μοντέλο με στόχο να μειώσει το συσχέτιση του ενος δέντρου από το άλλο και να αυξήσει ή να διατηρήσει την προβλεπτική ικανότητα του καθενός. Πρότεινε να μην χρησιμοποιείται ολοκληρο το εύρος των χαρακτηριστών σε κάθε κόμβο αλλά ένα τυχαίο υποσύνολο αυτών. Το χαρακτηριστικό και η τιμη του που θα καταλήξουμε να κάνουμε διχοτομηση του κόμβου επιλέγεται από αυτό το τυχαίο υποσύνολο. Το πλήθος αυτών είναι σταθερό για όλους κόμβους όλων των δεντρων και συνήθως αποτελεί μια συνάρτηση του συνολικού μεγέθους των χαρακτηρισικών. Μία από τις πιό σύνηθες και αυτή που καταλήγουμε είναι η τετραγωνική ρίζα, άλλη επίσης ευρέως χρησιμοποιούμενη ο λογάριθμος του.Η προσθήκη των παραπάνω τεχνικών συμβάλλει στην εισαγωγή ποικιλίας μεταξύ των δέντρων και στη βελτίωση της ικανότητας του μοντέλου να γενικεύει. Αυτό αντιμετωπίζει προβλήματα όπως το overfitting και ενισχύει τη συνολική απόδοση.

Η διαδικασία inference ενός Ρανδομ Φορεστ περιλαμβάνει τα εξής βήματα:

- Είσοδος: Για κάθε δείγμα Ξ για το οποίο θέλουμε να κάνουμε μια πρόβλεψη, το ΡΦ παίρνει αυτόν τον διάνυσμα χαρακτηριστικών ως είσοδο.

- Διάσχιση Δέντρου:Η είσοδος περνά σε κάθε δέντρο απόφασης. Ξεκινώντας από τη ρίζα, το δείγμα διασχίζει τη δομή του δέντρου βασιζόμενο στα κριτήρια διαίρεσης σε κάθε εσωτερικό κόμβο.Σε κάθε εσωτερικό κόμβο, χρησιμοποιείται ένα συγκεκριμένο χαρακτηριστικό και σημείο διαίρεσης για να καθοριστεί η κατεύθυνση που πρέπει να ακολουθήσει το δείγμα.Η τιμή του χαρακτηριστικού στο δείγμα εξετάζεται σε σχέση με το σημείο διαίρεσης, και με βάση το αποτέλεσμα, το δείγμα μετακινείται στον αριστερό ή δεξιό κόμβο-παιδί. Αυτή η διαδικασία συνεχίζεται αναδρομικά μέχρι το δείγμα να φτάσει σε ένα φύλλο κόμβου, που αντιπροσωπεύει μια πρόβλεψη.

- Ψηφοφορία ή Μέσος Όρος:Τα δέντρα κάνουν συλλογικές προβλέψεις για κάθε είσοδο. Σε εργασίες δυαδικής ταξινόμησης, κάθε δέντρο αποφάσισης ¨ψηφίζει' για μια ετικέτα τάξης βασισμένη

στην πλειοψηφική κλάση στο φύλλο κόμβου όπου καταλήγει το δείγμα εισόδου.Η τελική πρόβλεψη γίνεται μέσω πλειοψηφικής ψηφοφορίας μεταξύ όλων των δέντρων απόφασης. Σε εργασίες παλινδρόμησης, οι προβλεπόμενες τιμές από κάθε δέντρο αποφάσεων λαμβάνονται μέσο όρο για να προκύψει η τελική έξοδος παλινδρόμησης.

- Έξοδος:Το RF εξάγει την τελική πρόβλεψη για το δείγμα εισόδου, η οποία μπορεί να είναι μια ετικέτα τάξης (σε κατηγοριοποίηση) ή αριθμητική τιμη (σε παλινδρόμηση)

1.4 FPGA & HLS Εργαλεία

Ένα FPGA είναι ένα ενσωματωμένο κύκλωμα (IC) που μπορεί να προγραμματιστεί για διάφορους αλγόριθμους μετά την κατασκευή. Τα σύγχρονα FPGAs αποτελούνται από έως δύο εκατομμύρια λογικά κελιά που μπορούν να διαμορφωθούν για την υλοποίηση διάφορων λογισμικών αλγορίθμων. Παρόλο που η παραδοσιακή διαδικασία σχεδιασμού σε FPGA είναι περισσότερο παρόμοια με ένα κανονικό IC παρά με έναν επεξεργαστή, ένα FPGA παρέχει σημαντικά οφέλη σε κόστος και χρόνο συγκριτικά με μια προσπάθεια ανάπτυξης IC και προσφέρει τον ίδιο βαθμό απόδοσης στις περισσότερες περιπτώσεις. Αυτό οφείλεται στη δυνατότητα του FPGA, σε σύγκριση με το IC, να ανακατασκευάζεται δυναμικά. Αυτή η διαδικασία, που είναι παρόμοια με το να φορτώνετε ένα πρόγραμμα σε έναν επεξεργαστή, κάνει αλλαγές στο πραγματικό υλικό και τους διαθέσιμους πόρους στο υλικό του FPGA.Η γενική αρχιτεκτονική του FPGA είναι ετερογενής πλατφόρμα υπολογισμού που αποτελείται από τρία είδη μονάδων. Αυτές είναι οι μονάδες εισόδου/εξόδου, οι μονάδες διακοπής/διασύνδεσης καλωδίων και οι διαμορφώσιμες λογικές μονάδες (CLB).

Η Σύνθεση Υψηλού Επιπέδου (High-Level Synthesis - HLS) είναι η διαδικασία μετατροπής μιας προδιαγραφής σε γλώσσα C σε ένα επίπεδο μεταφοράς καταχωρητών (Register Transfer Level - RTL) για να μπορεί να συνθετιστεί σε ένα FPGA. Αποτελεί γέφυρα μεταξύ λογισμικού και υλικού, επιτρέποντας στους σχεδιαστές να εκφράσουν τα σχέδιά τους χρησιμοποιώντας γλώσσες προγραμματισμού υψηλού επιπέδου όπως η C ή C++. Η HLS προσφέρει αρκετά πλεονεκτήματα σε σύγκριση με τις παραδοσιακές γλώσσες περιγραφής.

Ορισμένα από αυτά περιλαμβάνουν τα εξής. Αφαιρώντας τις λεπτομέρειες σε χαμηλό επίπεδο, οι σχεδιαστές μπορούν να επικεντρ-

ωθούν στο να εκφράσουν την επιθυμητή λειτουργικότητα των σχεδίων τους, αφήνοντας τις λεπτομέρειες υλοποίησης στο εργαλείο HLS. Επίσης, μπορεί να επικυρώσει τη λειτουργική ορθότητα του σχεδιασμού πιο γρήγορα σε σύγκριση με τις VHDL ή Verilog. Οι πολλές οδηγίες βελτιστοποίησης που είναι διαθέσιμες επιτρέπουν έλεγχο επάνω στη διαδικασία σύνθεσης και δημιουργούν ειδικές υλοποιήσεις υψηλής απόδοσης. Με τη βοήθεια των οδηγιών βελτιστοποίησης, μπορεί να δημιουργήσει πολλαπλές υλοποιήσεις από τον πηγαίο κώδικα C. Τέλος, μπορούμε να δημιουργήσουμε αναγνώσιμον και φορητόν πηγαίον κώδικα C, να επαναστείλουμε τον πηγαίο κώδικα C σε διάφορες συσκευές και να τον συμπεριλάβουμε σε νέα έργα.

Μόλις ολοκληρωθεί η σύνθεση του κώδικα από το HlS παράγεται μια αναφορά σύνθεσης. Πληροφορίες σχετικά με τα μετρήσιμα κριτήρια απόδοσης παρέχονται σε αυτήν την αναφορά. Η κατανόηση των κριτηρίων που χρησιμοποιούνται για να μετρήσουν την απόδοση σε ένα σχεδιασμό που δημιουργήθηκε με τη χρήση της HLS είναι κρίσιμη για να επιτευχθεί αυτό αποτελεσματικά. Τα τρία βασικά από αυτά είναι η έκταση (Area), η καθυστέρηση (Latency) και το διάστημα εκκίνησης (Initiation Interval - II).

Χρησιμοποιήσαμε την Ενοποιημένη Πλατφόρμα Vitis για την επιτάχυνση της εφαρμογής μας. Η εφαρμογή χωρίζεται κυρίως σε δύο μέρη, τον κεντρικό υπολογιστή (host) και το FPGA. Ο κεντρικός υπολογιστής λειτουργεί ως η κύρια μονάδα επεξεργασίας που αλληλεπιδρά με το FPGA για την εκτέλεση συγκεκριμένων εργασιών. Ο σκοπός του κεντρικού υπολογιστή είναι ο έλεγχος, η συντονισμός και η επικοινωνία με το FPGA για την ανάληψη υπολογιστικών εργασιών ή την επιτάχυνση συγκεκριμένων τμημάτων μιας εφαρμογής. Ο επεξεργαστής του κεντρικού υπολογιστή επικοινωνεί με το FPGA για τη μεταφορά δεδομένων, παραμέτρων διαμόρφωσης και οδηγιών. Διαμορφώνει το FPGA για την εκτέλεση, στέλνει δεδομένα για επεξεργασία από πυρήνες FPGA και λαμβάνει πίσω τα αποτελέσματα από το FPGA. Επίσης, είναι υπεύθυνος για τον έλεγχο της εκτέλεσης εργασιών στο FPGA. Βεβαιώνεται για τον σωστό συγχρονισμό μεταξύ των εργασιών που τρέχουν στον κεντρικό υπολογιστή και στο FPGA για τη διατήρηση της σωστής ακολουθίας λειτουργιών. Ένας άλλος κρίσιμος ρόλος του κεντρικού υπολογιστή είναι η Προεπεξεργασία και Μεταεπεξεργασία Δεδομένων.

1.5 Υλοποίηση του Random Forestγια Σύνθεση Υψηλού Επιπέδου

Για το κομμάτι του τραινινγ χρησιμοποιήθηκε η python βιβλιοθήκη Scikit-learn. Η Scikit-learn μας έδωσε την δυνατότητα να παραμετροποιήσουμε το μοντελο στις ανάγκες της εφαρμογής μας. Εξετάσαμε πολλές παραμέτρους για την εκπαίδευση των μοντέλων μας.Οι δύο κρίσιμες παράμετροι που επιλέξαμε να εξερευνήσουμε είναι ο αριθμός των εκτιμητών και ο αριθμός των χαρακτηριστικών. Επικεντρωθήκαμε σε αυτές τις δύο επειδή μπορούσαμε να τις εξερευνήσουμε χωρίς να κάνουμε βαριές υποθέσεις σχετικά με το σύνολο δεδομένων και τη διαδικασία δημιουργίας των δέντρων. Ακόμα και για τον ίδιο αριθμό χαρακτηριστικών, δημιουργούνται διάφορα δέντρα, δεδομένου ότι η επιλογή των χαρακτηριστικών είναι τυχαία. Αυτά τα δέντρα έχουν διαφορετικούς μεγέθη και δομές, επομένως η οριοθέτηση στην ανάπτυξή τους οδηγεί σε σημαντική μείωση της ακρίβειας. Επίσης, αυτό το όριο θα ήταν ανεπιτυχές επειδή η δημιουργία του δέντρου είναι μη-ντετερμινιστική. Ο λόγος διαίρεσης που επιλέχθηκε, βασισμένος σε γενικές πρακτικές, είναι 70/30. Το 70% του συνόλου δεδομένων χρησιμοποιήθηκε για την εκπαίδευση και το άλλο 30% για τον έλεγχο.Οι μετρικές που αξιολογήσαμε ήταν η Ακρίβεια (Accuracy), η Ανάκληση (Recall), η Ακρίβεια (Precision) και το F-σκορ.

Εκπαιδεύσαμε τα μοντέλα χρησιμοποιώντας τη γλώσσα προγραμματισμού Πψτηον, όπως αναφέρθηκε προηγουμένως, με διάφορους συνδυασμούς των εκτιμητών και των χαρακτηριστικών. Για κάθε ένα από αυτά τα μοντέλα, καταγράψαμε τις μετρήσεις F-σκορ κατά τη διάρκεια του συμπερασμού στο σύνολο δεδομένων ελέγχου. Αποκλείουμε τα μοντέλα με F-σκορ ˂ 99% από τη συνεχόμενη ανάλυσή μας για να διατηρήσουμε αυτά με την καλύτερη προβλεπτική ικανότητα.

Μεταφέραμε την Python υλοποίηση σε C++ κωδικά κατάλληλο για να μπορεί να εκτελείται μέσω του Vitis. Αποθηκεύουμε την δομή των Δέντρων Απόφασης σε πίνακες. Για την ανακατασκευή σε C++, θα χρειαστούμε πέντε συγκεκριμένους πίνακες: children left, children right, feature, threshold, και value.Περεταίρω αλλαγές πρέπει να γίνουν ώστε να εκτελείται ο κώδικας χρησιμοποιώντας το HLS. Δεδομένου ότι η εντολή while δεν υποστηρίζεται από το εργαλείο, πρέπει να δημιουργηθεί ένας βρόγχος φορ αντί αυτού. Το αναμενόμενο όριο του βρόγχου for είναι το μήκος της διαδρομής μέχρι να φτάσουμε σε ένα φύλλο. Το μήκος της μακρύτερης διαδρομής θεωρείται ως ένα άνω όριο, διότι είναι δύσκολο να προβλέψουμε ποια από

όλες τις πιθανές διαδρομές θα ακολουθηθεί. Το μήκος της μακρύτερης διαδρομής υπολογίζεται χρησιμοποιώντας τον αλγόριθμο DFS.Στον κώδικα, χρησιμοποιήσαμε διανύσματα (vectors) επειδή κατά την εξερεύνηση του χώρου σχεδίασης, ένα από τα παράμετρα που ρυθμίζουμε είναι ο αριθμός των χαρακτηριστικών. Ωστόσο, η δυναμική εκχώρηση μνήμης δεν υποστηρίζεται στο Vitis, επομένως αντικαταστήσαμε τα διανύσματα με στατικά διαμορφωμένους πίνακες.

1.6 Αρχιτεκτονικές βελτιστοποιήσεις του Random Forest βασισμένες στο HLS

1.6.1 Παραλληλισμός εσωτερικά του Random Forest

Στόχος της διπλωματικής εργασίας είναι η επιτάχυνση κατά το inference. Προκειμένου να το πετύχουμε το παραπανω αντιμετωπίσαμε αρχικά 2 κύριες προκλήσεις. Η πρώτη πρόκληση είχε να κάνει με το να βρούμε ποία κομμάτια του αλγορίθμου να επιταχύνουμε και πώς αυτό μπορούμε να το υλοποιήσουμε σε αποτελεσμάτικο κώδικα για το εργαλέιο Vitis . Η δεύτερη πρόκληση αφορούσαι στην παραμετροποιήση του παραπάνω κώδικα ώστε να έχουμε το βέλτιστο αξιοποιήση πόρων.

Η πρώτη πρόκληση είναι να μπορέσουμε να μετατρέψουμε την ακουλιακή εκτέλεση του RF αλγορίθμού σε παράλληλη. Αρχικά έπρεπε να εντοπίσουμε ποιά τμήματα του αλγορίθμου μπορουμε να παρηλλοποιήσουμε. Στη συνέχεια έχοντας βρει ποιά είναι αυτά τα κομμάτια προτείνουμε αλλαγές στο δομή του κώδικα καθώς και εισάγουμε βοηθιτικές εντολές υπό την μορφή pragmas για να μπορέσουμε όντως να πετύχουμε την επιθυμητή παραλληλια. Οι αποφάσεις παίρνονται με ψηφοφορία απο τους διαφορετικούς και ανεξάρτητους μεταξύ τους εκτιμητές. Δεν υπάρχει κάποια εξάρτηση στη πρόβλεψη που κάνει κάθε DT, όποτε δεν υπάρχει ανάγκη για σειριακή εκτέλεση των DT. Αντίθετα προτείνουμε, το inference των DT να γίνεται παράλληλα

Προκειμένου να καταλάβει το Vitis εργάλειο ότι θέλουμε παράλληλη εκτέλεση των εκτιμητών, χρειάστηκε να εφαρμόσουμε pragmas. Πιο συγκεκριμένα προσθέσαμε το pragmas dataflow. Η δήλωση pragma DATAFLOW ενεργοποιεί τον παραλληλισμό σε επίπεδο εργασιών, επιτρέποντας σε συναρτήσεις και βρόγχους να επικαλύπτονται στη λειτουργία τους, αυξάνοντας τον βαθμό συγχρονισμού της υλοποίησης RTL και αυξάνοντας τον συνολικό ρυθμό επεξεργασίας του σχεδιασμού. Η βελτιστοποίηση DATAFLOW επιτρέπει στις λειτουργίες ενός βρόγχου ή μιας συνάρτησης να ξεκινήσουν τη λειτουργία τους

πριν ολοκληρωθούν όλες οι λειτουργίες του προηγούμενου βρόγχου ή συνάρτησης, υπό τον όρο ότι δεν υπάρχουν εξαρτήσεις δεδομένων.

Πετυχαίνουμε παράλληλη εκτέλεση των decision trees εντός ενός rf δηλώνοντας πολλές φορές την ίδια συνάρτηση και εφαρμοζοντας τα απαραίτητα pragmas όπως το dataflow. Η δεύτερη πρόκληση που κλήθηκαμε να αντιμέτωπίσουμε είχε να κάνει με το να διατηρήσουμε την παραπάνω παραλληλία αλλα να μειώσουμε σημαντικά του πόρους που απαιτούνται για την πετύχουμε. Το κύριο πρόβλημα πηγάζει από τον τρόπο που αποθηκεύουμε τους πίνακες των decision trees και επακόλουθα με τον τρόπο που τους διαβαζουμε. Για να το αντιμετωπίσουμε χρειάστηκε να μετακινήθουμε από το την γενικευμένη μορφή κώδικα που επιτρέπει iterative access/execution και να εστιάσουμε σε λύσεις με hardcoded συναρτήσεις για κάθε decision tree που επιφέρουν σημαντική μείωση στην αξιοποιήση πόρων.

1.6.2 Παραλληλισμός μεταξύ πολλών Random Forest

Προηγουμένως αναλύσαμε την μετατροπή από σειριακή εκτέλεση των εκτιμητών σε παράλληλη. Ωστόσο μπορούμε να πετύχουμε παραλληλία και σε άλλα κομμάτια του αλγορίθμου RF. Κάθε δείγμα εισόδου είναι ανεξάρτητο από τα υπόλληλα. Με δεδομένο αυτο εξερευνήσαμε δυο διαφορετικές προσεγγίσεις στην αρχιτεκτονική του kernel. Η πρωτη αφορά την παράλληλη εκτέλεση διαφορετικών αντιγράφων του instance του random forest, όπου το καθενα θα κάνει inference διαφορετικό δείγμα. Η δεύτερη πρόταση είναι μια pipeline προσέγγιση με στόχο να πετύχουμε Iteration Interval (II) ίσο με 1. Δηλαδή σε κάθε κύκλο ρολογιού να επεξεργάζεται μια καινουργια είσοδο, ή αντίστοιχα να παράγει μια πρόβλεψη ανά ένα κύκλο.

Η πρώτη αρχιτεκτονική είναι η Coarse Grained Design Parallel RFs. Δημιουργώντας πολλαπλά και ανεξάρτητα αντίγραφα της συνάρτησης ρανδομ φορεστ δίνεται η δυνατότητα για ταυτόχρονη και παράλληλη επεξεργασία εισόδων. Συγκεκριμένα για K σε πλήθος παράλληλα RFs εσωτερικά το καθένα έχει N παράλληλα. Η αρχιτεκτονική εισάγει ένα έξτρα επίπεδο παραλληλίας, πέρα αυτού των παράλληλων εκτιμητών καθώς έχει την δυνατότητα να επεξεργάζεται ταυτόχρονα K εισόδους.Ακουλήθησαμε την ίδια μεθοδολογία όπως και με τους εστιματορς για να πετυχουμε παραλληλία μεταξύ των rfs. Οπως και προηγουμένως χρησιμοποιουμε τα pragma dataflow και array_parition, για να διασφαλίσουμε την παράλληλη επικαλυπτομενη εκτέλεση.

Προκειμένου να έχουμε αξιόπιστες μετρήσεις τόσο για το χρόνο

inference αλλά και για την αξιοποιήση πόρων κανάμε hw implementation. Όπως είναι αναμενόμενο δημιουργώντας αντίγραφα από RFs πετυχαίνουμε παραλληλία αλλά χρησιμοποιούμε περισσότερους πόρους. Οι πόροι ανά FPGA, σε BRAM, FF και LUT, είναι καθορισμένοι για αυτό πρέπει να βρούμε πόσα αντίγραφα μπορούμε να χωρέσουμε. Ορίσαμε ενα άνω όριο για την χρήση πόρων που θεωρούμε ικανοποιητικό και δε θα θελαμε να υπερβούμε για να διασφαλίσουμε την ορθή λειτουργία του FPGA. Το όριο αυτό είναι 80% των προαναφερθέν πόρων. Κάναμε δυαδική αναζήτηση μέχρι να βρούμε το μέγιστο πλήθος RFs που η χρήση τους δεν θα ξεπερνά 80%. Κατά την εξερεύνηση δοκιμασαμε 1, 2, 4, 8, 16, και 24 παράλληλα RFs.

Παρατηρούμε ότι ο πόρος που μεταβάλλεται περισσότερο είναι η BRAM. Συμπεραίνουμε ότι για κάθε νέο αντίγραφο του RF, δεσμευονται εκ νέου πόροι για τους πίνακες των decision trees. Ένα πιθανός λόγος για αυτή την συμπεριφορά του εργαλείου είναι οι περιορισμένες ports (σε σύνολο 2) για κάθε bram. Είναι δύσκολο να γίνει ο συντονισμός για πρόσβασγ στους ίδιους πίνακες από πολλαπλά RFs. Εντείνει την παραπάνω δυσκολία το γεγονός ότι δεν υπάρχει κάποιο μοτίβο για το ποιά θέση του πίνακα επιδιώκουμε να διαβάσουμε, καθώς έχουμε τυχαία δίασχυση που εξαρτάται από την είσοδο. Όποτε δεν επιτυγχανεται κάποιος διαμοιρασμός των πινάκων με μόνη λύση του εργαλείου την αντιγραφή τους. Δεδομένου αυτό περιμένουμε να υπάρχει γραμμική αύξηση των πόρων σε σχέση με το πλήθος των RFs, όπως και γίνεται

Η δεύτερη προτεινόμενη αρχιτεκτονική είναι η Pipelined Random Forest. Κάθε εκτιμητής μπορει να μπορεί να αναπαρασταθεί ως μια σειρά από S στάδια, με το S να είναι ίσο με μακρύτερο μονοπάτι κάθε δέντρου. Στόχος είναι να μετατραπεί η σειριακή εκτέλεση των S σταδίων σε πιπελινε. Αυτό μπορεί να γίνει τοποθετώντας καταχωρητές μεταξύ των σταδίων. Σε κάθε κύκλο τροφοδοτείται με νέο δείγμα καί ύστερα από την αρχική καθυστέρηση και επιτυγχάνεται II=1.

Δείξαμε ότι ένα decision tree μπορεί να αναπαρασταθεί ως ένα pipeline. Μεταφέρουμε το παραπάνω στο rf, όπου αποτελεί μια συλλογή από ανεξαρτητα dt. Όποτε η προτεινόμενη αρχιτεκτονική απαρτίζεται από ανεξάρτητα εσωτερικά pipelines, τα οποία συνδυάζονται και επιτυγχανεται σε υψηλό επίπεδο η rf να είναι και αυτή pipeline. Για να το πετύχουμε αύτο χρησιμοποιήσαμε το pragma pipeline του Vitis

1.7 Εξερεύνηση με εφαρμογή Precision Scaling

Ο κυρίαρχος πόρος, δηλαδή εκείνος που αλλάζει περισσότερο και καθορίζει σε μεγαλύτερο βαθμό αν μπορεί να υλοποιήθει το σχέδιο, είναι το BRAM. Στόχος λοιπόν είναι διατηρώντας σταθερή την ακριβεια του μοντέλου να μειώσουμε το ποσοστό χρήσης των BRAM. Το καταφέραμε μέσω της τεχνικής Precision Scaling. Πιο συγκεκριμένα, μειώσαμε το πλήθος από bits που απαιτείται για την αναπαράσταση των στοιχείων των πινάκων των dt. Αθροιστικά η παραπάνω μείωση μπορεί να έχει μεγάλη επίδραση ιδίως για μεγάλα σχέδια, όπως τα 24 παράλληλα RFs, πετυχένοντας πτώση απο τα 70,69 σε 39,88 BRAM(%).

Το εργαλείο κάνει αυτόματα την μετατροπή από 32-int σε μικρότερου τύπου int. Δεν αναφέρεται κάπου τον τύπο που εντέλει καταλήγει (στα πλάσια που καταφέραμε να ελεγξουμε) αλλά έχει την ίδια απόδοση με τους τύπους ap_uint που ορίσαμε για να ελέγξουμε την παραπάνω υπόθεση. Δεν συμβαίνει και το ίδιο για πίνακα threshold όπου έχει τύπο float. Λόγω της δομής με την οποία αποθηκεύεται ένας float αριθμός δεν είναι δυνατόν να γίνουν αυτόματα βελτιστοποιήσεις σαν την παραπάνω. Δοκίμασαμε δύο διαφορετικές αναπαραστάσεις για τον threshold πίνακα τις : fixed point και integer.Το εργαλείο ΐτις δίνει την δυνατότητα να χρησιμοποιούνται arbitrary precision data types για fixed float και integer.

Κατά την εκπαίδευση των μοντέλων μας το σύνολο των δεδομένων ήταν κανονικοποιημένα μεταξύ (0,1). Όποτε δεν υπάρχει ακέραιο τμήμα και μπορούμε να εστίασουμε καθαρά στο δεκάδικο. Δεδομένου αυτού ο fixed point τύπος θα διαθέσει όλα τα bits, όσα αποφασίσουμε εμείς, μόνο για το δεκαδικό κομμάτι. Η εξερεύνηση που θα κάνουμε αφορά το πλήθος αυτών των bits. Συγκεκρίμενα θα εξετάσουμε 8, 16, 24 και 32 βιτς.

Η fixed point αναπαράσταση μας ώθησε να πειραματήστουμε και με άλλες πίθανες αναπαραστάσεις για τους threshold πίνακες. Αρχικά σκεφτήκαμε την integer αναπαράσταση ορμώμενη από το γεγονός ότι οι πράξεις και συγκρίσεις με integer είναι πιο γρήγορες σε σύγκριση με τους float χάριν της απλούστερης αναπαράστασης. Επιπλέον διαπίστωσαμε στην συνέχεια ότι πέρα της επιπλέον επιτάχυνσης, με integer αναπαράσταση σε ιδιως για χαμηλό precision όπως 16 και 8 bits, το μοντέλο έχει σημαντικά καλύτερή ακρίβεια εν συγκρίση με τους fixed point.

Η μετατροπή ενός float σε integer διατήρωντας τα χαρακτηριστικά την κατανομή τους γίνεται με την τεχνική του κβαντοποιήσης.Η

κβαντοποίηση είναι μια τεχνική για τη μείωση των υπολογιστικών και μνήμης δαπανών κατά την εκτέλεση της πρόβλεψης, αναπαριστώντας τα βάρη και τις ενεργοποιήσεις με χαμηλής ακρίβειας τύπου δεδομένων, όπως το 8-bit integer (int8) αντί του συνηθισμένου 32-bit floating point (float32). Το πλάνο κβαντοποίησης που εφαρμόσαμε είναι βασισμένο στη συνάρτηση εύρους . Η γραμμική κβαντοποίηση μετατρέπει τις κινητές υποδιαστολές σε ακέραιους χρησιμοποιώντας έναν παράγοντα κλίμακας. Η κβαντοποίηση βασισμένη στο εύρος υπολογίζει αυτόν τον παράγοντα βάσει των πραγματικών τιμών του αντικειμένου, πιθανώς αποκλείοντας τις ακραίες τιμές, ενώ άλλες μέθοδοι χρησιμοποιούν σταθερά ή εκμάθησης κατώφλια για το κλείσιμο των τιμών. Εμείς εφαρμόσαμε τον συνδυασμό των δύο και συγκεκριμένα συμμετρική κβαντοποιήση.

Με το precision scaling συμφωνούμε σε ένα trade off αξιοποιήσης πόρων και απόδοσης με την ακρίβεια του μοντέλου. Συγκρίνουμε το F-σκορ για inference σε όλο το dataset του αρχικού (float) μοντελου με τα αντίστοιχα fixed_point και fixed_int για 8,16,24 και βιτς. Για πλήθος 32 και 24 bits η ακρίβεια παραμένει σταθερή και για τι δύο αναπαραστάσεις. Όμως για 8 και 16 bits το fixed_point μοντέλο χανεί σημαντικά την προβλεπτικη ικανότητα. Καλύτερη συμπεριφορά υποδεικνύει το κβαντισμένο μοντέλο. Για 16 bits πρακτικά μένει σταθερό το F-σκορ και για 8 bits μειώνεται ελάχιστα. Προχωρόντας παρακάτω εφαρμόσαμε precision scaling μόνο για 24 και 32 στους float_fixed και 16,24 και 32 στους fixed_int.

1.8 Πειραματική Αξιολόγηση

Πριν προχωρήσουμε στη πειραματική αξιολόγηση ωφείλούμε να εξηγήσουμε πως επιλέξαμε το μοντελο δηλαδη τον συνδιασμό εκτιμητών και αριθμό χαρακτηριστικών, για το οποίο κάναμε το hw εξερεύνυση όπως το περιγράψαμε στα κεφάλαια. Χρησιμοποιήσαμε μια ευριστική ευριστική για να καταλήξουμε σε ενα μοντέλο. Συγκεκριμένα για τα μοντέλα που εξετάσαμε κατά την εκπαίδευση και έιχαν F-σκορ ’ 99 κάναμε hw emulation με παραμέτρους αρχητεκτονικής coarse grained parallel rf με RF_SIZE=1 και χώρις precision caling. Από τις 37 παραγώμενες υλοποιήσεθς και βρήκαμε ποιά ήταν πάνω στο pareto front. Η ευριστική που χρησιμοποιήσαμε για καταλήξουμε στο βέλτιστο μοντέλο ήταν απο αυτά που βρίσκονται στο pareto front να διαλέξουμε εκείνο με υψηλότερο F-σκορ, όπου και ήταν το μοντέλο με 10 εκτιμητές εκπαιδευμένο με 32 χαρακτηριστικά.

Εξετάσαμε την επίδραση του precision scaling στην για την pipeline και parallel αρχιτεκτονική. Δηλαδή 10 εκτιμητές εκπαιδευμένους με 32 χαρακτηριστικά, με τα παραγόμενο μοντέλα να έχουν pipeline αρχιτεκτονική και parallel αρχιτεκτονική με 1,2,4,8,16 και 24 παράλληλα RFs . Επιπλέον εφαρμόσα μετατροπή από float 32 σε fixed_float για 32 και 24 bits και σε fixed_int για 32,24 και 16 bits.

Στο βοξ πλοτ συμπεριλάβαμε τις μετρήσεις για φλοατ και τους παραπάνω τύπους για πρεσισιον σελινγ. Για μπορέσουμε να συμπτίξουμε σε μια μετρική την επίδραση του πρεσισιον σσαλινγ συλογικά στα ρεσουρσες του ΦΠΓΑ παραμετροπιήσαμε τον τύπο απο το εξής παπερ σύμφωνα με τον ακόλουθο τύπο. Αφαιρέσαμε το ΔΣΠ υτιλιζατιον μιας και το μοντέλο μας δεν χρησιμοποιήσε καθόλου ΔΣΠς.

Για να μπορέσουμε να συμπεράνουμε τα καλύτερα configurations λαμβάνοντας υπόψη το inference time και το resource utilization σχεδίασαμε pareto plots. Το y-axis παρέχει το inference time κάθε configuration και το x-axis το resource utilization. Επιλεξαμε να σχεδίασουμε μόνο για το BRAM μίας και είναι το πιο δομιναντ και εξαντλειται γρηγοροτερα. Σχεδίασαμε διαφορετικά διαγράμματα για κάθε μέγεθος εισόδου, ώστε να μπορούμε να βρούμε τα καλύτερα configurations για μικρά και μεγάλα test datasets. Όπως τα inputs sizes είναι 480,4800,48000,28334 (όλοκληρο το δατασετ). Επίσης συμπερίλαβαμε το unptomized RF design.

Στα pareto διαγραμματα σχεδιάσαμε και τα configurations που βρίσκονται στο pareto front. Από αυτό του pareto fronts σχετικά με το performance για όλες τις περιπτώσεις εκτός για μικρό input size 480 samples το πιο γρήγορο μοντέλο εχει pipeline αρχιτεκτονική. Σε δεύτερη θέση είναι το parallel μοντέλο με 24 παράλληλα RF. Κοινό χαρακτηριστικό σχεδόν όλων των configurations στα pareto fronts είναι ότι εκείνα που είναι πιο γρήγορα και έχουν χαμηλότερο BRAM utilization έχει εφαρμοστεί precision scaling με integer 16 bits. Συμπεραίνουμε ότι η εφαρμογή του precision scaling έχει σημαντική επίδραση στην μείωση των πόρων αλλά βελτιώνει το performance. Για την σύγκριση με c++ και python επιλέγουμε τα γρηγορότερα configurations από τις δύο αρχιτεκτονικές δηλαδή pipeline fixed integer 16 bits και parallel 24 rf integer 16 bits.

Από τα πειράματα μας μπορούμε να εξάγουμε χρήσιμα συμπεράσματα για την χρησιμότητα της hw επιτάχυνσης. Για όλα τα μεγεθή εισόδων καταφέρνουμε να πετύχουμε επιτάχυνσης. Την μικρότερη επιτάχυνσης 1.5 την έχουμε για το μικρό μέγεθος dataset (480 δείγ-

ματα) σε σύγκριση με το inferece σε c++. Η εικόνα αυτή αλλάζει όσο μεγαλώνει το μέγεθος εισόδου. Για όλες τις υπόλοιπες περιπτωσεις 4800,48000 δείγματα και όλοκληρο το dataset καταφέρνουμε να πετύχουμε πολύ καλύτερή επιτάχυνση. Το καλύτερο ςονφιγυρατιον για αυτά μεγέθη είναι η pipeline αρχιτεκτονική με precision scale integer σε 16 bits, με μόνη εξαίρεση τα 480 δείγματα όπου η parallel υλοποιήση 24 RF πετυχαίνει καλύτερο χρόνοο.Πίο είδικα το speed up που πετύχαμε φαίνεται στον πίνακα. Ενδιαφέρον έχει ότι για inference σε python ο χρόνος μένει πρακτικά σταθερός γύρω στα 140 ms. Λόγω αυτού όσο μεγαλώνει το μέγεθος του test για evaluation το inference time της python προσεγγίζει αυτό της c++, με την περίπτωση για περισσότερα από 48000 να έχει καλύτερη απόδοση. Οι υλοποιήσεις μας κάθε φορά συγκρίνονται με το γρήγορο από τα δύο (c++,python) και αντίστοιχα προκύπτει το ανάλογο speed up. Το καλύτερο speed up 20,03 το πετυχαμε για 48000 δείγματα.

input size	compare to	native time(ms)	best fpga time(ms)	speed up
480	c++	1.47	0.99	1.5
4800	c++	14.07	1.64	8.6
48000	python	141.61	7.07	20
288344	python	141.8	37.15	3.8

Table 1: Achieved speed up.

Όλα τα παραπάνω μας οδηγούν στα εξής συμπεράσματα ότι για μικρά input sizes γύρω στα 500 samples δεν έχουμε σημαντικά οφέλη από το hw acceleration. Το μέγεθος είναι αρκετα μικρό ώστε το transfer time overhead από το host στο fpga να υπερτερεί οποιουδήποτε acceleration που πετυχαίνουμε χάρις στην παραλληλία των υλοποιήσεων μας. Θα μπορούσε σε αυτή την περίπτωση το inference να γίνει natively σε c++ χωρίς κάποια σημαντική διαφορά. Όσο μεγαλώνει το input size είναι ξεκάθαρο ότι το παραπάνω overhead σκίαζεται από τα acceleration που πετυχαινουμε. Το ιδανικό input size έιναι γύρω στα 50000 samples όπου και τα μοντέλα μας επιδυκνύουν και το καλύτερο performance. Τέλος για input sizes πολλαπλάσια όλοκληρου του dataset είναι πιθανόν η python υλοποιήση να πετυχαίνει καλύτερα αποτελέσματα λόγω του batch processing . Όμως λίγες μπορούμε να φανταστούμε είναι οι περιπτώσεις όπου θα χρειαστεί να γίνει inference σε τόσο μεγάλο dataset για security critical introduction detection εφαρμογές όπως αυτή που εξετάσαμε σε αυτη την διπλωματική ερ-

γασία.

1.9 Συμπεράσματα και Μελλοντική Εργασία

Στα πλαίσια αυτής της διπλωματικής εργασίας εξέτασαμε τον αλγόριθμο μηχανικής μάθησης Random Forest για να κάνουμε Anomaly detection σε Βιομηχανικές Εγκαταστάσεις. Ερευνήσαμε τρόπους με τους οποίους μπορούμε να πετυχουμε hardware acceleration σε FPGA του RF κατα το inference. Εντοπίσαμε τα κομμάτια του αλγορίθμου που μπορούμε να επιταχύνουμε και να παραλληλοποιήσουμε εκμεταλλευόμενοι τις δυνατότητες τις δυνατότητες του FPGA. Τα decision trees που απαρτίζουν το RF ειναι ανεξαρτητα μεταξυ τους και μπορούν να παραλληλοποιηθούν. Στηριζόμενοι σε αυτή την ιδιότητα προτείναμε δύο διαφορετικές αρχιτεκτονικές τις Coarse grained Parallel RFs και Pipeline RF, στο εσωτερικό και των δύο οι estimators εκτελούνται παράλληλα. Στην πρώτη έχουμε πολλαπλά αντίγραφα του ίδου RF, εισάγοντας ένα δεύτερο παραλληλίας αυτό την παράλληλης ταυτόχρονης επεξεργασίας πολλαπλών δειγμάτων εισόδου, ίσο σε αριθμο με αυτών των αντιγράφων. Στη δεύτερη αρχιτεκτονική ακολουθήθηκε pipeline προσέγγιση όπου ύστερα από ένα αρχικό latency επιτυγχανεται iteration interval II=1, δηλαδή παραγωγή ενός νέου classification σε κάθε νέο κύκλο. Επιπλεον εισάγαμε την τεχνική βελτιστοποίησης precision scaling με σκόπο να μειώσουμε το resource utilization και καθώς και το inference time. Συγκρινόμενοι κάθε φορά με την πίο γρήγορη υλοποίηση του RF που έτρεχε natively σε C++ και Python καταφέραμε να πετύχουμε acceleration για όλα τα samples sizes που δοκιμάσαμε. Το καλύτερο speed up που μετρήσαμε ήταν 20 για 48000 samples. Παράγων της διπλωματικής εργασίας ήταν η δημιουργία ενός εργαλείου για την αυτοματοποίση της διαδικασίας design exploration σε FPGA, απαιτώντας ελάχιστη αλλεπίδραση με τον χρήστη.

Σε μελλοντική εργασία, επιθυμούμε να ευρήνουμε περισσότερες τεχνικές για acceleration σε ΦΠΓΑς ή ακόμη διαφορετικές hardware πλατοφορμες όπως τα embedded fpgas. Η έρευνα της διπλωματικής εύκολα μπορεί να μεταφερθεί και σε eFPGA, τα οποία αποκτούν δημοφιλία σε πληθώρα εφαρμογών στενα συσχετιζόμενη με την δική μας όπως είναι το edge computing. Μια άλλη ενδιαφέρουσα μελέτη θα μπορούσε να αποτελεί το acceleration ενός άλλου tree structure machile learning αλγορίθμου κατάλληλο για anomaly detection τα Isolation Forests. Τα IF θυμιζουν αρκετα το RF με την βασικότερη διαφορά τους ότι

μπορούν να κάνουν και unsupervised anomaly detection. Ενδιαφέρον θα ήταν η σύγκριση των δύο αλγοριθμων σε hw acceleration performance αλλα και προβλεπτική ικανότητα. Συμπέρασμα αυτής της διπλωματιής εργασίας είναι ότι τα πεδία του HW accelartion και Anomaly detection διαρκώς αναπτύσσονται και θα μας απασχολούν σίγουρα και στο μέλλον.

2 Introduction

In recent years, it has been observed that more and more Industrial Facilities are becoming the target of attacks. Many of these systems were not designed with security in mind. Until recently, operation technologies were separated from information technologies. In addition, specialized knowledge was required to handle and properly operate them[7]. Because of this, industrial plants were safer than they are today. The ever-increasing growth and demand for Internet of Things(IoT) technology has reinforced the above phenomenon. Despite the clear benefits, in terms of productivity, efficiency and automation, offered by IoT integration and interconnection of systems, there is a risk of exposing more vulnerabilities.

In order to protect industrial facilities from attacks, we need to quickly detect them and develop deterrents. The aim of this thesis is to accelerate the inference stage of a predictive model that detects cyber-physical attacks to a Secure Water Treatment Facility. To achieve this we apply Anomaly Detection utilizing machine learning algorithms such as Random Forest ensemble. We chose to accelerate the generated model on a FPGA device with the help of Vitis HLS tool.

Anomaly detection, also known as outlier detection, refers to the problem of finding patterns in data that do not conform to expected behavior [1]. We focus mainly at intrusion detection, the process of identifying hostile activity (break-ins, penetrations, and other types of computer abuse) in a system [1] . Using historical data we train the Random Forest model to make such predictions. Random Forest is a supervised learning algorithm which consists of a collection of Decision Trees. Each Decision Tree is trained independently of the others on a subset of the original dataset. To obtain a prediction, each tree independently makes its own prediction and then these are aggregated. In our case we are interested in classification, the final prediction is extracted by majority voting.

One of the main challenges related to anomaly detection implementation is fast and reliable inference. In our thesis we propose to use Field Programmable Gate Arrays (FPGA) and the Vitis-HLS tool-chain. The parallelism capabilities offered by FPGAs allows us to significantly reduce the inference time, which is essential for security critical applications like ours. Furthermore, through Vitis

we can write our code in a high-level language like C++ rather than a hardware description language such as Verilog or VHDL. It provides us with the flexibility and convenience to be able to focus on improving the design and the structure of the algorithm rather than on low-level implementation.

In the context of this thesis, we explored the possibility of parallelizing the RF algorithm to achieve acceleration. Inside an RF, each Decision Tree operates independently from the others, constituting the first level of parallelism we leveraged. To achieve this, we created different instances for each Decision Tree in the hardware, enabling their parallel execution. We also investigated higher levels of parallelism, specifically parallel processing of multiple input samples. We proposed two different architectures. The first one, named Coarse-Grained Multiple RF, involves instantiating multiple copies of the same RF, with each containing parallel decision tree estimators. This allows us to implement, for example, K parallel RFs, each with N parallel Decision Trees, internally evaluating K inputs simultaneously. It's clear that there is a trade-off between performance and resource utilization, for which we conducted an analysis.

The second proposed architecture is called Pipelined RF. We observed that the process of traversing a Decision Tree can be fragmented into individual sequential stages. While there's a dependency between the stages, by inserting buffers between them, we managed to transform the process into a pipeline. We combined parallel and independent pipelined Decision Trees to achieve a high-level Pipelined RF. The Pipelined RF can accept new input samples in each cycle, which are then propagated through the pipelined stages of each Decision Tree, achieving an initial Iteration Latency and then reaching an Iteration Interval of II=1.

In both architectures, we applied precision scaling techniques to achieve better performance during inference and lower resource utilization. Precision scaling is the technique of representing model elements in a different and lower precision format. We experimented with different numbers of bits lengths and data types beyond floating-point, such as fixed floats and integers. In the case of integers, we needed to apply quantization.

Having the two architectures and the aforementioned microarchitectural optimization techniques, we conducted hardware design

exploration to find the best configurations for comparison with C++ and Python implementations of the RF algorithm. We analyzed the findings from these comparisons and presented our conclusions in detail. A byproduct of this thesis is the creation of a framework tool capable of automating the design exploration process. The contribution of this thesis lies in the study of novel techniques for accelerating the RF algorithm on FPGA and providing the aforementioned tool.

The rest of the thesis has the following structure. In chapter 3 we report on related work on Anomaly Detection and acceleration in FPGA. We make an extensive analysis of the Random Forest algorithm and compare it with other state of the art classifiers. We also analyze the structure and the capabilities of the FPGA in relation to other platforms and the operation of the Vitis HLS. In chapter 4, we present the dataset we used and the methodology we followed to produce the first executable on FPGA. In chapter 5, we present the two architectures parallel and pipeline we implemented and the HLS directives we used to achieve the best predictive accuracy with maximum acceleration and lowest resource usage. In chapter 6, we present micro architectural optimizations, such as precision scaling, we applied to improve further the aforementioned architectures regarding the performance and resource utilization.In chapter 7 we present a framework that was produced through the thesis in order to automate the design exploration process in FPGA for the random forest algorithm.In chapter 8 we analyse the results from the design exploration of the previous chapters and we select the best ones implementations to compare with the equivalents C++ and Python that run natively. Finally, in chapter 9 we summarize with our conclusions and discuss further ideas for future optimizations.

3 Theoretical Background

3.1 Related Work

Anomaly detection is a widely found issue. Many different approaches have been tried in an attempt to develop a reliable model. One of them involves the comparison of Support Vector Classifires(SVM) and Random Forest(RF)[8]. Simon D. Duque Anton used 2 different datasets involving intrusion detection in industrial facilities. They also did feature selection in order to keep the most important features, with the aim of reducing training time while maintaining a relatively high accuracy rate. There is no straight forward solution in finding those features that are able to distinguish between malicious and no malicious instances.One suggested heuristic is the feature importance metric of RF, which shows us the importance of a feature according to its ability to increase the pureness of the leaves. The authors of the paper concluded that RF achieved significantly better performance and accuracy than SVM and had more linear behavior with respect to run-time.

A similar study [9] comparing SVM and RF for ID on the KDD99 dataset had different conclusions. Specifically, RF achieves better prediction for specific type of attacks such as probing and U2R and has higher precision than SVM. However, SVM has overall slightly better accuracy and lower False Negative Rate(FNR). Both of papers agree that RF is faster at training and running.

In another study [10], they sought to develop an intelligent and secure model to detect vulnerabilities in IoT systems and protect against cyber-attacks.To achieve this, they compared several different machine learning algorithms. In total 5 were used, which were Logistic Regression(LR), Support Vector Machine, Decision Trees(DT), Random Forest and Artificial Neural Networks (ANN). They concluded that a simple model like RF can, always depending on the dataset, bring better results in terms of prediction accuracy compared to much more complex ones like ANN.

It is evident that RF, thanks to its good accuracy and fast training and running times, is a popular choice in the field of anomaly detection. However, there is always a need for more precise, faster, and energy-efficient solutions. Therefore, efforts have been made to accelerate the model on a variety of hardware platforms such as GPUs and FPGAs. One such approach is presented in a paper

[11]that utilizes a variant of the classic RF algorithm called Compact Random Forest (CRF). The main differences lie in the number of decision trees (DT) and the maximum depth allowed for each tree to grow. These changes are motivated by the fact that the RF algorithm is memory-bound, which is not ideal for hardware acceleration with limited resources.

Regarding FPGA implementation, a pipeline architecture is suggested with n different pipelines for n trees. Each pipeline consists of s stages, where each stage represents the node that the sample traverses in the corresponding tree. In the s-1 stage, the classification of the sample by the tree takes place. The last stage of the design is the majority voting from all the pipelines. This particular implementation achieves processing of each sample in every clock cycle, meaning an Iteration Interval (II) of 1.

In the CP-GPU implementation, the focus is on maximizing the utilization of parallelism in multicores and achieving optimal data reuse in the caches. To achieve this, each streaming multiprocessor selectively traverses a subset of trees from the forest for each sample. The study concludes that FPGA provides higher performance/performance per watt and can handle larger classifiers without compromising its capabilities, with the necessary hardware resources. On the other hand, CP-GPU offers better cost-effectiveness in terms of efficiency per dollar, although it experiences a significant reduction in its performance when dealing with the maximum size of the Compact Random Forest (CRF). However, the CP-GPU implementation allows for easier scalability.

Based on the previous work, we have decided to perform Anomaly Detection (AD) using Random Forest (RF) on an FPGA. However, we have made some differences compared to [11]. Firstly, we will not impose any restriction on the size of the RF during training. Additionally, we will compare two different architectures on the FPGA. One architecture is a pipeline approach, similar to the one proposed in the paper, while the other involves a more parallel implementation.

Thanks to the increased capabilities of modern FPGAs, we will aim to achieve the maximum possible number of parallel and independent RF instances that can fit on the board. All instances will be identical copies of each other. Internally, the trees will process each sample in parallel in both proposed architectures. Our goal is

to compare which of the two approaches, pipeline or parallel, will provide us with higher performance and to examine the trade-off between resources and performance.

3.2 Random Forest Algorithm

The RF algorithm was first proposed by Leo Breiman. It combines multiple techniques and introduces new ones. It is a variation of the CART algorithm, in which bagging and random feature selection are applied.

First, let's analyze the Classification And Regression Tree (CART) algorithm [12], which consists of 4 basic steps. In the first step, we start from the root and construct the tree by recursively creating nodes and splitting the data. Each node acquires a class based on the distribution of classes in the learning data at that node, as well as the decision cost matrix, regardless of whether the node will be further split or not. The importance of the label of all node will evident at the analysis of stage 3. One of the most common criteria used to find the optimal splitting variable in a node is the Gini index. The Gini index measures the degree or probability of a particular variable being wrongly classified when randomly chosen. It takes values from 0 to 1. If all elements belong to a single class, the index has a value of 0 and is considered pure. Conversely, if the elements are randomly distributed across classes, the index has a value of 1 and is considered impure. As expected, in each node, the feature and value that give us the lowest Gini index are selected. The formula for calculating the Gini index is as follows.

$$Gini = 1 - \sum_{i=1}^{n} (pi)^2 \tag{1}$$

'pi' is the probability of an object being classified to a particular class.

The second stage is the stopping of tree building. This occurs under 3 conditions:

- there is only one instance in a child node
- there many instances at a child node but all of them have the same classification
- the maximum number of nodes has been reached

In the third stage, we proceed with pruning the tree. The maximal tree always has better accuracy on the learning data, which is why there is a risk of overfitting. The purpose of pruning is to obtain simpler trees that can distinguish the correlations between the data and the noise. The technique used is called 'cost-complexity' pruning. Starting from the leaves, we remove nodes and compare the drop in accuracy relative to the complexity of the tree. In the fourth and final stage, we select the optimal tree that resulted from the pool of trees in the third stage. This is usually done by using a different dataset from the training dataset.

The CART algorithm, however, is prone to overfitting the data and performs poorly with outliers. Breiman proposed using the technique called bagging, also known as bootstrap aggregating. In bagging, multiple weak learners are trained in parallel, each on a different dataset generated by random redistribution of the original dataset. The training dataset for each learner (classifier or regressor) is created by randomly selecting N samples with replacement, where N is the size of the original dataset. The goal is to train each tree with a different dataset, to make the model more robust to new samples.

In most datasets, there are certain features that appear to be more influential than others in making decisions regarding node splitting and, consequently, in constructing decision trees. Even with the use of bootstrap sampling, it seems that the generated trees do not differentiate themselves enough to effectively address overfitting. Breiman attempted to introduce more randomness into the model with the goal of reducing the correlation between trees and either increasing or maintaining their predictive ability. He proposed that instead of using the entire range of features at each node, a random subset of features should be utilized. The feature and its corresponding value for splitting the node are selected from this random subset. The number of features considered in this process remains constant for all nodes and all trees, and it is typically a function of the total number of features. One commonly used approach is to select the square root of the total number of features, while another widely employed method is to take the logarithm.

This inclusion of bagging and random feature selection helps to introduce diversity among the trees and enhance the model's ability to generalize, thus addressing the issues of overfitting and improving

overall performance.

The inference process of a RF is the following:

- Input: For each given X sample which we want to make a prediction, the RF takes this feature vector as input.
- Tree Traversal: The input is passed to each decision tree. Starting from the root the sample moves through the tree structure based on the splitting criteria at each internal node. At each internal node, a specific feature and split point are used to determine the direction the sample should follow. The feature's value in the input sample is compared to the split point, and based on the result, the sample moves to the left or right child node. This process continues recursively until the sample reaches a leaf node, which represents a prediction
- Voting or Averaging: The trees collectively make predictions for each input. In binary classification tasks, each decision tree in the Random Forest "votes" for a class label based on the majority class in the leaf node where the input sample lands. The final prediction is made through majority voting among all decision trees. In regression tasks, the predicted values from each decision tree are averaged to obtain the final regression output.
- Output: The RF outputs the final prediction for the input sample, which can be a class label (in classification) or a numerical value (in regression).

The training and the inference,for classification, are visualized at figure 1.

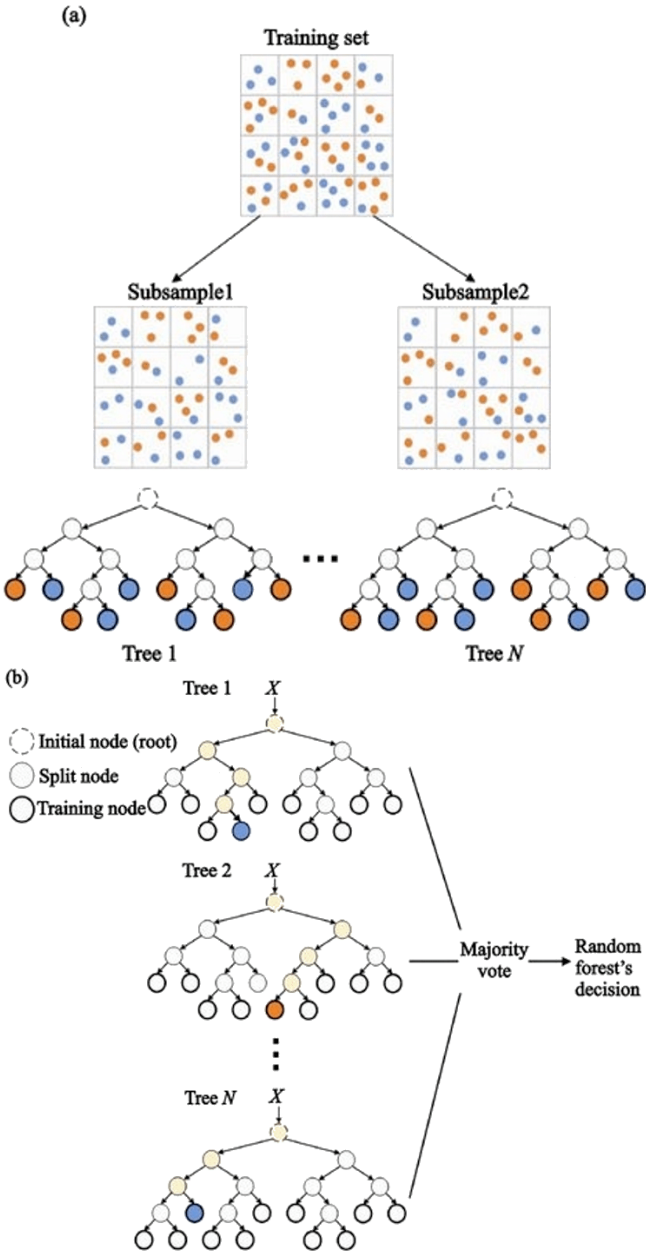

Figure 1: (a) Training process: Multiple decision trees constructed on bootstrap samples of the training set(in this fig there are only two classes shown -orange and blue)(b) Classification process: The classification decision is based on the majority voting results among all the trees.[13]

3.3 FPGA

An Field Programmable Gate Array or FPGA for short, is an integrated circuit (IC) that can be programmed for different algorithms after fabrication. Modern FPGAs consist of up to two million logic cells that can be configured to implement a variety of software algorithms. Although the traditional FPGA design flow is more similar to a regular IC than a processor, an FPGA provides significant cost and time advantages in comparison to an IC development effort and offers the same level of performance in most cases. This is due to the ability of the FPGA, when compared to the IC, to be dynamically reconfigured. This process, which is similar to loading a program in a processor, make changes to the actual hardware and the available resources in the FPGA fabric[15].

3.3.1 Understanding the FPGA Architecture

The general FPGA architecture is heterogeneous compute platforms which consists of three types of modules. They are I/O blocks, Switch Matrix/ Interconnection Wires and Configurable logic blocks (CLB). A basic FPGA architecture is shown at figure 2.

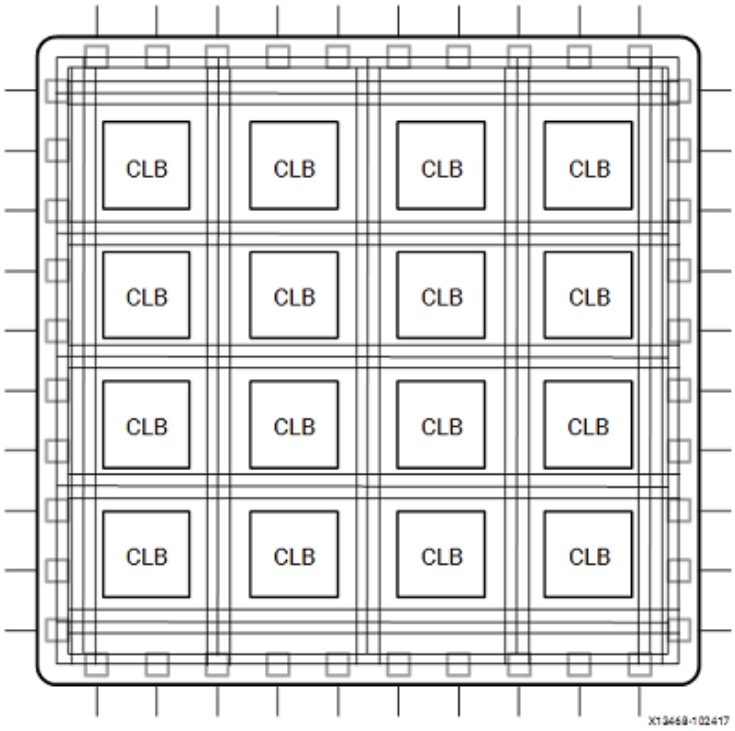

Figure 2: FPGA Architecture[14].

The CLB is the fundamental bulding element for the FPGA, it provides the basic logic and storage capability. In general, a logic block consists of a few logic cells (each cell is called an adaptive logic module (ALM), a logic element (LE), slice, etc.). A typical cell consists of a 4-input LUT, a full adder (FA), and a D-type flip-flop (DFF), as shown to the right. The LUTs are in this figure split into two 3-input LUTs. In normal mode those are combined into a 4-input LUT through the left mux. In arithmetic mode, their outputs are fed to the FA. The selection of mode is programmed into the middle multiplexer. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right, in the figure example. In practice, entire or parts of the FA are put as functions into the LUTs in order to save space. Figure 3 illustrates a simplified logic cell.[16]

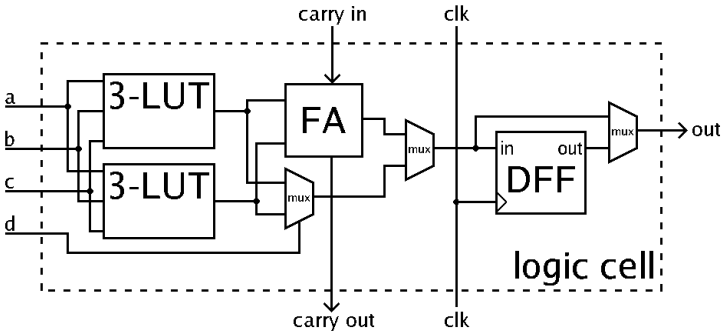

Figure 3: Simplified illustration of a logic cell.[16]

Programmable routing frequently makes up more than 50% of the fabric area and the application's critical path delay. So, it's effectiveness is essential. Pre-fabricated wiring segments and programmable switches make up programmable routing. Any function block output can be connected to any input by programming the appropriate switch sequence to be on [17]. Figure 4 illustrates the fundamentals of the routing configuration.

FPGAs include unique programmable IO structures to allow them to communicate with a very wide variety of other devices, making FPGAs the communications hub of many systems[17]. These days, nearly every Xilinx IP uses an AXI Interface. AXI, which means Advanced eXtensible Interface, is an interface protocol defined by ARM as par of the AMBA (Advanced Microcontroller Bus Architecture) standard[18]. There are 3 types of AXI4-Interfaces (AMBA 4.0), also shown at 5:

- AXI4 (Full AXI4): For high-performance memory-mapped requirements.
- AXI4-Lite: For simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4-Stream: For high-speed streaming data.

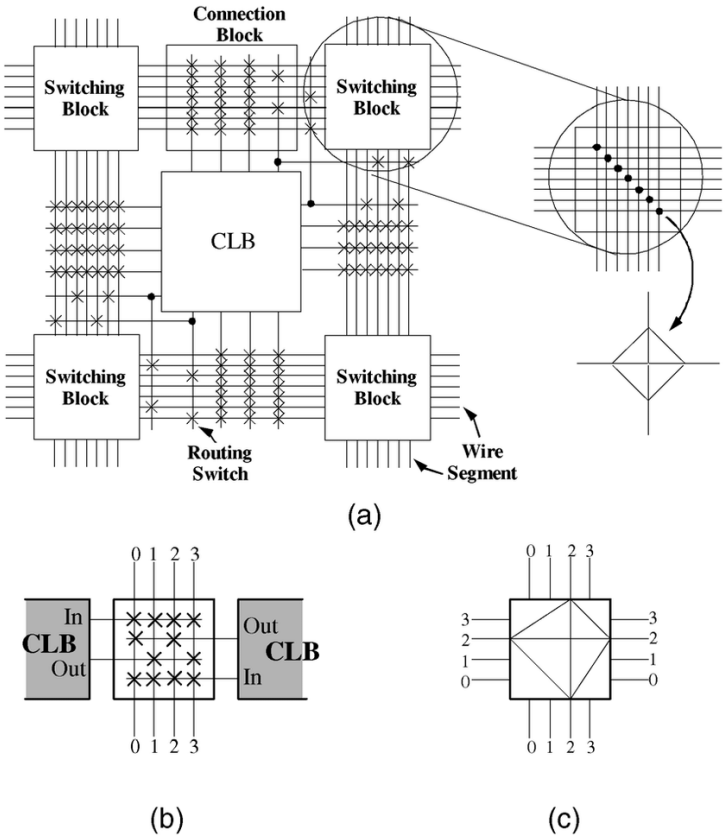

Figure 4: FPGA Routing configuration.[17]

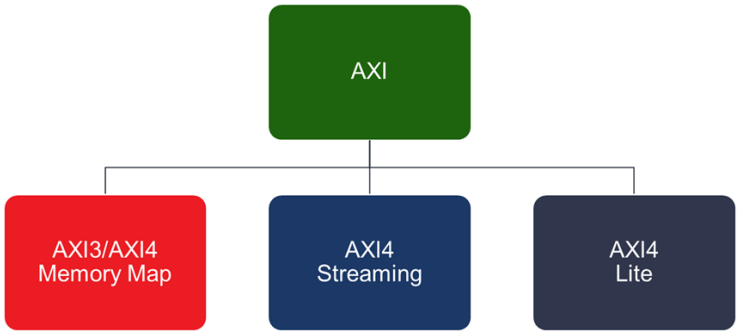

Figure 5: Axi protocol. [18]

Maxi protocol utilizes bursting, the aggregation of successive memory access requests. Also provides the ability to control the max data width, up to 1024 bits. Combining these two techniques it can theoretically achieve 17-19 GB/s for a DDR.[19]

3.3.2 FPGA Components

An FPGA's fundamental building blocks consist of the following components[15]:

- Look-up table (LUT)
- Flip Flops (FF)
- DSP
- Bram and other memories

The Look Up Table (LUT) is the fundamental component of an FPGA and has the ability to implement any logic operation involving N Boolean variables. This component is essentially a truth table where different input/output combinations implement various functions to produce output values. The truth table can only be as big as N, where N is the total number of inputs into the LUT.A LUT can be implemented in hardware by connecting a number of memory cells to a number of multiplexers. The multiplexer selects the outcome at a specific time point using the inputs to the LUT as selector bits. Both a function compute engine and a data storage component can be used with LUT[15].

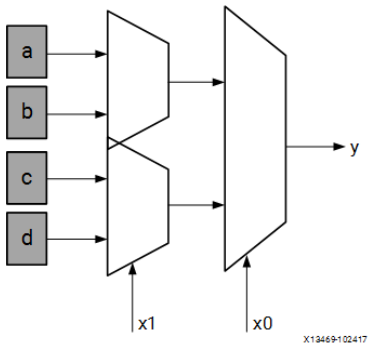

Figure 6: Lut Representation.[15]

A flip-flop's fundamental components are data input, clock input, clock enable, reset, and data output. Any value entered into the data input port during normal operation is latched and sent to the output on each clock pulse. The clock enable pin's function is to enable the flip-flop to hold a particular value for multiple clock pulses. Only when clock and clock enable are equal to one are new data inputs latched and passed to the data output port[15].

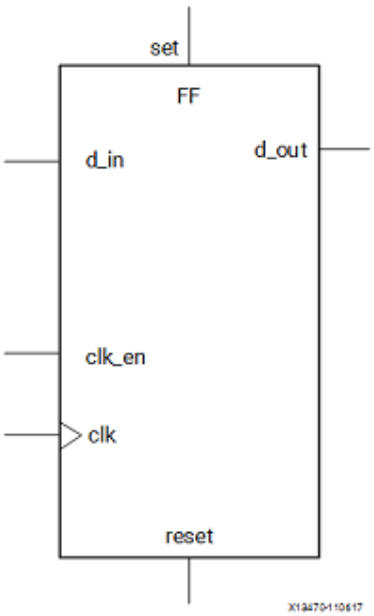

Figure 7: Flip Flop Representation.[15]

The Digital Signal Processing block (DSP), an embedded arithmetic logic unit (ALU) in the FPGA, is made up of a series of three different blocks. An add/subtract unit, a multiplier, and a final add/subtract/accumulate engine make up the DSP's computational chain[15]. A single DSP unit,thanks to this chain , can implement the following functions:

$$P \mathrel{+}= Bx(A+D)$$

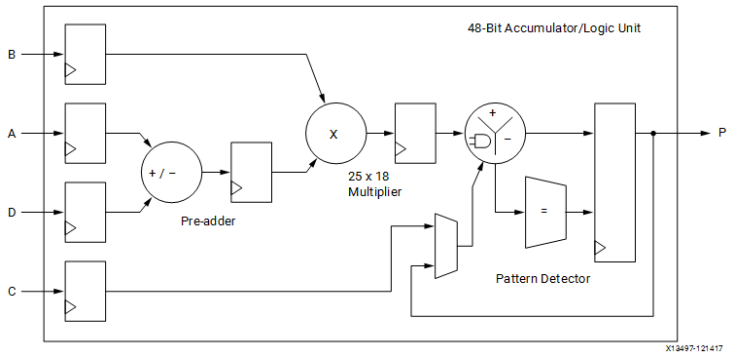

Figure 8: DSP Representation.[15]

Shift registers, read-only memory (ROM), and random-access memory (RAM) are all forms of embedded memory that are present in the FPGA fabric. To provide on-chip storage for a sizable amount of data, the FPGA fabric instantiates the BRAM, a dual-port RAM module. A device's two types of BRAM memory can store either 18k or 36k bits. These memories' dual-port design enables parallel, same-clock-cycle access to numerous locations[15].

In a RAM setup, data can be read and written at any point while the circuit is operating.in contrast, data can only be read from a ROM configuration while the circuit is running. Also for ROM memories, it is possible to implement multiple ports to allow multiple parallel access from different parts of the design, this will become handy later for our model[15].

3.4 High Level Synthesis

The High level Synthensis(HLS) is the process of transforming a C specification into a register transfer level(RTL) to be able to synthesized at a FPGA. It bridges the gap between software and hardware by allowing designers to express their designs using high-level programming languages like C or C++. HLS offers several advantages compared to traditional description languages[20].

Some of them are the following. By abstracting away the low-level details, designers can focus on expressing the desired functionality of their designs. While leaving the implementation details to the HLS tool, which otherwise would consume significant development time. Also, it can validate the functional correctness of the design more quickly, compared to VHDL or Verilog. The numerous

optimization directives available allow control over the synthesis process and still create specific high-performance hardware implementations.With the aid of optimization directives, generate numerous implementations from the C source code. Explore the design space to improve the likehood of finding the best solution. Finally, we can create readable and portable C source code. Retarget the C source into different devices as well as incorporate the C source into new projects[20].

High-level synthesis includes the following phases:

- Scheduling: Determines which operations are carried out during each clock cycle. It is based on the frequency or length of the clock cycle. Moreover, the target device's and the user-specified optimization directives' definitions of the operation's completion time.More operations can be finished in a single clock cycle, and all operations may be finished in a single clock cycle, if the clock period is longer or a faster FPGA is desired. Conversely, high-level synthesis automatically spreads out the operations over more clock cycles if the clock period is longer or a slower FPGA is being used, and some operations might need to be implemented as multicycle resources.

- Binding: identifies the hardware resource that will carry out each scheduled operation. High-level synthesis makes use of data about the target device to implement the best solution.

- Control logic extraction:Extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design.

The Xilinx Vivado HLS tool synthesizes a C function into an IP block to integrate into a hardware system. It is tightly integrated with the rest of the Xilinx design tools and provides comprehensive language support and features for creating the optimal implementation of the C algorithm.

Following is the Vivado HLS design flow stages:

- Compile, execute (simulate), and debug the C algorithm.

- Synthesize the C algorithm into an RTL implementation, optionally using user optimization directives.

- Generate comprehensive reports and analyze the design.

- Verify the RTL implementation using a pushbutton flow.
- Package the RTL implementation into a selection of IP formats.

The flow of using HLS is briefly descripted in 9.

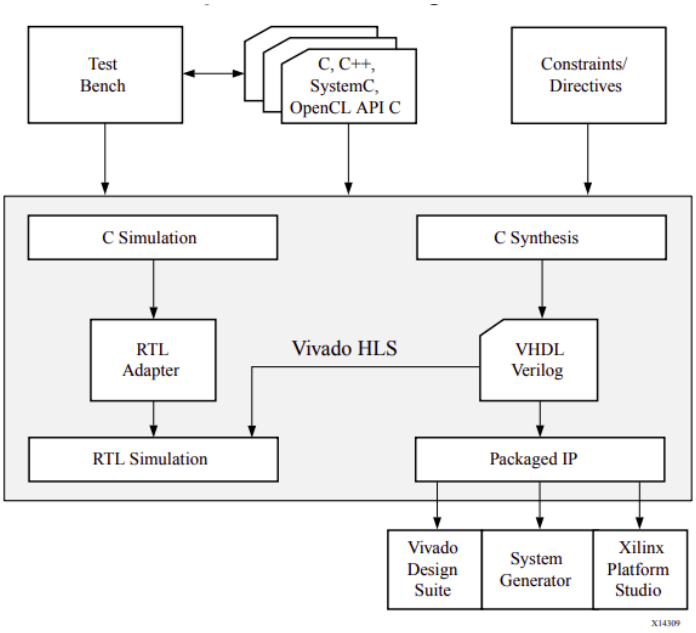

Figure 9: Vivado HLS Design Flow.[20]

A synthesis report is produced by HLS once synthesis is finished. Information on the performance metrics is provided in this report. Understanding the metrics employed to gauge performance in an HLS-created design, is crucial for accomplishing that effectively. Area, latency, and initiation interval are the three main ones.

The synthesis report contains information on the following performance metrics[20]:

- Area: Amount of hardware resources required to implement the design based on the resources available in the FPGA, including look-up tables (LUT), registers, block RAMs, and DSP48s.
- Latency: Number of clock cycles required for the function to compute all output values.

- Initiation interval (II): Number of clock cycles before the function can accept new input data.

3.5 Vitis and Host code

We used the Vtis Unified Platform to accelerate our application. In the Vitis environment, heterogeneous systems include software applications running on x86 host processors or Arm® embedded processors, compute kernels running in programmable-logic (PL) regions or Versal AI Engine arrays, and extensible platform designs that provide the foundation for building and running the heterogeneous systems[21].

The application is mainly separeted in two parts, the host and the fpga. The host serves as the main processing unit that interacts with the FPGA to perform specific tasks.The purpose of the host is to control, coordinate, and communicate with the FPGA to offload computation-intensive tasks or accelerate specific parts of an application. The host CPU communicates with the FPGA to transfer data, configuration parameters, and instructions. It sets up the FPGA for execution, sends data to be processed by FPGA kernels, and receives results back from the FPGA. Also, is responsible for controlling the execution of tasks on the FPGA. It ensures proper synchronization between tasks running on the host and FPGA to maintain the correct sequence of operations. The above are achieved utilizing Opencl Commands and API calls[21].

Another crucial role of the host is the Data Preprocessing and Post-processing. The host may perform data preprocessing before sending it to the FPGA for acceleration. For example, based on the number of features that we decided to train our model, we strip the dataset from those that we ended up not using. This allowed us, to reserve bandwith and storage resources and simultaneously achieve better performance by eliminating unused data. Similarly, after receiving results from the FPGA, the host may perform post-processing to present the final output or integrate it with other parts of the application. The post-processing we did is to determine the quality of the model's prediction by calculating the accuracy, or other metrics such as precision,recall etc.

The development flow of our Vitis application is the following:

- Application compilation using G++

- Kernel compilation using Vitis HLS
- PL kernel linking using Vitis Tools
- Running the Application

Firstly, the host code which is written in C/C++, with Opencl API calls, is compiled using the G++ compiler to generate a executable to run on a x86 processor. Through that, we will communicate with the PL where our kernel will be executed. The function we decide to accelerate is synthesised to a hardware function, which is called kernel. Each C++ kernel when synthesized, using Vitis HLS, produce a Xilinx object (.xo) file. Those Xilinx object (.xo) files are then linked together with the target hardware platform by the Vitis linker to create a device binary file (.xclbin). The command line utility v++, provided by the Vitid AMD, is the one we use for the compiling and linking process. Finally, to run the application the host executable loads the .xclbin file, usually as a command line argument (./host.exe kernel.xclbin). The development flow is illustrated at figure 10.

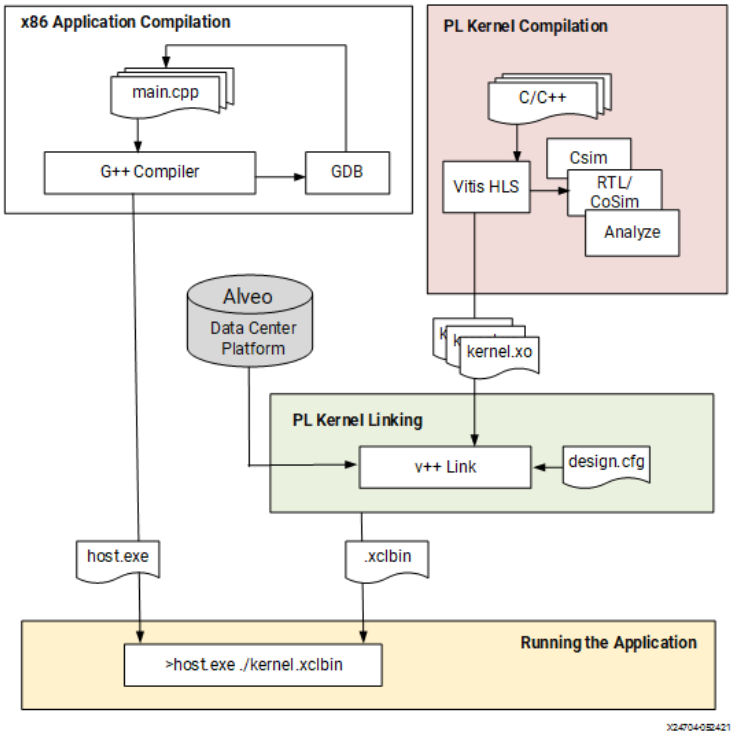

Figure 10: Vitis Development Flow[21].

The steps involving the execution of our application is as follow[21]:

- The host program writes the data needed by a kernel into the global memory of the attached device through the PCIe interface on an Alveo Data center accelerator card.
- The host program sets up the kernel with its input parameters
- The host program triggers the execution of the kernel function on the FPGA.
- The kernel performs the required computation while reading data from global memory, as necessary
- The kernel writes data back to global memory and notifies the host that it has completed its task.
- The host program reads data back from global memory into the host memory and continues processing as needed.

The communication of the CPU and FPGA is shown at figure 11.

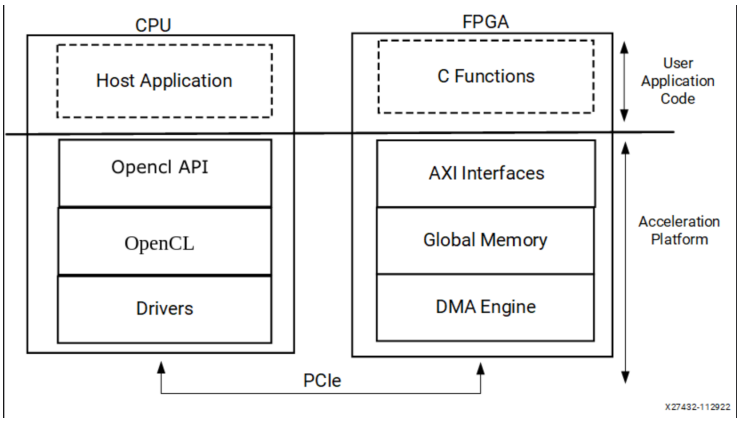

Figure 11: CPU/FPGA Interaction[21].

The nature and contents of the .xclbin file are determined by the build target of the AMD Vitis™ tool we choose during compiling and linking. There are three potential targets, two emulations and the hardware target, each one with it's own functionality. We can specify the potential build target by setting the -t (target) at the v++ command[21]. Those are:

- Software emulation
- Hardware emulation
- Hardware execution

Software emulation (sw_emu) is a crucial step in the FPGA development process using Vitis. Its primary aim is to ensure the correctness of both the host program and FPGA kernels. Sw_emu is valuable for refining algorithms, debugging, and quickly iterating through code improvements without the need for physical FPGA hardware. During software emulation, FPGA kernels are executed natively, and the host program runs on the CPU. It does not consider timing delays or latency and focuses solely on functional execution without providing performance data[21].

Hardware emulation in FPGA development executes an RTL simulation of the programmable logic design, combining PL kernels with

a cycle-approximate hardware platform model. It serves various purposes, including functional correctness verification, testing kernel interactions, and obtaining initial performance estimates. Through cycle-accurate waveforms, we can gain in-depth visibility into kernel activity.Although it has longer compile and execution times than software emulation, it is a valuable tool for validating FPGA designs and obtaining critical insights into the hardware implementation[21].

Finally, once we settle on a design we choose the hardware build target. The v++ builds the FPGA binary for the AMD device by running Vivado synthesis and implementation on the design. It is normal for this build target to take a longer period of time than generating either the software or hardware emulation targets. However, the final FPGA binary can be loaded into the hardware of the accelerator card, or embedded processor platform, and the application can be run in its actual operating environment. Also, we get the actual summary of the resources and timing, on the contrary the hw_emu produce an estimate[21].

4 Random forest implementation for HLS

In this chapter, we describe the methodology followed to generate the initial form of code that can be executed on the FPGA. First, we present the dataset used for training and testing. Then, we describe the process of training the models in Python, as well as the parameters we fine-tuned. Finally, we demonstrate the migration of the models from Python to C++ code for execution on a CPU, and later on an FPGA.

4.1 Model training and tuning

The name of the dataset we used is SWaT A1 & A2 DEC 2015 and was collected from a Secure Water Treatment facility (SWaT). Our aim is to be able to use it to improve the design of a secure Cyber Physical System (CPS). The dataset was generated after 11 days of continuous operation of the facility, with the first 7 under normal operation and the next 4 under attack. The types of attacks are divided into two categories Physical and Network. Our research will focus on Physical attacks, with the goal of developing and accelerating a model for early detection of these types of attacks.

The dataset consists of 3 excel files, the first 2 are related with the normal operation of the installation and the third with the one under attack. In each of these files we have measurements from the 50 sensors of SWaT collected every second, as well as another field where we have the status at the given moment, i.e. Normal or Attack. The pre-processing we did is to compress the 3 files into a single CSV file, which will be the one we will use in training and testing. In addition we removed the Timestamp field, which showed the time of the measurement. The reason for this is that we don't want the generated model to make decisions based on the time it "saw" in the training phase.

At Table 2. we list the name of each sensor and the function it serves. There is an extra column showing the range of values we have for each variable. We notice that most of the values have a small range.Following best practices in ML training, we have normalized the dataset. We used the Scikit-Learn library, specifically the Normalizer class[3]. The normalized data retains the same distribution as the original data, ensuring the utility of your application. We will take advantage of the dense distribution in chapter 6, when

for acceleration and resource saving purposes, we will seek to convert the data from a float_32 representation to some lower precision representation such as integer_16.

Table 2: Features

No	Name	Type	Description	Ranges
1	FIT-101	Sensor	Flow meter; Measures inflow into raw water tank.	0.0-2.760145
2	LIT-101	Sensor	Level Transmitter; Raw water tank level.	120.6237-1000.0
3	MV-101	Actuator	Motorized valve; Controls water flow to the raw water tank.	0-2
4	P-101	Actuator	Pump; Pumps water from raw water tank to second stage.	1-2
5	P-102 (backup)	Actuator	Pump; Pumps water from raw water tank to second stage.	1-2
6	AIT-201	Sensor	Conductivity analyser; Measures NaCl level.	168.0338-272.5263
7	AIT-202	Sensor	pH analyser; Measures HCl level.	6.0-8.988273
8	AIT-203	Sensor	ORP analyser; Measures NaOCl level.	285.3371-567.4699
9	FIT-201	Sensor	Flow Transmitter; Control dosing pumps.	0.0-2.826899
10	MV-201	Actuator	Motorized valve; Controls water flow to the UF feed water tank.	0-2
11	P-201	Actuator	Dosing pump; NaCl dosing pump.	1-2
12	P-202 (backup)	Actuator	Dosing pump; NaCl dosing pump.	1-1
13	P-203	Actuator	Dosing pump; HCl dosing pump.	1-2
14	P-204 (backup)	Actuator	Dosing pump; HCl dosing pump.	1-2
15	P-205	Actuator	Dosing pump; NaOCl dosing pump.	1-2
16	P-206 (backup)	Actuator	Dosing pump; NaOCl dosing pump.	1-2
17	DPIT-301	Sensor	Differential pressure indicating transmitter; Controls the back- wash process.	0.0-45.0
18	FIT-301	Sensor	Flow meter; Measures the flow of water in the UF stage.	0.0-2.376197
19	LIT-301	Sensor	Level Transmitter; UF feed water tank level.	132.8185-1201.0
20	MV-301	Actuator	Motorized Valve; Controls UF-Backwash process.	0-2
21	MV-302	Actuator	MotorizedValve;ControlswaterfromUFprocesstoDe- Chlorination unit.	0-2
22	MV-303	Actuator	Motorized Valve; Controls UF-Backwash drain.	0-2
23	MV-304	Actuator	Motorized Valve; Controls UF drain.	0-2
24	P-301 (backup)	Actuator	UF feed Pump; Pumps water from UF feed water tank to RO feed water tank via UF filtration.	1-2
25	P-302	Actuator	UF feed Pump; Pumps water from UF feed water tank to RO feed water tank via UF filtration.	1-2
26	AIT-401	Sensor	RO hardness meter of water.	0.0-148.8561
27	AIT-402	Sensor	"ORP meter; Controls the NaHSO3dosing(P203), NaOCl dosing (P205)."	140.8357-333.8118
28	FIT-401	Sensor	Flow Transmitter ; Controls the UV dechlorinator.	0.0-1.747862
29	LIT-401	Actuator	Level Transmitter; RO feed water tank level.	130.3896-1003.935
30	P-401 (backup)	Actuator	Pump; Pumps water from RO feed tank to UV dechlorinator.	1-1
31	P-402	Actuator	Pump; Pumps water from RO feed tank to UV dechlorinator.	1-2
32	P-403	Actuator	Sodium bi-sulphate pump.	1-2
33	P-404 (backup)	Actuator	Sodium bi-sulphate pump.	1-1
34	UV-401	Actuator	Dechlorinator; Removes chlorine from water.	1-2
35	AIT-501	Sensor	RO pH analyser; Measures HCl level.	7.303769-8.307037
36	AIT-502	Sensor	RO feed ORP analyser; Measures NaOCl level.	129.8385-272.8531
37	AIT-503	Sensor	RO feed conductivity analyser; Measures NaCl level.	244.8731-297.2635
38	AIT-504	Sensor	RO permeate conductivity analyser; Measures NaCl level.	7.344271-442.4635
39	FIT-501	Sensor	Flow meter; RO membrane inlet flow meter.	0.0-1.757754
40	FIT-502	Sensor	Flow meter; RO Permeate flow meter.	0.0-1.361983
41	FIT-503	Sensor	Flow meter; RO Reject flow meter.	0.0-0.7636911
42	FIT-504	Sensor	Flow meter; RO re-circulation flow meter.	0.0-0.3170099
43	P-501	Actuator	Pump; Pumps dechlorinated water to RO.	1-2
44	P-502 (backup)	Actuator	Pump; Pumps dechlorinated water to RO.	1-1
45	PIT-501	Sensor	Pressure meter; RO feed pressure.	8.891951-264.6437
46	PIT-502	Sensor	Pressure meter; RO permeate pressure.	0.0-3.668343
47	PIT-503	Sensor	Pressure meter;RO reject pressure.	3.108177-200.6376
48	FIT-601	Sensor	Flow meter; UF Backwash flow meter.	0.0-1.80271
49	P-601	Actuator	Pump; Pumps water from RO permeate tank to raw water tank (not used for data collection).	1-1
50	P-602	Actuator	Pump; Pumps water from UF back wash tank to UF filter to clean the membrane.	1-2

The python library Scikit-learn was used for the training part[4]. The algorithm chosen for the classification is the Random forest ensembler. Scikit-learn allowed us to customize the model to the needs of our application.

The main parameters for the training phase are as follows[5]. The n_estimators controls the number of decision trees in the forest, depending on the size we choose the accuracy increases but at the same time the training time increases. During the fitting phase of the model, we set the n_features_in_ parameter to the desired number of features we will use.The criterion refers to the criterion where

it measures the quality of the separation at each node. The function we have chosen is gini, i.e. Gini impurity, other widely used ones are log_loss and entropy, metrics related to Shannon information gain. Max_depth controls maximum height of each tree, we chose not to set a limit since we found that it can drastically limit the accuracy of the predictions. Other parameters worth pointing out are mid_sample_split and min_sample_leaf. These define the minimum number of samples required for a node to be split and can be considered a leaf, respectively. For the same reason I mentioned earlier we left them at their default values, i.e. 1 and 2.

As we mention on section 2.2, rf use the bootstrap technique. To evaluate the robustness of our model to new samples we used the metric Out of Bag Error(OOB).The model takes as inputs the samples it did not "see" during training and shows us the percentage of those it predicted incorrectly. The implementation we finally came up with achieves 0.004 %(OOB).

The two crucial parameters we choose to explore are the number of estimators and the number of features. We emphasized to those two because we could explore them without making heavy assumptions about the dataset and the tree creation process. Even for the same number of features different trees are created, since the feature selection is random. Those trees have different sizes and structures, so setting a limit at the growth results at significant reduction on accuracy. Also, that limit would be inefficient because the tree creation is non deterministic. The split ratio that was chosen, based on general practice, is 70/30. 70% of the dataset was used for the training and other 30% for the testing.

We experimented with a wide range of estimators and number of features. Specifically:

- Estimators: 1, 2, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
- Number of Features: 2, 4, 8, 16, 32, 50

There are total $12(estimatos) * 6(features) = 72$ designs. The metrics that we evaluated were Accuracy, Recall, Precision and F_score.

Accuracy measures the overall correctness of the model's predictions. It is calculated as the ratio of the correctly predicted instances (true positives and true negatives) to the total number of instances in the dataset. A high accuracy indicates that the model is making

correct predictions for most of the data points. For unevenly distributed datasets like ours, if we constantly made the prediction to the majority of labels, we would still get a high accuracy. For example, from a test dataset of 432516 entries there 416080 labeled as Normal and 16436 as Attack. We could achieve a accuracy of 96% by just selecting the Normal label. It is evident that the accuracy metric is not well suited for our application, that why we primarily focus on the other tree.

Since our main goal is to detect attacks, we define as true positive(tp) the correct prediction of an attack. Subsequently, we mark as true negative(tp) the correct prediction of a normal instance. On same principle, the definitions of false positive(fp) and false negative(fn) are derived.

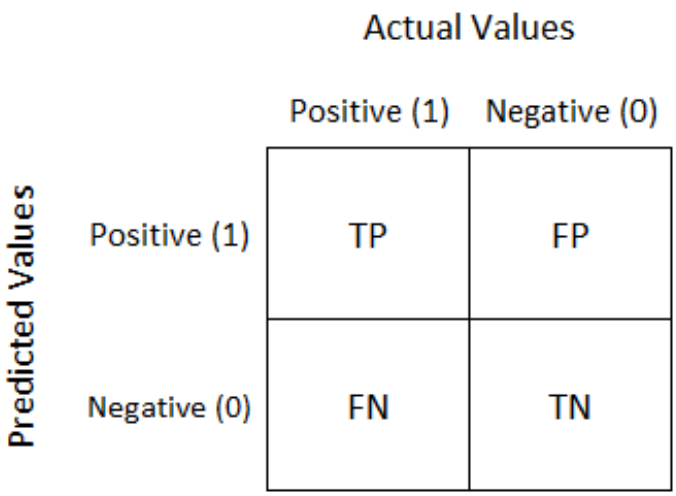

Figure 12: Confusion Matrix[22].

Precision measures the accuracy of positive predictions made by the model. It is intuitively the ability of the classifier not to label a negative sample as positive.It is calculated as the ration of the true positives to the total amount of positives (tp + fp).Recall measures the ability of the model to correctly identify positive instances. It is calculated as the ratio of the true positives to the sum of true positives and false negative(tp+fn).F1_score is a single metric that combines both the precision and recall. It is the metric we will

mostly rely on to evaluate the predictive ability of our model.

Accuracy	$\frac{tp+tn}{tp+tn+fp+fn}$
Precision	$\frac{tp}{tp+fp}$
Recall	$\frac{tp}{tp+fn}$
F1_score	$\frac{2*precision*recall}{precision+recall}$

Table 3: Evaluation metrics.

We trained the models using Python, as mentioned earlier, with various combinations of estimators and features. For each of these models, we recorded the F-score measurements during inference on the test dataset. We visualized these measurements in Figure 13. It is evident that the F-score improves as both the number of estimators and features increase. Beyond 32 features and with more than 2 estimators, all models converge to an F-score greater than 99%. This suggests that the number of features plays a more crucial role than the number of estimators. We exclude models with an F-score $< 99\%$ from our subsequent analysis to retain those with the best predictive ability.. The dashed line in the chart represents that threshold.

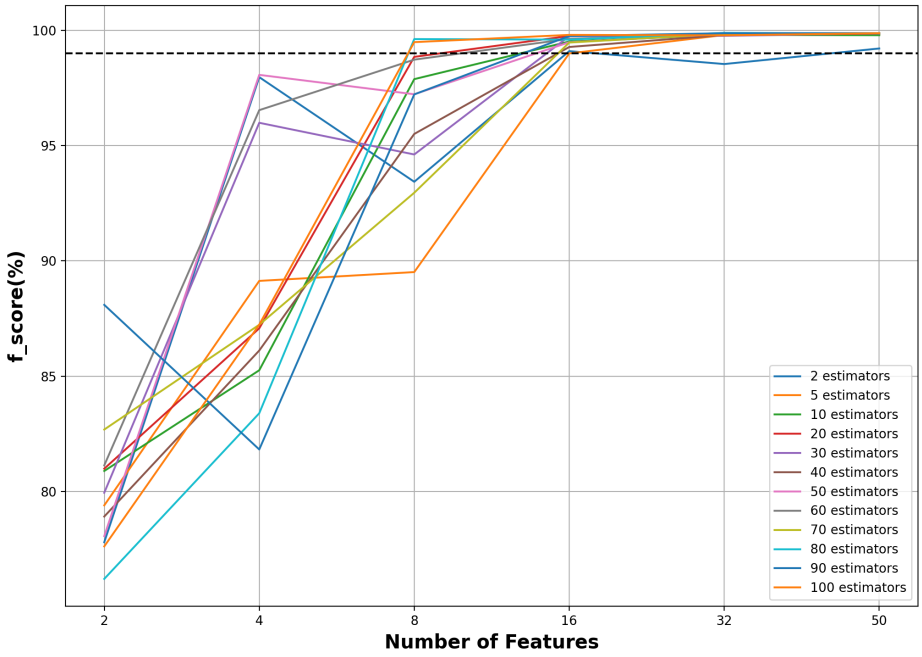

Figure 13: Fscore metrics[22].

4.2 Code transformation for HLS

4.2.1 Python to C++ Model Conversion

As it was mentioned before the training of the forest is done in python, however our goal is to accelerate the inference on FPGA with the HLS C++ tool. So first we need to convert the generated model from python to c++. The model generated by Scikit-learn is saved in a .pkl file using the joblib library. Through this we can access the estimator - decision trees of the forest and store their structure in a c++ suitable format. More specifically, in header file format.

According to Scikit-learn[6], Decision Trees store their structure in arrays. For the reconstruction in C++, we will need five specific arrays: children_left, children_right, feature, threshold, and value.

- The children_left array stores the index of the left child of each

node in the tree. If a node has no left child, it is represented by -1.

- The children_right array is the as the children_left, for the right child.
- The feature array contains the feature index for each node. It represents which feature is used for splitting at each node.
- The threshold array holds the threshold value for each node. It represents the value at which the feature is compared to determine the splitting direction.
- The value array stores the class probabilities for each leaf node. For classification tasks, it represents the distribution of classes in each leaf.

For each tree we know the total number of nodes(node_count) and each one has its own index. The index represents its position in the tables. The root of each tree has index 0 and the rest in the range 1-node_count-1. To illustrate how the aforementioned arrays are used for decision tree traversal, we depicted them in the figure 14. Specifically, we show 3 different decision trees, and for simplicity, we chose to display only the left and right children arrays. In all 3 examples, the node's ID is used as an index in the arrays. Depending on the comparison result at the node, the value is selected from the left or right array. The process stops when we reach a leaf node, meaning both values in the left and right arrays are equal (and equal to -1) for the node's ID.

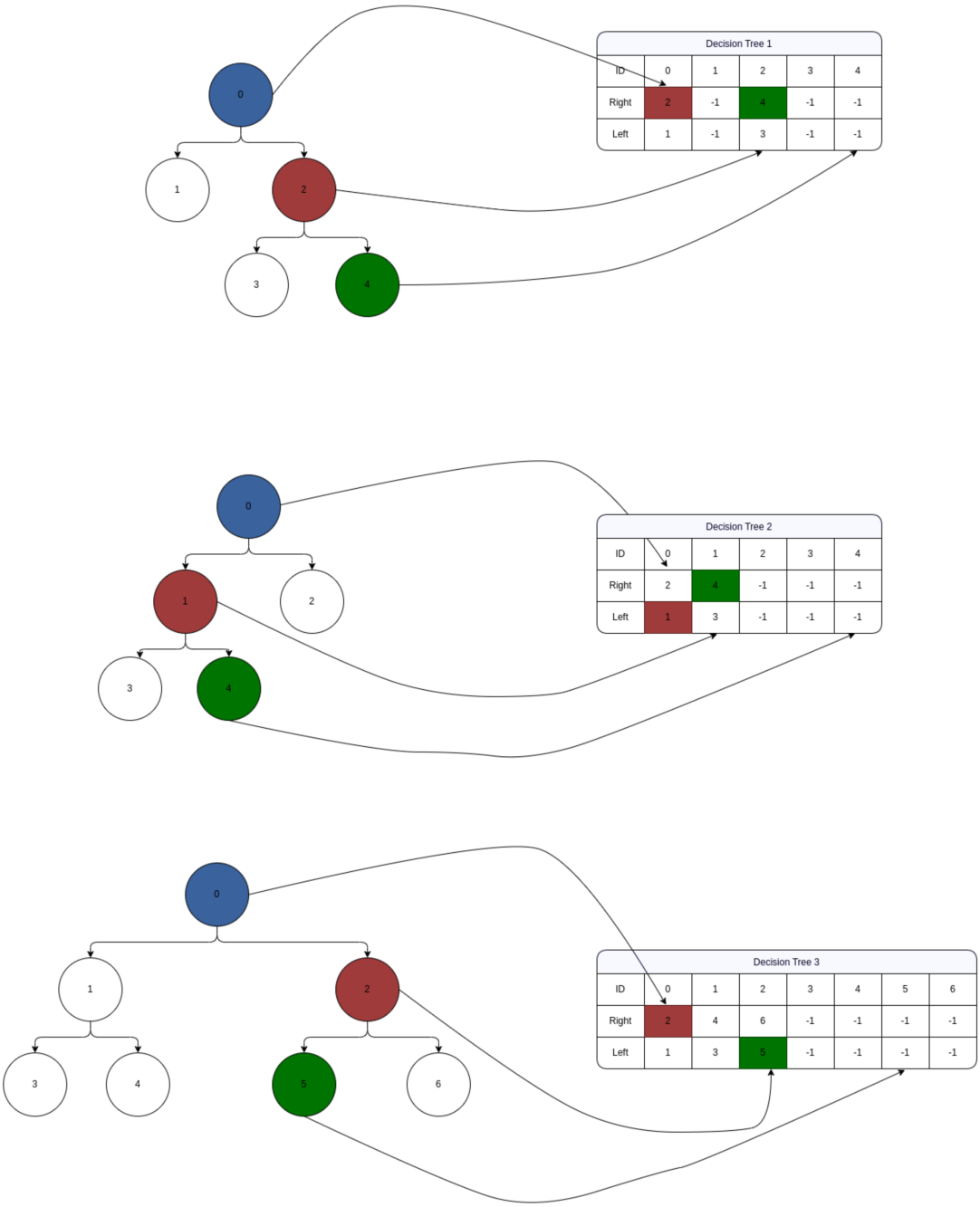

Figure 14: Decision Trees traversal using arrays.

To automate the conversion process a python script was written.

Through this we had access to each tree and its structure. For each of them we started at root and traversed the tree with the DFS algorithm. In the header file trees.hpp the tables are stored in 2D format. .The index of the first dimension represents the respective tree and the second one the value we want to store. The size of each array is equivalent to the product of the number of estimators with the maximum number of nodes from all trees.

$$\text{Array Size} = \text{n\_estimators} \times \text{max\_nodes}$$

The type of children_left , children_right and feature is integer. The threshold is float, because most of our data is float. We chose the value array to be of type bool since our classification is binary. The value 0 is considered as the "Normal" state and the value 1 as the "Attack" state. For example:

$$\text{int children\_left[n\_estimators][max\_nodes]}$$

Having the forest structure stored in the header file, we prepared the first implementation of the code in C++. As we analyzed in the beginning the Random Forest Classifier algorithm is a collection of Binary Decision Trees. At the internal nodes a comparison with a selected feature is made until we reach a leaf. The classification for a specific input is found in the leaf nodes. Finally, a majority voting is performed among all the trees to determine the final classification.

At Listing 1 is the C++ implementation that runs on the CPU:

Listing 1: Random Forest C++

```
#include "trees_packed.hpp"
#include <vector>

void random_forest(const vector<float> &row, bool & prediction){

    const int *left_ptr, *right_ptr, *feature_ptr;
    const float *threshold_ptr;
    const bool *value_ptr;

    int node, class_0, class_1;
    class_1 = 0;

    for(int est=0; est<ESTIMATORS; est++){
        left_ptr = children_left[est];
        right_ptr = children_right[est];
        feature_ptr = feature[est];
        threshold_ptr = threshold[est];
        value_ptr = value[est];
```

```
        node = 0;
        while(left_ptr[node] != right_ptr[node]){
            node = (row[feature_ptr[node]] <= threshold_ptr[node])
                ? left_ptr[node] : right_ptr[node];
        }

        class_1 += value_ptr[node];
    }

    class_0 = ESTIMATORS-class_1;
    prediction = (class_0>=class_1) ? 0 : 1;
}
```

From the above code it is obvious that we reach a leaf only in the case $right\_ptr[node] == left\_ptr[node] == -1$. The path that is followed to traverse each tree is different and unknown beforehand.

4.2.2 C++ to Vitis HLS Modifications

Some changes must be done in order to run this code using HLS on an FPGA. Since the while statement is not supported by the tool, a for loop must be created instead. The expected limit of the for loop is the length of the path until we reach a leaf node. The length of the longest path is thought of as an upper limit because it is difficult to predict which of all potential paths will be taken. The length of the longest is calculated using the DFS algorithm, when we generate the trees.hpp file as we described in the previous section.

In the code, we used vectors because during the subsequent design space exploration, one of the parameters we manipulate is the number of features. For this reason, vectors are convenient for us as the size of each input is variable. However, dynamic allocation is not supported in Vitis, so we replaced the vectors with static allocated arrays.

The aforementioned changes led us to the initial template code that we executed on Vitis and later optimized. At Listing 2 we present the code.

Listing 2: Template For FPGA

```
#include "trees.hpp"
#include "header.hpp"

void random_forest(float (&row)[FEATURE_SIZE],bool & prediction){

    const int *left_ptr, *right_ptr, *feature_ptr;
    const float *threshold_ptr;
    const bool *value_ptr;
```

```
    int node, class_0, class_1;
    class_1 = 0;

    for(int est=0; est<ESTIMATORS; est++){
        left_ptr = children_left[est];
        right_ptr = children_right[est];
        feature_ptr = feature[est];
        threshold_ptr = threshold[est];
        value_ptr = value[est];

        node = 0;
        for(int path=0; path < MAX_PATH; path++){
            if (left_ptr[node] != right_ptr[node]){
                node = (row[feature_ptr[node]] <= threshold_ptr[
                    node]) ? left_ptr[node] : right_ptr[node];
            }
        }

        class_1 += value_ptr[node];
    }

    class_0 = ESTIMATORS-class_1;
    prediction = (class_0>=class_1) ? 0 : 1;
}
```

The reader easily notices that converting the while statement to a for loop introduces overhead. Regardless of how quickly it reaches to a leaf node, we must fully run the loop to ensure the correctness of the algorithm, as if we were taking the longest route. However, we can offset this overhead thanks to the parallelization capabilities offered by the FPGA. We will discuss this in detail at chapter 4.

In the "header.hpp" file, we define the variables that we will modify during the design exploration and need to be known at compile time. An example at Listing 3 is as follows , with 12 ESTIMATORS, 16 FEATURES_SIZE and a maximum path length of MAX_PATH 39.

Listing 3: Example of Header file

```
#ifndef _HEADER_
#define _HEADER_

#define ESTIMATORS 12
#define FEATURE_SIZE 16
#define MAX_PATH 39

#endif
```

5 Architectural optimizations for HLS-based Random forest

5.1 Estimator parallelism within a Random Forest

In Chapter 3.2, we analyzed the Random Forest (RF) algorithm concerning both training and inference. The goal of this thesis is to accelerate the inference phase. To achieve this, we initially identified two main challenges. The first challenge was to determine which parts of the algorithm to accelerate and how to efficiently implement this in code for the Vitis tool. The second challenge involved configuring the code to achieve the optimal resource utilization. We extensively present these challenges as well as proposed solutions in the subchapters.

5.1.1 Challenge 1:Multi-threaded execution on hardware

The first challenge is to transform the sequential execution of the RF algorithm into a parallel one. Initially, we needed to identify which parts of the algorithm could be parallelized. Subsequently, having identified these sections, we proposed changes to the code structure and introduced helper commands in the form of pragmas to achieve the desired parallelism. In the rest of this subchapter, we provide an in-depth analysis of the optimizations we applied to achieve multithreaded execution in hardware.

Decisions are made through majority voting by different and independent estimators. There is no dependency in the prediction made by each decision tree, so there is no need for sequential execution of the decision trees. Instead, we propose performing the inference of decision trees in parallel. More specifically, the input sample will traverse each tree simultaneously. Before proceeding with the implementation of this approach, we modified the generic code from listing 2 to that in listing 4. The reason for this change is to enhance the understanding of the algorithm's operation. Additionally, this restructuring will enable both Vitis and us to apply optimizations more easily.

Listing 4: Updated Code

```
#ifndef _FUNCTIONS_HPP
#define _FUNCTIONS_HPP
#include "trees.hpp"
#include "header.hpp"
```

```
void add(bool * dt_prediction, bool & prediction){
    int count=0;
    loop_predictions:for(int est=0; est < ESTIMATORS; est++){
        count += dt_prediction[est];
    }
    prediction = (ESTIMATORS-count >= count)? 0 : 1;
}

void decision_tree(const float row[FEATURE_SIZE], bool & prediction
    , int est_index){
    #pragma hls inline off

    int node_index=0;

    for(int path=0; path<MAX_PATH; path++){
        if(children_left[est_index][node_index] != children_right[
            est_index][node_index]){
            float threshold_value = threshold[est_index][node_index
                ];
            float row_value = row[feature[est_index][node_index]];

            node_index = ( row_value <= threshold_value) ?
                children_left[est_index][node_index] :
                children_right[est_index][node_index];
        }
    }
    prediction = value[est_index][node_index];
}
void random_forest(const float row[FEATURE_SIZE], bool & prediction
    ){
    bool dt_prediction[ESTIMATORS];
        for(int est=0;est<ESTIMATORS;est++){
            dt(row,dt_prediction[est],est);
        }
    add(dt_prediction,prediction);
}

#endif
```

The function ‘decision_tree‘ has replaced the body of the initial loop. It takes the sample and the ‘prediction‘ variable as arguments, where it will store its own prediction. The ‘est_index‘ argument indicates which estimator the function is called for, allowing the correct arrays to be read. The ‘add‘ function takes predictions from all estimators in the form of an array and returns the classification after majority voting.

The ‘random_forest‘ function, as before, is responsible for returning the classification based on the sample. It does so by calling the ‘decision_tree‘ function for each estimator and then using the ‘add‘ function.

Vitis provides us with the ability to define the architecture of our model at a high level. One way to achieve this is through pragmas, where the user can add them to their code. During hardware implementation, each function can be built into an instance that performs the function's operation. In sequential execution, as described up to listing 2, one instance of the 'decision_tree' function is built. Thus, the estimators use the same instance and must wait for previous execution to complete before starting their own.

In FPGAs, parallel execution of a function is accomplished by building multiple instances/copies. To achieve parallel execution of the estimators, multiple instances of the 'decision_tree' function need to be built. Each instance will correspond to an estimator.Each instance will correspond to an estimator. The HLS pragma we will use to achieve this is the unroll. The unroll pragma transforms loops by creating multiple copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel[19]. Through the "factor" parameter, we can select the degree to which we perform unrolling. Since our goal is the parallel execution of all estimators, we have not specified the "factor," opting for a complete unroll of the loop.

In order for the Vitis tool to understand that we want parallel execution of the estimators, we needed to apply additional pragmas. Specifically, we added the dataflow pragma. The dataflow pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design. The dataflow optimization enables the operations in a function or loop to start operation before the previous function or loop completes all its operations, provided that there are no data dependencies [19].

Given that we want parallel execution, we will also need parallel access to shared data, such as the arrays representing the tree structures and the sample input. Since we will have only read-only access, one approach is to use the array_partition pragma. The pragma array_partition partitions an array into smaller arrays or individual elements, resulting in RTL (Register-Transfer Level) with multiple small memories or multiple registers instead of one large memory. This effectively increases the amount of read and write ports for the storage[19].Due to the size of the arrays for each tree, which contain more than 1024 elements, and the random traversal

of these arrays, it is not possible to partition them. The random access occurs because in each iteration, the next node is recalculated based on the comparison result between the elements of the current node and the input. As a result, these arrays are stored as BRAM in a read-only format because, apart from reading the elements, no other processing is performed on them.

Since no partitioning of the arrays is perfomed in the subsequent iterations of the code, we replaced the array_partition pragma with bind_storage. The bind_storage pragma assigns a variable (array or function argument) in the code to a specific memory type (type) in the RTL. The HLS tool implements the memory using the specified implementations (impl) in the hardware[19]. As the type, we define ROM_NP (multiple port ROM memory), and we choose bram as the impl.

The above changes, as shown in listing 5, transform the sequential execution into parallel, with a significant impact on the latency cycles of a sigle sample inference. In the table 4, it is evident that we managed to reduce latency from 2135 to 339, achieving a speedup of 6.3. As expected, there is a trade-off between performance and resources utilization. The most significant increase was observed was in the utilization of BRAMs, which went from approximately 67(1%) to 400(9%). These measurements pertain to the hw_emu environment, and as mentioned in Chapter 3, they can only approximate the real ones from the hw implementation. The goal is to capture the operation and impact of the above changes, as well as those we will present later. For this purpose, the information from the synthesis report is sufficient.

	Cycles	Bram(%)	Lut(%)	FF(%)
Sequential	2139	1.55	0.52	0.16
Parallel	339	9.25	0.87	0.43

Table 4: Sequential vs Parallel.

Listing 5: Parallel estimators

```
void random_forest(const float row[FEATURE_SIZE], bool & prediction
    ){
```

```
    #pragma hls dataflow

    #pragma HLS array_partition variable=children_left type=
        complete
    #pragma HLS array_partition variable=children_right type=
        complete
    #pragma HLS array_partition variable=threshold type=complete
    #pragma HLS array_partition variable=feature type=complete
    #pragma HLS array_partition variable=value type=complete


    bool dt_prediction[ESTIMATORS];
    #pragma HLS ARRAY_PARTITION variable=dt_prediction type=
        complete
    #pragma HLS ARRAY_PARTITION variable=row type=complete

    for(int est=0;est<ESTIMATORS;est++){
        #pragma hls unroll
            dt(row,dt_prediction[est],est);
    }

    add(dt_prediction,prediction);
}
```

5.1.2 Challenge 2: Efficient BlockRAM utilization.

In the previous subsection, we demonstrated that we can achieve parallel execution of decision trees within a random forest (RF) by instantiating the same function multiple times and applying the necessary pragmas, such as unroll and dataflow. The second challenge we had to address was to maintain the above parallelism while significantly reducing the required resources. The main problem stemmed from the way we stored the decision tree arrays and how we accessed them. To address this, we needed to move away from the generic code that allows iterative access/execution and focus on solutions with hardcoded functions for each decision tree that significantly reduce resource utilization.

Further experimentation showed that we could achieve greater resource savings, specifically in the number of Block RAM (BRAM) required to represent each estimator. Up until then, we stored the RF arrays in a 2D representation. As mentioned in the chapter 4.2, each decision tree in an RF requires five arrays to operate, which are left_children, right_children, threshold, feature, and value. We decided to aggregate the arrays from all decision trees that perform the same utility (e.g., left_children) and store them in a data structure as an array of arrays. The primary reason for doing this was

to have iterative access to the sub-arrays of a given decision tree through a loop by simply changing the estimator index of the first dimension, as shown in the listing 5.

We observed the following behavior: Vitis, the tool we used, did not understand which subset of the RF arrays to load into the instances it was building when the estimator index was given through a loop. As a result, it loaded all the RF arrays and then selected which ones to use. While this achieved parallelism, it led to resource underutilization. The solution to this problem was to make the tool understand which subset of arrays it intended to use before hand and load only that subset. To do this, we had to manually unroll the loop. First, we attempted to give the estimator index as an input to the decision tree function, either through the loop or hardcoded, as shown in the listing6. However, this did not help reduce the number of BRAM. From the above, we concluded that regardless of whether the decision tree function received the estimator index as an argument (through the loop or hardcoded), it would load all the arrays and then choose which ones to use.

Listing 6: Manual Unroll using the generic form of the decisio tree function.

```
void random_forest(const float row[FEATURE_SIZE], bool & prediction
    ){
    #pragma hls dataflow

    #pragma HLS bind_storage variable=children_left_0 type=ROM_NP
    #pragma HLS bind_storage variable=children_right_0 type=ROM_NP
        impl=bram
    #pragma HLS bind_storage variable=threshold_0 type=ROM_NP impl=
        bram
    #pragma HLS bind_storage variable=feature_0 type=ROM_NP impl=
        bram
    #pragma HLS bind_storage variable=value_0 type=ROM_NP impl=bram

                          .
                          .
                          .

    bool dt_prediction[ESTIMATORS];
    #pragma HLS ARRAY_PARTITION variable=dt_prediction type=
        complete
    #pragma HLS ARRAY_PARTITION variable=row type=complete


        dt(row,dt_prediction[0],0);
        dt(row,dt_prediction[1],1);
        dt(row,dt_prediction[2],2);
        dt(row,dt_prediction[3],3);
        dt(row,dt_prediction[4],4);
        dt(row,dt_prediction[5],5);
```

```
        dt(row,dt_prediction[6],6);
        dt(row,dt_prediction[7],7);
        dt(row,dt_prediction[8],8);
        dt(row,dt_prediction[9],9);

    add(dt_prediction,prediction);
}
#endif
```

We needed to change how we stored the RF arrays. Now, the arrays are stored independently as 1D arrays, meaning they are not aggregated and stored in 2D format based on their utility. Each array's name is a combination of its utility(e.g., left_children) and the decision tree it belongs to. For example, the enumeration 5 shows how we defined the arrays for decision tree 1. During array declaration, we can now use the actual size (e.g., 1547 elements) and not that of the largest array/tree (decision tree 8: 1739 elements), as we were forced to do with the array of arrays data structure. This allowed us to adjust the number of BRAM for each array and not take the worst-case scenario for all of them.

1. int children_left_1[1547]
2. int children_right_1[1547]
3. float threshold_1[1547]
4. int feature_1[1547]
5. int value_1[1547]

Having the arrays in the above format, we wrote different hardcoded functions for each decision tree, as shown in the listing 7. The main difference from the generic form of the decision tree function, as presented in the listing5, is that the subset of arrays used is hardcoded and not determined by the estimator index argument, which is no longer needed. Essentially, we transitioned from iterative parallel execution of one generic function using pragma unroll to dataflow parallel execution of specific hardcoded functions.

Listing 7: Hardcoded Random Forest

```
#ifndef _FUNCTIONS_HPP
#define _FUNCTIONS_HPP
#include "trees_fpga.hpp"
#include "header.hpp"
#include <iostream>
```

```
using namespace std;
void print_row(const float row[FEATURE_SIZE]){
    for(int i=0; i<FEATURE_SIZE; i++){
        std::cout<<(float)row[i]<<" ";
    }
    std::cout<<endl;
}

void add(bool * dt_prediction, bool & prediction){
    int count=0;
    loop_predictions:for(int est=0; est < ESTIMATORS; est++){
        #pragma hls unroll
        count += dt_prediction[est];
    }
    prediction = (ESTIMATORS-count >= count)? 0 : 1;
}

void dt_0(const float row[FEATURE_SIZE], bool & prediction){
    #pragma hls inline off
    // int est_index=0;

    int node_index=0;

    loop_dt_0:for(int path=0; path<depth[0]; path++){
        if(children_left_0[node_index] != children_right_0[
            node_index]){
            float threshold_value = threshold_0[node_index];
            float row_value = row[feature_0[node_index]];

            node_index = ( row_value <= threshold_value) ?
                children_left_0[node_index] : children_right_0[
                node_index];
        }
    }
    prediction = value_0[node_index];
}

                              .
                              .
                              .

void rf(const float row[FEATURE_SIZE], bool & prediction){
    #pragma hls dataflow

    #pragma HLS bind_storage variable=children_left_0 type=ROM_NP
    #pragma HLS bind_storage variable=children_right_0 type=ROM_NP
        impl=bram
    #pragma HLS bind_storage variable=threshold_0 type=ROM_NP impl=
        bram
    #pragma HLS bind_storage variable=feature_0 type=ROM_NP impl=
        bram
    #pragma HLS bind_storage variable=value_0 type=ROM_NP impl=bram

                          .
                          .
                          .
```

```
    bool dt_prediction[ESTIMATORS];
    #pragma HLS ARRAY_PARTITION variable=dt_prediction type=
        complete
    #pragma HLS ARRAY_PARTITION variable=row type=complete


        dt_0(row, dt_prediction[0]);
        dt_1(row, dt_prediction[1]);
        dt_2(row, dt_prediction[2]);
        dt_3(row, dt_prediction[3]);
        dt_4(row, dt_prediction[4]);
        dt_5(row, dt_prediction[5]);
        dt_6(row, dt_prediction[6]);
        dt_7(row, dt_prediction[7]);
        dt_8(row, dt_prediction[8]);
        dt_9(row, dt_prediction[9]);

    add(dt_prediction, prediction);
}
#endif
```

As shown in the figure 15, from the 400(9%) BRAMs required for the unroll implementation, we managed to reduce it to 102(2%). In the figure 16, we are comparing single-sample inference for the implementations, with the best on being the hard coded unroll. We achieved 6.5 speedup compared to sequential implementation with only increasing the BRAM utilization by 52%.

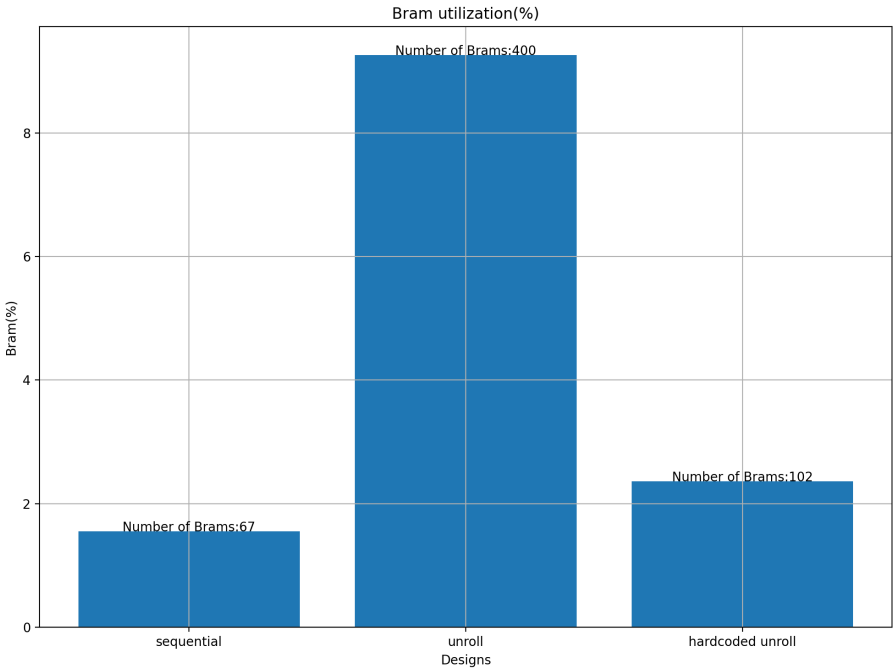

Figure 15: Bram utilization.

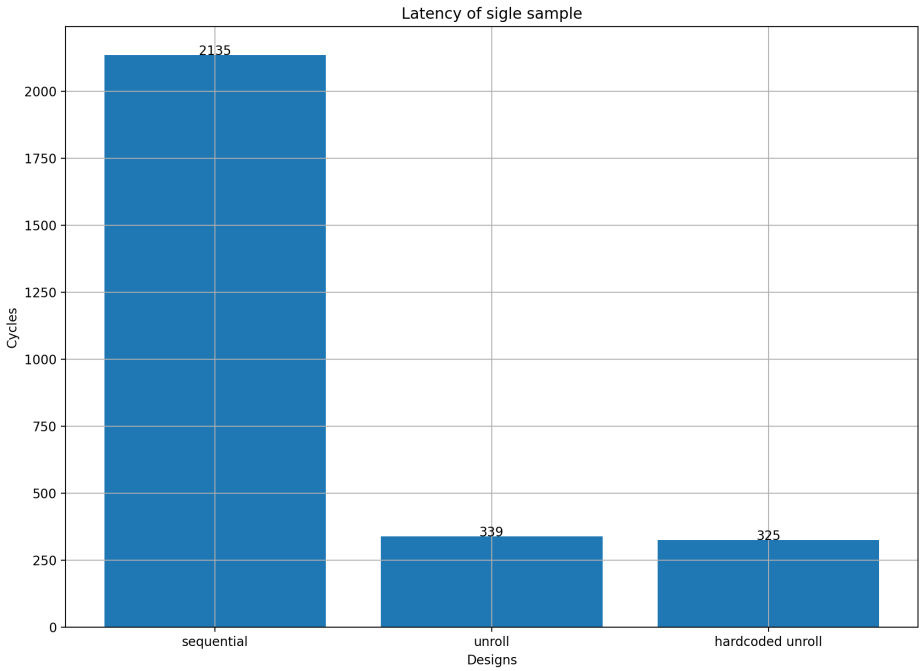

Figure 16: Latency of an single sample.

In Figure 17, screenshots captured from the Vitis HLS tool display a comparison of how many BRAMs are required for the representation of arrays for each instance. Specifically, for the hardcoded implementation, 10 BRAMs are needed, while for the unroll version, 36 BRAMs are required, achieving a 72% savings. It is evident that this significant reduction becomes even more pronounced as the number of estimators increases.

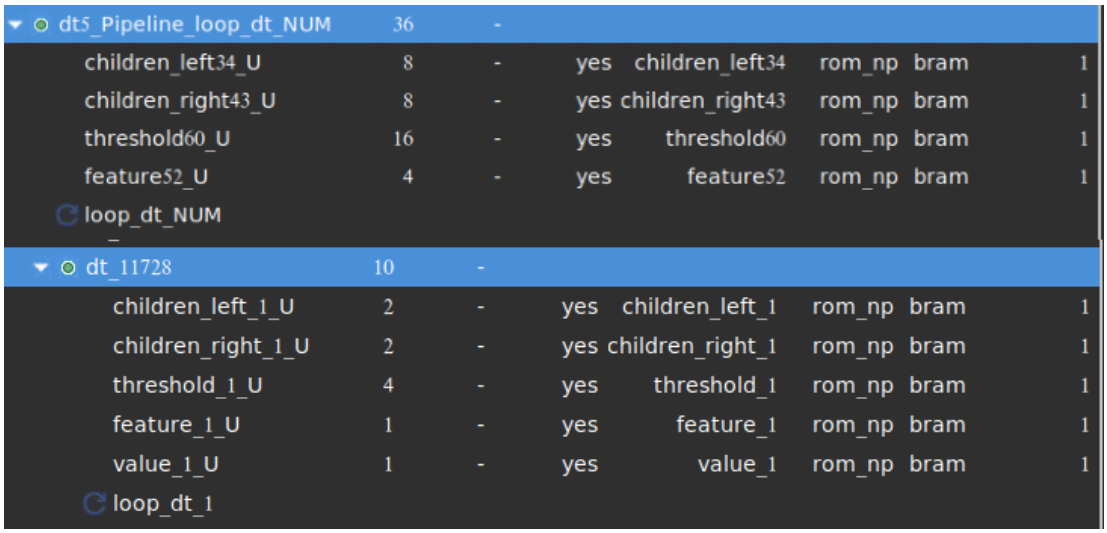

Figure 17: Unroll vs Hardcoded resources.

The same structure,as presented at 5,is followed for the arrays of the remaining decision trees as well.We customized the original Python script used to generate the header file for the arrays, as mentioned in 4.2. The new file, named trees_fpga.hpp, contains individual arrays for each decision tree. We kept the original format for inference on the host, renaming the file to trees_host.hpp, where we perform validation by comparing the predictions with those from the FPGA inference.

5.2 Parallelism across multiple random forests

In the 5.1 subsection, we analyzed the transformation from sequential execution of the estimators to parallel execution. However, we can achieve parallelism in other parts of the RF algorithm as well. Each sample input is independent of the others. Given this, we explored two different approaches of the kernel's architecture.The first approach involves parallel execution of different copies of the random forest instance, where each one will perform inference on a different input sample. The second proposal is a pipeline approach with the goal of achieving an Iteration Interval (II) equal to 1. This translates to processing a new input or producing a prediction in every clock cycle, after the initial latency.

Before we proceed to analyze the two different architectures, we

should examine the common functions that we will parameterize. The "krnl_rf" function is the top-level function that accelerates on the FPGA and communicates with the host. It takes three arguments: "input," which corresponds to the samples, "output," where we return the classifications, and "size," indicating the number of samples. The first two are implemented as MAXI ports by Vitis.

The body of krnl_rf consists of a loop over the entire input, calling the predict function. The predict function takes the same arguments as krnl_rf. Internally, predict performs three main operations. First, it reads the input argument into a local register ,named samples, using the read_input function. Then, it performs classification using the parallel_rf and pipeline_rf functions, each for one of the two different architectures respectively . The generated classifications are temporarily stored in the rf_predictions register. Finally, the contents of rf_predictions are transferred to the AXI output using the write_output function. These three functions have different code and implementations in hardware for the two architectures but serve the same purpose.

5.2.1 Coarse Grained Design

By creating multiple, independent instances of the Random Forest function, we enable the simultaneous and parallel processing of inputs. An illustration of the proposed architecture is shown in the figure 18. Specifically, we represent K parallel RFs, internally each one has N parallel estimators. This architecture introduces an extra layer of parallelism beyond that of the parallel estimators, as it can simultaneously process K inputs.The number of RFs (K), which moving on we will refer to as RF_SIZE, is a hyperparameter that we explored to understand its impact on both inference time and resource utilization. Since this parameter(like the ESTIMATORS or FEATURE_SIZE) must be known during synthesis, we included it in the header.hpp file, as shown in listing 8.

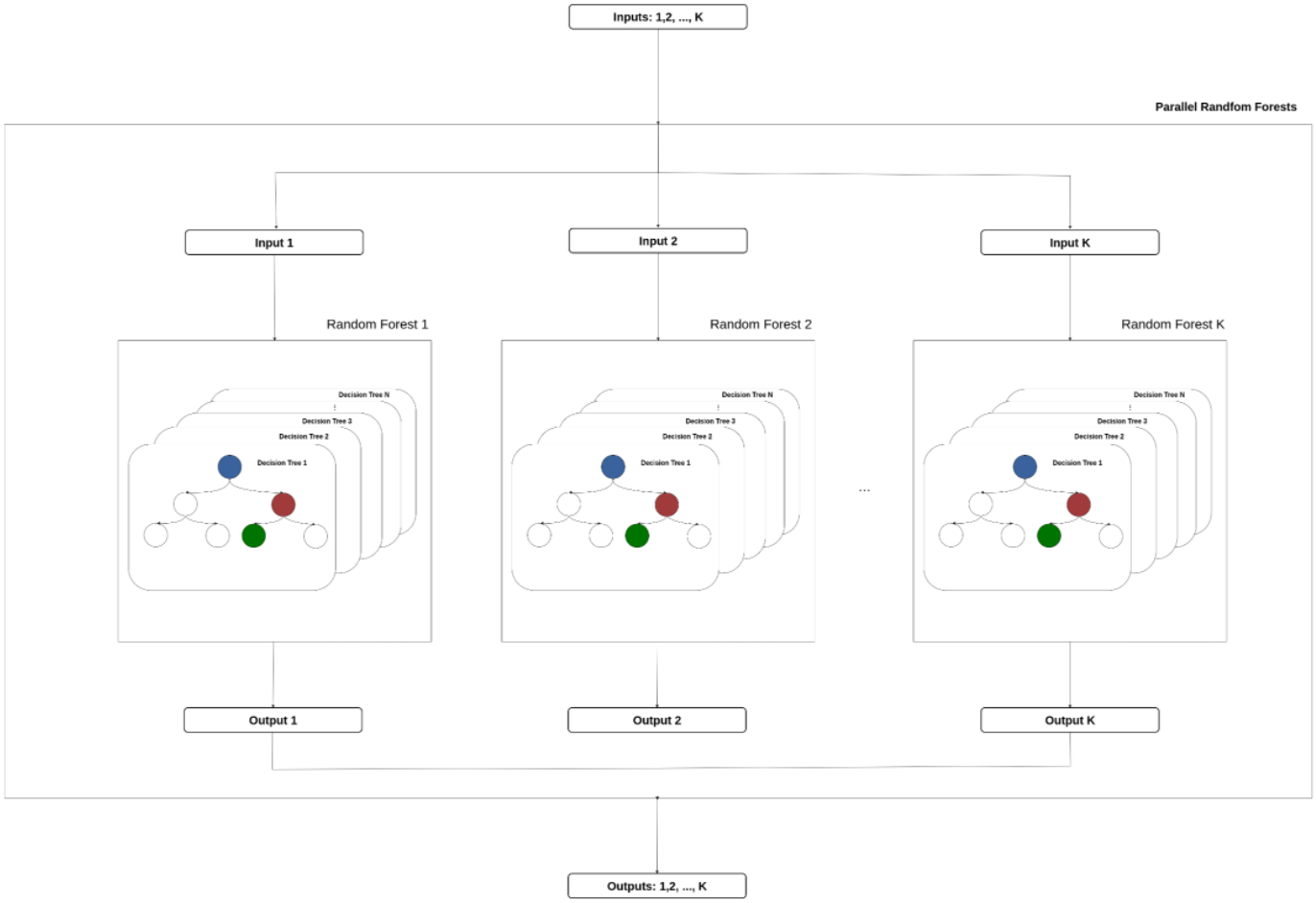

Figure 18: K parallel RFs with N parallel estimators for simultaneous processing of K inputs.

Listing 8: Header file with RF_SIZE

```
#ifndef _HEADER_
#define _HEADER_

#define RF_SIZE 8
#define ESTIMATORS 10
#define FEATURE_SIZE 16


#endif
```

We followed the methodology similar to the estimators to achieve

parallelism among the copies of the random forests. Specifically, we created a new function called parallel_rf, as shown at listing 9, in which we hard-coded the calls to the random forest. It takes two arguments, samples a 2D array with dimensions RF_SIZE*FEATURE _SIZE, and rf_predictions. Same as before, we used the pragma dataflow to ensure parallel overlap execution of the rf instances.

Listing 9: parallel_rf RF_SIZE=8

```
void parallel_rf(const float samples[RF_SIZE][FEATURE_SIZE], bool *
    rf_predictions){

        rf(samples[0],rf_predictions[0]);
        rf(samples[1],rf_predictions[1]);
        rf(samples[2],rf_predictions[2]);
        rf(samples[3],rf_predictions[3]);
        rf(samples[4],rf_predictions[4]);
        rf(samples[5],rf_predictions[5]);
        rf(samples[6],rf_predictions[6]);
        rf(samples[7],rf_predictions[7]);

}
```

The listing 10 shows the implementations of the read_input, write _output, and predict functions. To achieve the minimum possible delay during input reading, we set the MAXI port's bus width to the maximum value, which is 1024. We implemented it using the configuration command config_interface -m_axi_max_widen_bitwidth 1024. Additionally, to assist the tool to instrument this functionality, we defined a synthetic type for the input port, indicating that we aim to perform parallel reads of FEATURE_SIZE elements.

$$typedef hls :: vector < float, FEATURE\_SIZE > v\_row$$

Reading from the input MAXI is done in segments of size RF_SIZE, so each call of read_input corresponds to reading total RF_SIZE * FEATURE_SIZE size of data from the MAXI. The index argument is used to determine the starting position for reading from MAXI and is updated with each call of read_input to point to the correct location. The write_output function operates at a similar fashion. In both functions, we used the pragma unroll in order to achieve parallel reads and writes. In the predict function, we define and partition the registers samples and rf_predictions and also call the previously mentioned functions accordingly.

Listing 10: Function definitions

```
void read_input( const v_row * input, float (&rows)[RF_SIZE][
    FEATURE_SIZE], int index){
    loop_rf_row_input: for(int rf=0; rf<RF_SIZE; rf++){
        #pragma HLS unroll
        loop_feature:for(int j=0; j<FEATURE_SIZE; j++){
            #pragma HLS unroll
            rows[rf][j] = input[index+rf][j];
        }
    }
}

void write_output( bool * rf_predictions , bool * output, int index
    ){
    loop_rf_row_outpus: for(int rf=0; rf<RF_SIZE; rf++){
        #pragma HLS unroll
        output[index+rf]=rf_predictions[rf];
    }
}

void predict(const v_row * input,bool * output, int index){
    #pragma hls dataflow

    float rows[RF_SIZE][FEATURE_SIZE];
    bool rf_predictions[RF_SIZE];

    #pragma HLS ARRAY_PARTITION variable=rows type=complete
    #pragma HLS ARRAY_PARTITION variable=rf_predictions type=
        complete

    read_input(input,rows,index);
    multiple_rf(rows,rf_predictions);
    write_output(rf_predictions,output,index);
}
```

The top kernel function krnl_rf is shown in the listing 11. The bounds of the loop are dynamically determined by the size argument. However, the step at which we iterate over the input is fixed and equal to the number of parallel RFs. As a result, regardless of the input size, inference will be performed for at least RF_SIZE times. In cases where the "size" is not an integer multiple of RF_SIZE, the kernel will read whatever is available on the MAXI BUS and proceed to classify it as if it were valid input. Consequently, this can lead to underutilization of the kernel, which is why it's important to find the appropriate RF_SIZE.

Listing 11: Top kernel function

```
#include "functions.hpp"

using namespace std;

extern "C" {
```

```
void krnl_rf(const v_row *input, bool *output, int size){
    loop_rf:for(int index=0; index<size; index+=RF_SIZE){
        #pragma hls dataflow
        predict(input, output, index);
        }
    }
}
```

In order to have reliable measurements for both inference time and resource utilization, we performed a hardware implementation. As expected, creating copies of RFs achieves parallelism but utilizes more resources. The resources on each FPGA, including BRAM, FF, and LUT, are fixed, so we needed to determine how many copies we can accommodate. We started with the hardware implementation of a single RF, which serves as our baseline. We defined a threshold for resource utilization that we consider acceptable and do not want to exceed to ensure the correct operation of the FPGA. This threshold is set at 70% of the previously mentioned resources. We conducted a binary search to find the maximum RF_SIZE that would not exceed 70% utilization. During the exploration, we tried RF_SIZE values of 1, 2, 4, 8, 16, and 24. With RF_SIZE=24 being the largest model with utilization as shown in the table 5.

RF_SIZE	Bram(%)	Lut(%)	FF(%)
1	20.21	15.91	10.86
2	22.52	16.42	11.16
4	27.15	17.49	11.71
8	36.41	19.53	12.79
16	52.18	22.73	14.06
24	70.69	26.84	16.14

Table 5: Resources Utilization for HW implementation.

We observe that the resource that changes the most is BRAM. In order to have parallel access to the same array there is need for multiple ports. However there is a limited number of ports (a total

of 2) for each BRAM. In our case, we need all trees to operate in parallel, therefore more than 2 ports are required. A potential solution would be to partition the arrays so that tree nodes that are accessed at the same time are placed in different arrays and therefore BRAMs. However, there is no specific pattern for which position in the array we are trying to read, as we have random access that depends on the input. Therefore, the only solution the tool employs is copying them. Given this, we should expect a linear increase in resources with respect to RF_SIZE, which is indeed the case. We observe that for each additional RF, there is an average overhead of 2.2% in BRAM, 0.5% in LUT, and 0.25% in FF (Flip-Flops). We will present charts related to inference time after analyzing the pipeline architecture, so that we could draw more conclusions by comparing the two architectures.

5.2.2 Pipeline architecture

Each estimator can be represented as a sequence of S stages, where S is equal to the longest path in each tree, as shown at figure 19.

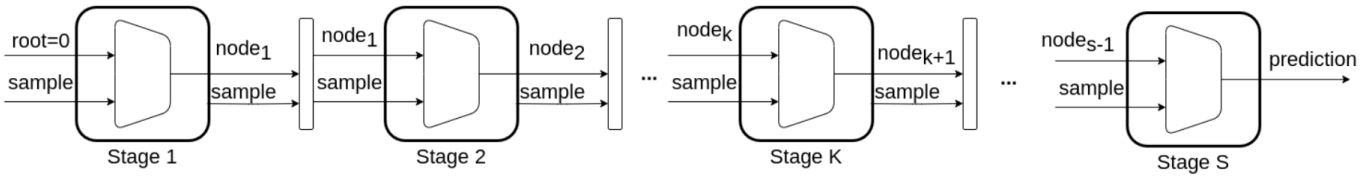

Figure 19: Stages of a DT.

Let K represent a stage with $0 < K <= S$. It takes two inputs: the node index of the previous stage $node_k$, and the sample for which we are making an inference. As output, it produces the next node_id ,$node_{k+1}$, which results from comparing the sample with the estimator's arrays. A visual representation of a stage is shown at figure 20. The final stage, S, generates the estimator's prediction. The goal is to transform the sequential execution of the S stages into a pipeline. This can be achieved by placing buffers between the stages. As a result, it is possible to save and shift the node_id and the sample through the pipeline stages.

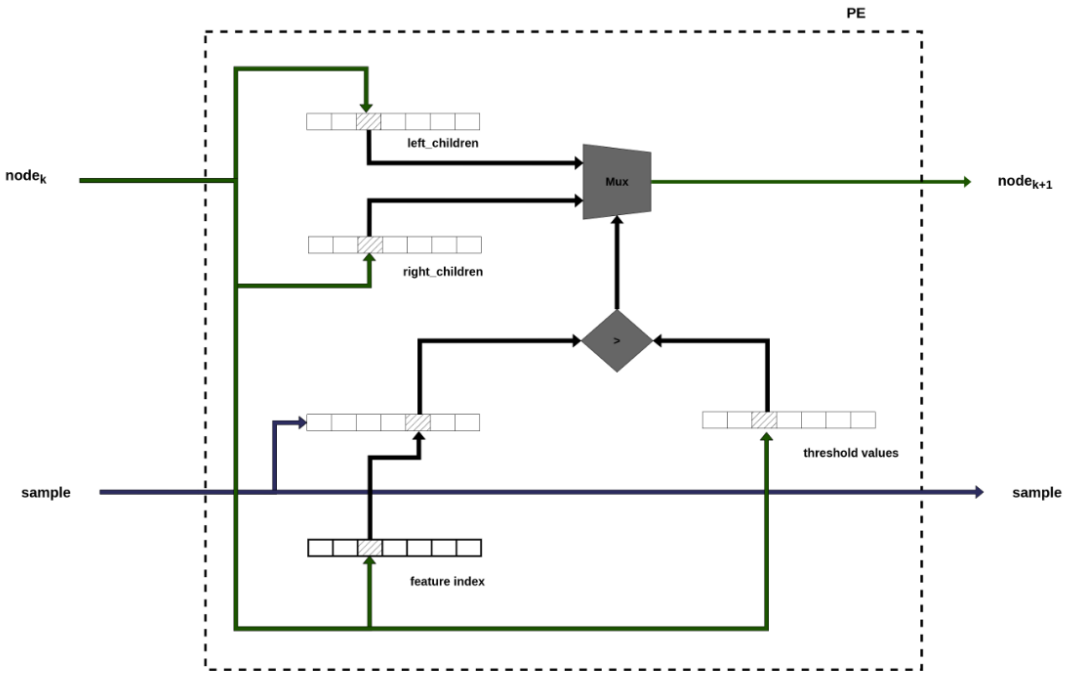

Figure 20: Representation of a stage.

In each cycle, it is fed with a new sample, and after the initial latency (S) an II=1 is achieved. A conceptual representation of the propagation of the multiple inputs through the dt pipeline is shown at figure 21.

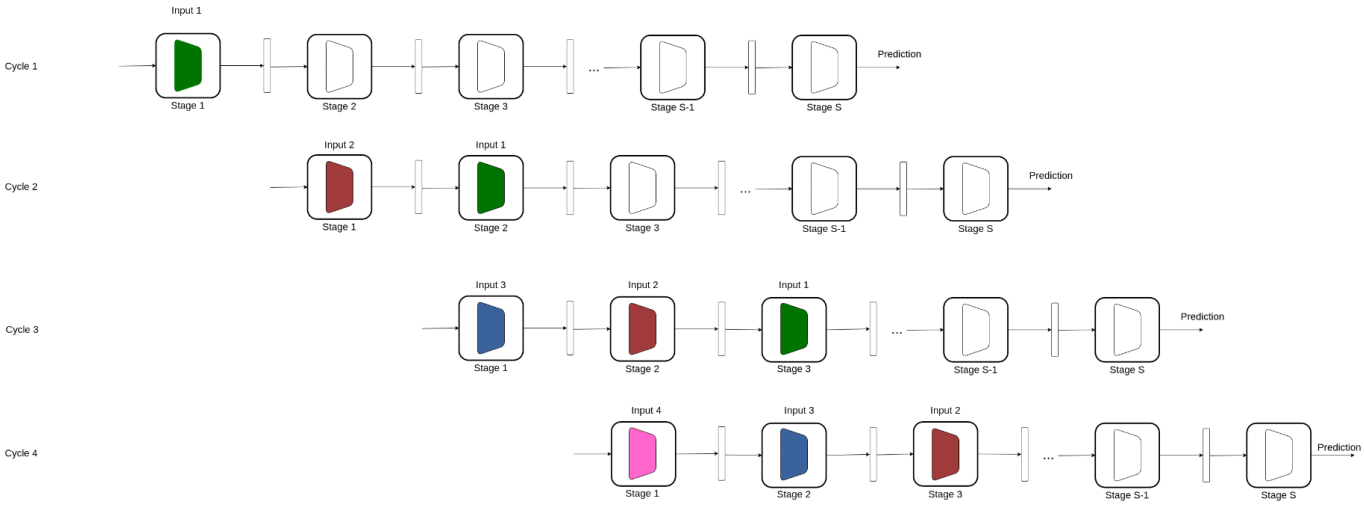

Figure 21: Propagation of input data through the pipeline design.

We have shown that a decision tree can be represented as a pipeline. We extend this concept to the random forest (RF), which consists of a collection of independently functioning decision trees. Therefore, the proposed architecture comprises internally independent parallel pipelines, which are combined to achieve at high-level a RF that is also a pipeline. An illustration of this architecture is shown at the diagram 22.

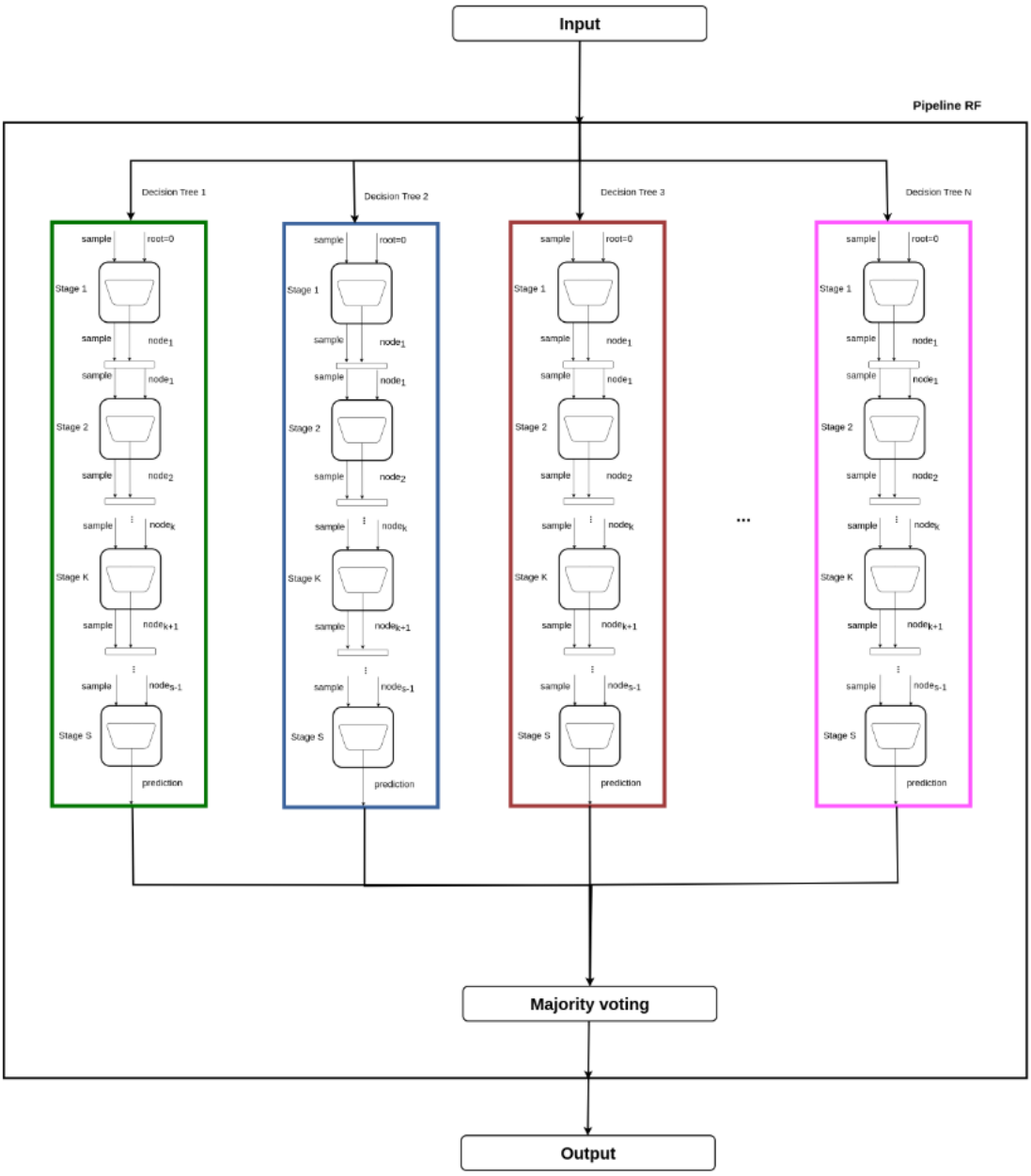

Figure 22: Representation of RF pipeline.

The changes we made to the code to generate a pipeline architecture are relatively minor compared to those for the parallel architecture. The main change is that we completely removed the

pragma dataflow and placed the pragma pipeline in the appropriate places. Specifically, we removed the pragma dataflow from the krnl_rf,predict and rf functions and renamed the later to pipeline_rf. Then, we modified the read_input and write_output functions to read/write only one sample at a time. In the predict function, the 'samples' array is defined as a 1D array with a size of FEATURE_SIZE, and rf_predictions is now a scalar. We kept the v_row type for the same benefits described at the 4.2 subsection. The aforementioned changed are displayed at listing 12 .

Listing 12: Functions definition for pipeline arch

```
void read_input( const v_row * input, float (&data)[FEATURE_SIZE],
    int i){
    loop_feature:for(int j=0; j<FEATURE_SIZE; j++){
        #pragma HLS pipeline
        data[j] = input[i][j];
    }
}

void write_output( bool &result, bool * output, int index){
    output[index]=result;
}

void predict(const v_row * input,bool * output, int index){
    float samples[FEATURE_SIZE];
    bool rf_predictions;

    #pragma HLS ARRAY_PARTITION variable=sample type=complete

    read_input(input,sample,index);
    pipeline_rf(sample,prediction);
    write_output(prediction,output,index);
}
```

Finally, we added the pragma pipeline to the krnl_rf function,as shown at listing 13, to force the tool to pipeline the inner loop. The index is increment by 1(not by RF_SIZE) at each iteration. A consequence of applying the pipeline is that all loops at lower levels are automatically unrolled, such as in the read_input and dt functions, so the use of pragma unroll is now unnecessary.

Listing 13: Top kernel fuction for pipeline arch

```
#include "functions.hpp"
using namespace std;

extern "C" {

void krnl_rf(const v_row *input, bool *output, int size){
        loop_samples:for(int i=0; i<size; i++){
            #pragma hls pipeline
```

```
                pipe(input,output,i);
                }
        }
}
```

The iteration latency of the RF is equal to that of the longest decision tree plus the delay from the other parts of the architecture.However, after this period, an II=1 is achieved.The theoretical formula for calculating the required cycles for inference of a dataset of size D for the parallel and pipeline architecture is shown at table 6.

Architecture	Total Cycles
Pipeline	$Iteration\_Latency_{pipeline} + D - 1$
Parallel	$\lceil \frac{D}{RF\_SIZE} \rceil * Iteration\_Latency_{parallel}$

Table 6: Required Cycles for Dataset of size D.

Similarly to the parallel architecture, we proceeded with the hardware implementation of the pipeline architecture. The measurements for resource utilization are shown at table 7.In the shared diagrams 23 24 25, we compare the resource utilization for the two architectures.

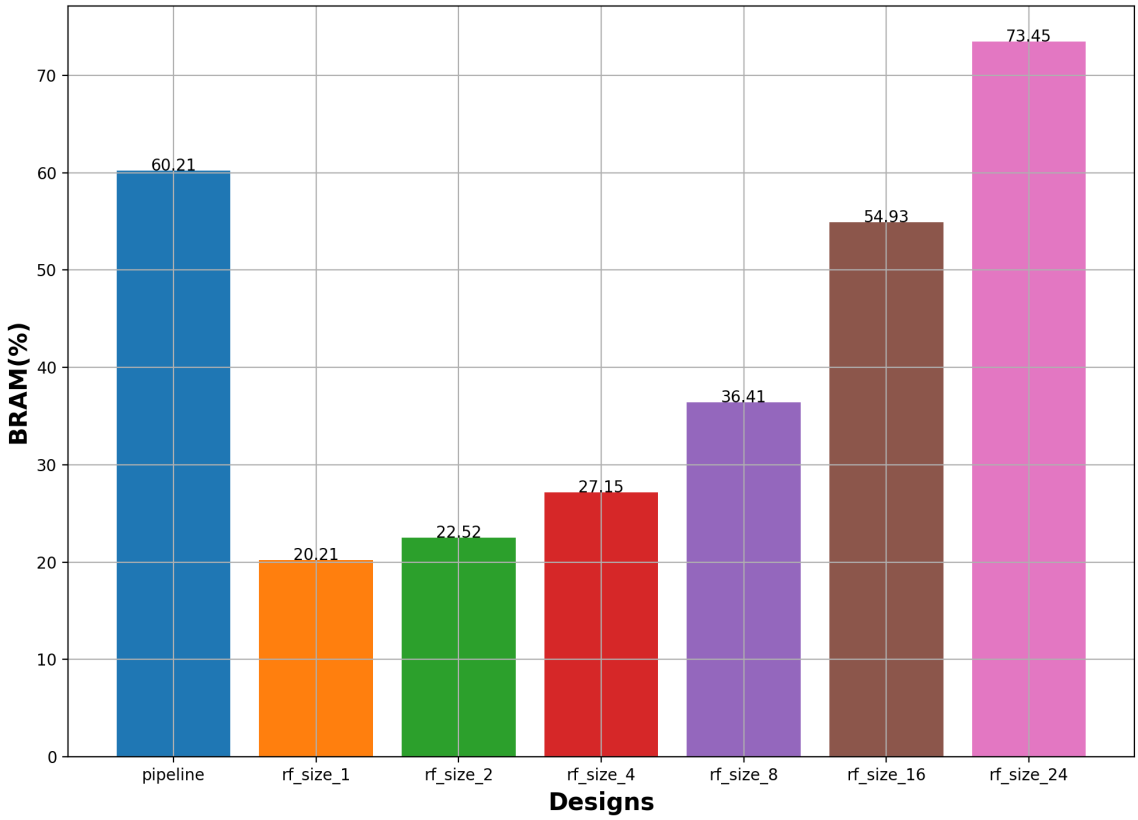

Figure 23: Bram Utilization.

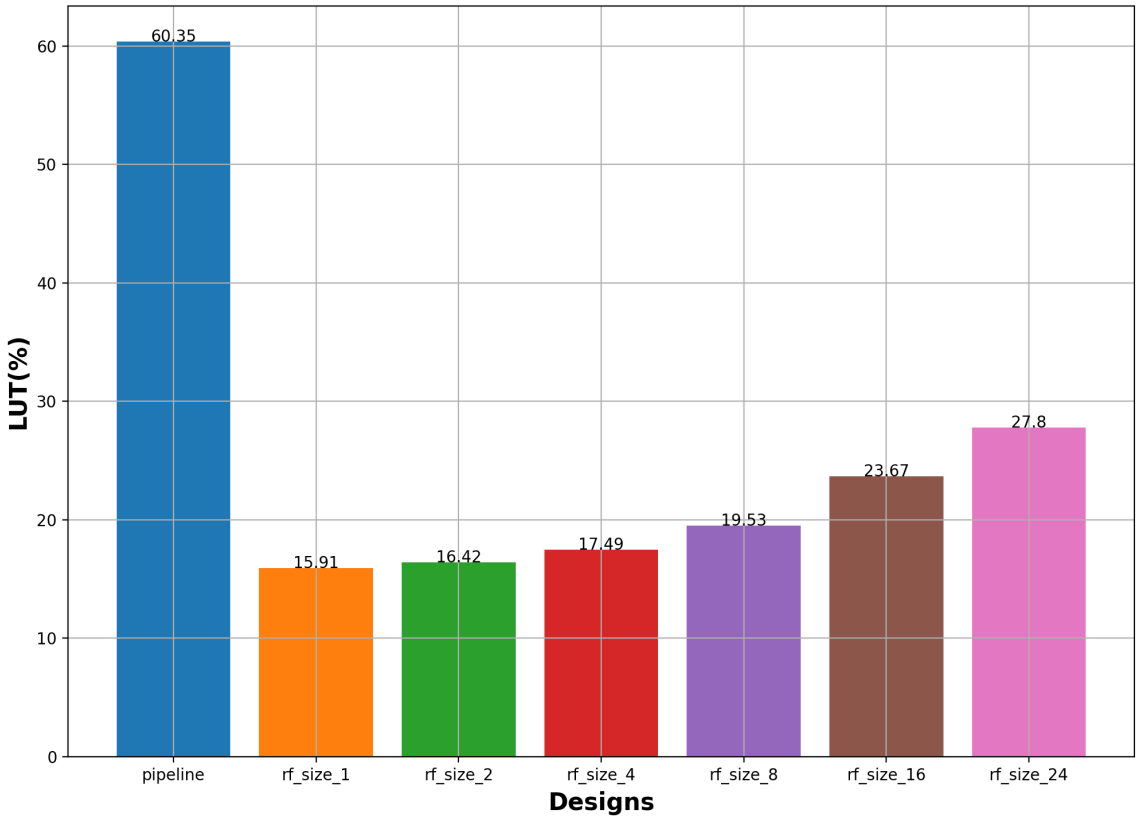

Figure 24: LUT Utilization.

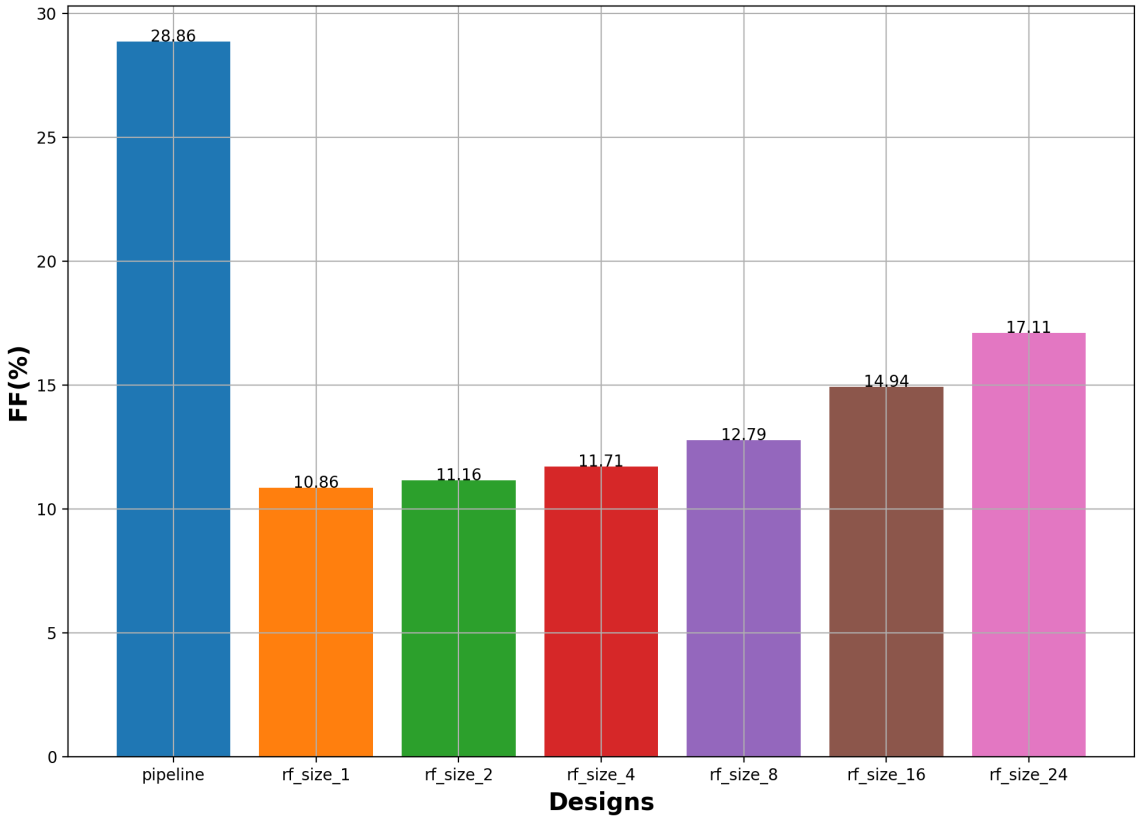

Figure 25: FF Utilization.

Architecture	Bram(%)	Lut(%)	FF(%)
Pipeline	60.21	60.35	28.86

Table 7: Pipeline Resources Utilization.

In the figure 26, we compare the inference time for all the designs for a common input size. The input sizes we evaluated are 48, 480, 4800, 48000, and 288344 (the entire dataset). To fully utilize

the parallel RFs, we selected the minimum input size as the least common multiple of the RF sizes, which is lcm(1, 2, 4, 8, 16, 24) = 48. Except for the entire dataset case, the other sizes are multiples of the lcm.

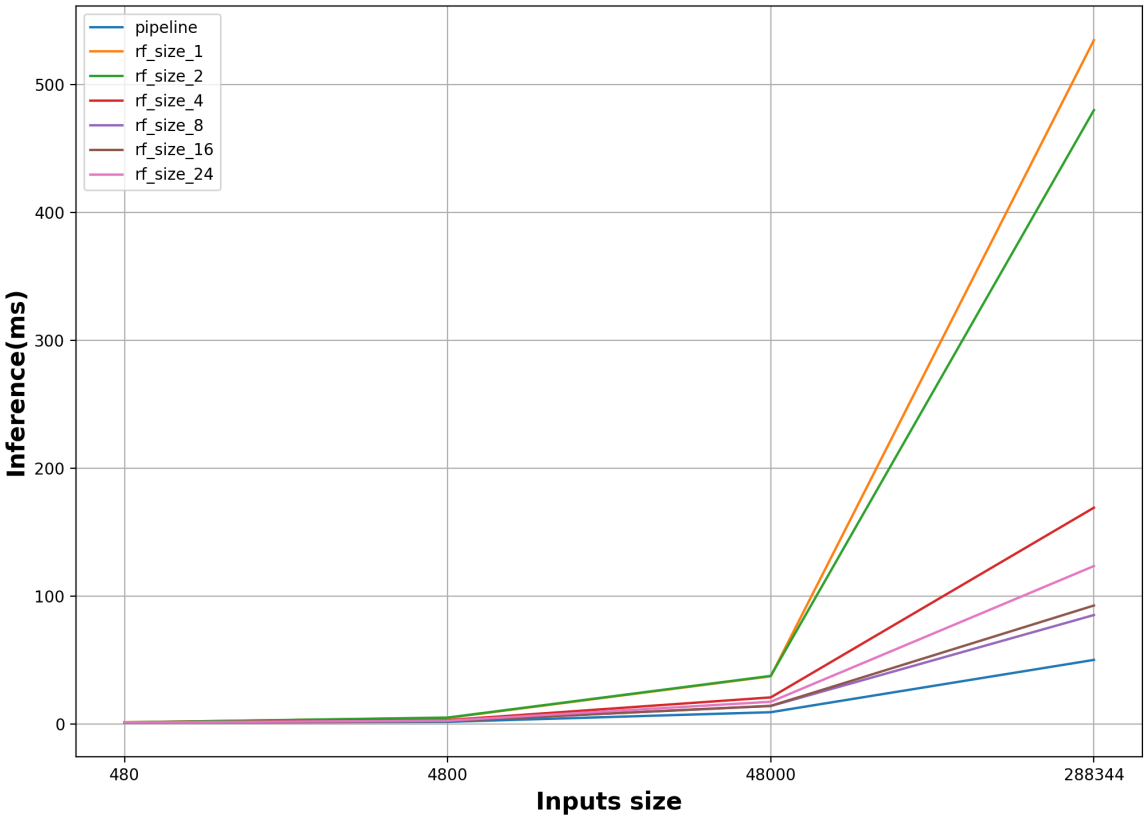

Figure 26: Inference times.

6 Precision Scaling Exploration

From the diagram in the previous chapter, we observed that the most dominant resource, meaning the one that changes the most and significantly influences whether the design can be implemented, is the BRAM. Therefore, the goal is to reduce the percentage of BRAM usage while keeping performance stable. We achieved this through the technique of Precision Scaling. More specifically, we reduced the number of bits required to represent the elements in the decision tree's arrays.

The tool already performs such optimizations, which is where we got inspired. At the figure 17 we showed the number of BRAM(18K bits) required for the representation of arrays. If we focus on the hardcoded implementation, we can observe the aforementioned optimization. All arrays in Decision Tree 1 have the same size (1701) and the same type integer, except for the threshold, which has a float type. The range for the values of the arrays is ther following:

- children_left, children_right: (-1,1700)
- feature: (-1,49)
- value : 0 or 1

The number of bits required to represent a integer n is $log_2 n + 1$. So the elements of children arrays demand 12 bits for representation, for feature 7, and for value 1. it is evident that there is no need the full 32 bits of a standard int. The number of 18k bits BRAM used for each array is as follows:

- children_left, children_right:$\lceil \frac{1701*12}{18000} \rceil = \lceil 1.134 \rceil = 2$ BRAM
- feature: $\lceil \frac{1701*7}{18000} \rceil = \lceil 0.661 \rceil = 1$ BRAM
- value: $\lceil \frac{1701*1}{18000} \rceil = \lceil 0.094 \rceil = 1$ BRAM

Those calculations align with the results from the figure 17. We concluded that the tool automatically converts the standard int type, with which we defined the arrays, into smaller int types. It's not explicitly mentioned which specific type it ends up with (in the degree we were able to check), but it performs similarly to the ap_uint types that we manually defined to investigate this hypothesis.

The same does not happen for the threshold array, which has a float type. Due to the structure in which a float number is stored, automatic optimizations like the one mentioned above are not possible. A breakdown of how single-precision floating-point numbers are typically structured is as following[23]:

- Sign bit (1 bit): This bit represents the sign of the number. 0 indicates a positive number, and 1 indicates a negative number.
- Exponent bits (8 bits): These bits represent the exponent of the number. They are biased, meaning that a certain value is subtracted from them to allow for both positive and negative exponents. In single-precision, the bias is 127, so the actual exponent is calculated as (exponent bits - 127).
- Fraction bits (23 bits): These bits represent the significand or mantissa of the number. They are used to store the digits of the number in binary form.

The formula to calculate the value of a single-precision floating-point number is :

$$(-1)^{sign} * 2^{(exponent-bias)} * 1.fraction$$

It's evident that to represent a number in floating-point format, 32 bits are required. Regardless of whether the number is simple, meaning it doesn't have many decimal digits, it's not possible to use fewer bits for the exponent or mantissa since the count is hardcoded in the hardware, a ilustration is shown at figure 27. As a counter-measure to this logic, we suggest using fixed-point and even integer representations, after appling the appropriate transformations to the weights of your model.

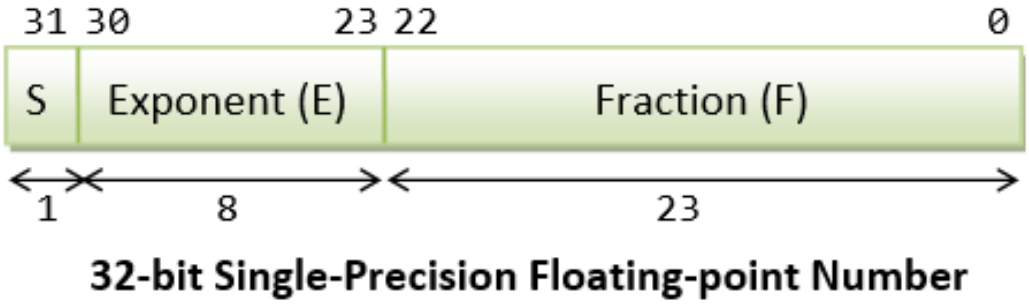

Figure 27: Floating Point Representation[23].

6.1 Fixed Point representation

The Vitis tool provides the capability to use arbitrary precision data types for both floating-point and integer representations. For fixed-point representation, we can customize to a specific number the required bits for the integer and fractional parts of a number. Using the ¡ap_fixed.h¿ library, we can define fixed-point types with the command $ap\_fixed < W, I, Q, O, N >$[19]. Additionally, there is an option for unsigned fixed-point types by instead applying the ap_ufixed prefix, which we will choose since all the features are greater than 0.

- W denotes the word length in bits.
- I specifies the number of bits used to represent the integer value.
- Q and O determine the quantization and overflow mode.
- N defines the number of saturation bits in overflow wrap modes.

For our application, we will only tune the W, I, and Q parameters, leaving the other two flags at their default settings.

As mentioned in the training subsection 4.1, the features are normalized between (0,1). Therefore, there is no integer part, and we can focus solely on the fractional part. Given this, the fixed-point type will allocate all the bits, as specified by us, exclusively for the fractional part. The exploration we will conduct concerns the number of these bits. Specifically, we will examine 8, 16, 24, and 32 bits. We used the quantization flag AP_RND, as it gave us the best accuracy through our testing. An example for 16-bit unsigned

fixed-point format is $ap_u fixed < 16, 0, AP\_RND >$. For simplicity and also to ensure a common structure and terminology in all the generated codes through the exploration, we renamed each ap_ufixed type to float_fixed using the typedef command. The specific type is included in each header.hpp file of the models we wish to explore, as shown at listing 14.

Listing 14: Header for fix point type

```
#ifndef _HEADER_
#define _HEADER_
#include "ap_fixed.h"

#define DATATYPE_SIZE 16

#define ESTIMATORS 10
#define FEATURE_SIZE 32

typedef ap_ufixed<DATATYPE_SIZE,0,AP_RND> float_fixed;

#endif
```

To apply precision scaling with fixed-point numbers, we only need to change the data type in the threshold arrays as defined in trees_fpga.hpp . The tool takes care of converting the decimal values to the float_fixed format. Additionally, quantization and overflow, when assigned to lower precision types like 16 or 8 bits, are handled automatically by Vitis.In order to automate the generation of modified header files, we added extra functionalities to the python scripts as mentioned in 5.1. It is expected that there will be trade-offs between resources, performance, and model accuracy when applying precision scaling. We proceed to the analysis of the findings after presenting the integer representation.

6.2 Integer representation

Our exploration of fixed-point representation led us to experiment with other possible representations for the threshold matrices. Initially, we considered integer representation due to the fact that arithmetic operations and comparisons with integers are faster compared to floating-point numbers, thanks to their simpler format. Furthermore, we later discovered that, in addition to the additional acceleration, the integer types exhibit better accuracy. Especially for low precision such as 16 and 8 bits, the integer models achieve significantly better accuracy compared to fixed-point counterpart.

Converting a float range to an integer range while preserving the distribution of features is achieved through the technique of quantization. Quantization is a method used to reduce the computational and memory costs of running inference by representing weights and activations with low-precision data types like 8-bit integers (int8) instead of the usual 32-bit floating point (float32)[24]. The scheme we applied is Range-Based Linear Quantization. Linear quantization converts floats to integers using a scaling factor. Range-based quantization calculates this factor based on the actual tensor values, potentially excluding outliers, while other methods use fixed or learned thresholds for clipping values. We applied a combination of both methods.

Two widely used quantization methods are symmetric and asymmetric[25]. In asymmetric mode, we map the min/max values in the float range to the min/max values in the integer range. This is achieved by using a zero-point (also known as quantization bias or offset) in addition to the scale factor. On the other hand, in symmetric mode, we select the maximum absolute value between min/max without using a zero-point. As a result, the floating-point range we are effectively quantizing becomes symmetric with respect to zero, and so does the quantized range. We chose the symmetric method as it led to better accuracy.

The calculation of the scale factor q_i of a feature column of dataset, denoted as f_i, is given by the following formula:

$$q_i = \frac{(2^n - 1)/2}{max(abs(f_i))} \tag{2}$$

For each feature column i of the dataset we calculate, by applying the above formula, the corresponding q_i. Given the value xf_i of the column f_i, the quantized value qx_i is obtained by rounding the product of the scale factor q_i and xf_i as show below.

$$qx_i = round(q_i * xf_i) \tag{3}$$

The features of the dataset are not symmetric around 0 as they have a range of (0,1). As a result, the quantized range for N bits will be $(0,2^N - 1)$, even though we apply symmetric quantization. This serves our purpose well as it maintains a consistent distribution, depending on the value of N, ensuring better predictive capability of

the quantized model.In correspondence with fixed-point representation, we experimented with conversion to integers for 8, 16, 24, and 32 bits. The resulting range for each case is as follows:

- 8 bits: (0,255)
- 16 bits: (0,655.35)
- 24 bits: (0,16.777.215)
- 32 bits: (0,4.294.967.295)

Following the above procedure, we generated the quantized datasets for each of the aforementioned bit lengths. One possible approach to create a quantized model could be to retrain a RF model using the quantized dataset. The disadvantage of this approach is that we are not transforming an existing model, as we did with the fixed-point representation, instead we generate a new one with a different structure and predictive ability. Comparing the integer model with the floating-point and fixed-point models, in terms of resource utilization and accuracy loss, may not be valid since they would generally be different models. For this reason, we transformed the model, we have been presenting so far, into a quantized one.

As previously mentioned at 4.1, within the inner workings of each node of a decision tree, information is stored about which feature f_i will be used for the comparison and the corresponding threshold value $x_f i$ of the feature. Knowing the feature f_i and subsequently the scale factor q_i, we can perform symmetric quantization according to the formulas 2 3. The result of this process is that we can convert the float arrays of thresholds for each decision tree into quantized integer form. We added this conversion process as an extra functionality to the automated scripts, as mentioned in previous chapters. The parameter we are exploring is the number of bits, N, for integer representation.

The format of the threshold arrays for fixed-point and integer representation is similar to that for floating-point. In correspondence to the example in the previous section 5, we have the following:

- fixed_float threshold_1[1701]
- fixed_int threshold_1[1701]

The Vitis tool also provides the capability to define Arbitrary Precision Integer Data Types through the ap_int.h library. Similarly to 6.1, we will use unsigned types with the $ap\_uint < W >$ command for integer representation. For integer representation, we only need to specify the bit length by parameterizing the variable W. For the same reasons we mentioned in the previous section 6.1, we used the typedef command to maintain the generated integer types under a common name, which is fixed_int. An example of the header file for 16-bit integers is shown in the listing15.

Listing 15: Header for integer type

```
#ifndef _HEADER_
#define _HEADER_
#include "ap_int.h"

#define DATATYPE_SIZE 16

#define ESTIMATORS 10
#define FEATURE_SIZE 32

typedef ap_uint<DATATYPE_SIZE> fixed_int;

#endif
```

The changes required in the code, as presented in the previous subsections 5.2.1 & 5.2.2, to apply precision scaling for both parallel and pipeline are made at two levels. The first level involves including the updated header files header.hpp and the trees_fpga.hpp, which contain the definitions of the fixed_point and fixed_int types, as well as the new arrays of the decision tree for each representation, respectively. The second stage simply involves replacing the float type within the functions with fixed_point or fixed_int type. The final step makes clear the utility of a common name for fixed_point or fixed_int, regardless of the number of bits. As an example at listings 16 & 17, in correspondence with the listing 7, we show the new definition of the dt_0 function for both representations.

Listing 16: Fixed point representation of decision tree function

```
void dt_0(const fixed_float row[FEATURE_SIZE], bool & prediction){
    #pragma hls inline off
    // int est_index=0;

    int node_index=0;

    loop_dt_0:for(int path=0; path<depth[0]; path++){
        if(children_left_0[node_index] != children_right_0[
            node_index]){
```

```
            fixed_float threshold_value = threshold_0[node_index];
            fixed_float row_value = row[feature_0[node_index]];

            node_index = ( row_value <= threshold_value) ?
                children_left_0[node_index] : children_right_0[
                node_index];
        }
    }
    prediction = value_0[node_index];
}
```

Listing 17: Integer representation of decision tree function

```
void dt_0(const fixed_int row[FEATURE_SIZE], bool & prediction){
    #pragma hls inline off
    // int est_index=0;

    int node_index=0;

    loop_dt_0:for(int path=0; path<depth[0]; path++){
        if(children_left_0[node_index] != children_right_0[
            node_index]){
            fixed_int threshold_value = threshold_0[node_index];
            fixed_int row_value = row[feature_0[node_index]];

            node_index = ( row_value <= threshold_value) ?
                children_left_0[node_index] : children_right_0[
                node_index];
        }
    }
    prediction = value_0[node_index];
}
```

With precision scaling, we agree to trade off resource utilization and performance with the accuracy of the model. In the figure 28, we compare the F-score (over the entire dataset) of the original (float) model with the corresponding fixed_point and fixed_int models for 8, 16, 24, and 32 bits. For 32 and 24 bits, the accuracy remains stable for both representations. However, for 8 and 16 bits, the fixed_point model significantly loses predictive ability, with the 8-bit model not making any correct classifications. The quantized model displays better results. For 16 bits, the F-score remains practically stable, and for 8 bits, it decreases slightly but enough to not meet the required 99% threshold, as defined in Chapter 4.1. Moving forward, we applied precision scaling only for 24 and 32 bits for float_fixed and 16, 24, and 32 bits for fixed_int.The analysis of the impact on performance and resource utilization will be conducted in the experimental section.

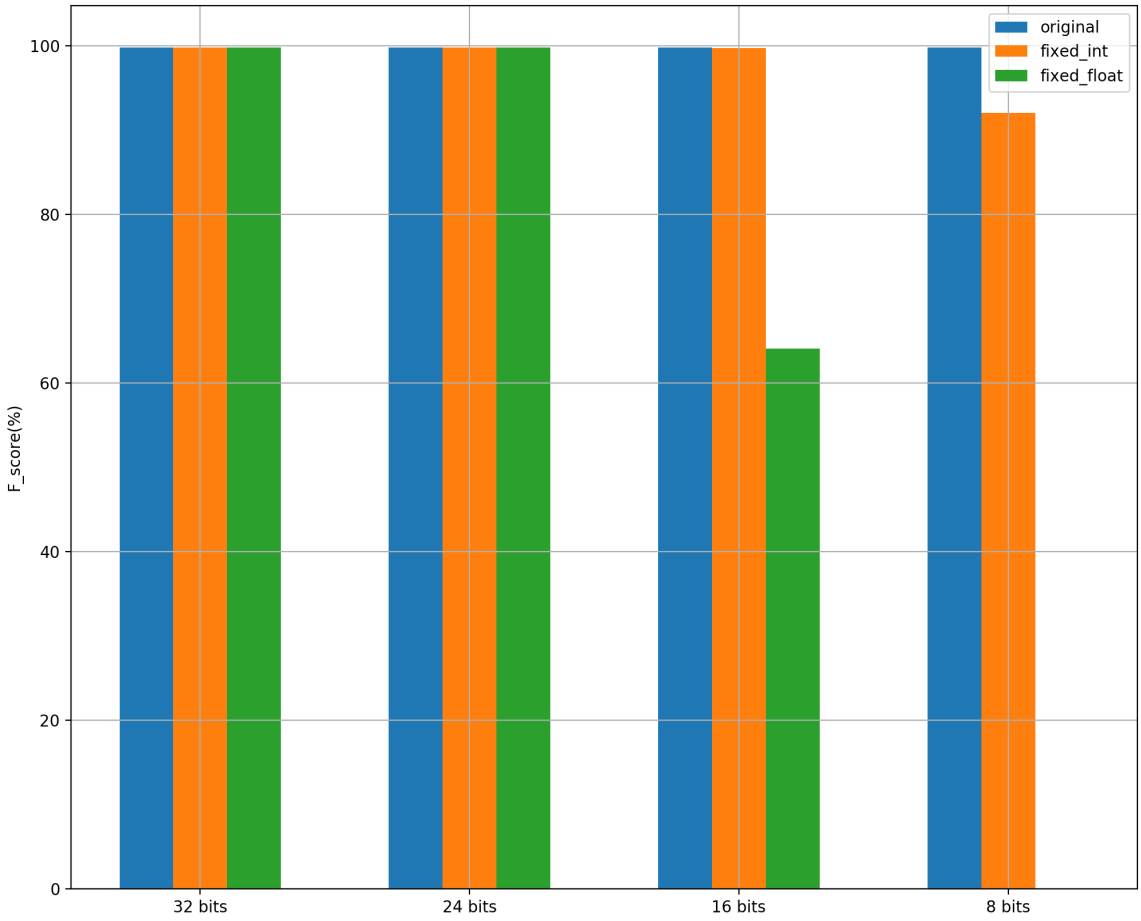

Figure 28: Precision scaling fscore[23].

7 Automated framework for hw design exploration

As mentioned in various parts of the thesis, many of the processes, from model training in Python to finding the optimal implementations using the Pareto plot, are automated. This led us to consolidate all the scripts into a unified framework capable of automating the entire exploration process by simply specifying the initial conditions.

The proposed framework consists of the following components:

- Model training in Python and data preprocessing.
- Code generation for HLS (High-Level Synthesis).
- Hardware emulation (hw_emu) for finding the Pareto-optimal model.
- HW implementation (hw) for design exploration in pipeline and parallel RFs architectures.
- Inference and resource utilization analysis.

The user inputs required are:

- The dataset on which training and inference will be performed.
- A list of how many estimators and features the user wants to include in the training.
- A list of input sizes for inference.

During the training phase, all combinations of estimators and features are trained. The train/test ratio has a default value of 0.3, but the user can adjust it. After training, the models are validated on the test dataset. Any model with an F-score less than the desired threshold is excluded from the next steps of the exploration. The default metric is the F-score, but other metrics such as accuracy, recall, or precision can be selected. The threshold value can also be adjusted, with a default value of 99%. The Scikit-learn framework is used for model training, and the generated models are saved in .pkl format. Once the training is completed, the models that meet the above condition are the ones that will be candidates for HW emulation and implementation. Data preprocessing involves normalizing

the dataset and performing quantization, with default values of 8, 16, 24, and 32 bits, but the user can specify other values.

The code generation part involves generating code in a format suitable for execution in Vitis. The goal is to automatically generate code by specifying high-level parameters. Such parameters include the type of architecture :pipeline or dataflow , the data types (float, fixed_float, or fixed_int), and the bit length for the last two cases. Additionally, for the dataflow architecture, the number of parallel RFs can be specified. Code generation is done by parameterizing template files according to the user's inputs. Another aspect of code generation is extracting the model structure into header files in the form of arrays. The result of this process is the creation of a directory containing all the code for the host and FPGA required to proceed to synthesising and later for inference.

Given the selected models from the training phase, the corresponding directories are automatically generated through the code generation module. The next step is hardware (HW) emulation to find the Pareto-optimal model, where we will proceed with HW implementation exploration. Exhaustive HW implementation for all models, applying all optimizations as presented in Chapters 5 and 6, would be prohibitive in terms of resources, including storage and time. On average, an HW implementation consumes 4 GB of storage and takes about 8 hours. Instead of an exhaustive search, we propose a greedy heuristic.

Specifically, for the models, we will apply HW emulation using a dataflow architecture with rf_size=1 and a standard 32 bit float data type. The reason for choosing this configuration is that it is the simplest one that allows us to have a good understanding of resource utilization and iteration latency. With this configuration, we automatically generate the Pareto plot of utilization/latency. The candidates for further exploration are the models located on the Pareto front. Having the Pareto front, we can either explore all of them or select one using some criteria. We followed the second option, and the criterion we chose to select a model is the F-score. More specifically, from the Pareto front, we choose the model with the best F-score or any other desired metric the user wish to use.

Once we have selected a model, we proceed with HW implementation exploration. In this stage, we implement both pipeline and parallel architectures while applying precision scaling. We have ex-

tensively discussed these concepts in 5 and 6. Initially, we determine the bit length for fixed_float and fixed_int data types that satisfy the F-score threshold defined in the training phase. Knowing the parameters of precision scaling, we apply them to the exploration of both architectures. Our goal is to find the configuration that provides the best inference performance. For the pipeline architecture, we only apply precision scaling. While for the parallel architecture, we perform a binary search in the space of rf_size, starting from rf_size=1, until we find the maximum number of parallel RFs that fit within the FPGA. We set an upper limit of 75% of the available resources, but this limit can be adjusted. All the necessary directories and codes are automatically generated through the code generation modules.

During this exploration phase, parallelization is important due to the time-consuming nature of HW implementation. By parallelizing the build process, multiple configurations can be processed concurrently, speeding up the overall exploration process. This parallelization can be controlled by adjusting the size of the process pool to prevent resource exhaustion.You used the Python 'multiprocessing' library for this purpose. This approach was chosen initially because even for the exploration of a single model dictated from the hardware emulation, it can take weeks. The result of the above process is the rapid exploration of the design space.

After exploration, the next steps involve inference and testing. Inference can be performed on input sizes specified by the user or generated by the framework based on the least common multiple (LCM) of the RF sizes, as discussed in 5. Reports are generated, including information on inference time, prediction accuracy, and resource utilization on the FPGA. Additionally, utilization/inference time Pareto plots are generated for each input size, and configurations lying on the Pareto front are returned. These reports provide valuable insights for further analysis.There are many possibilities for data analysis in the reports, one of which we implemented concerns the speedup of the best configuration compared to native execution in C++ and Python.

In summary, the proposed framework automates the entire process, from model training to HW implementation exploration, and provides tools for efficient exploration and selection of the most suitable configurations for HW implementation.At figure 29, we present

a conceptual diagram illustrating the framework's components and workflow.

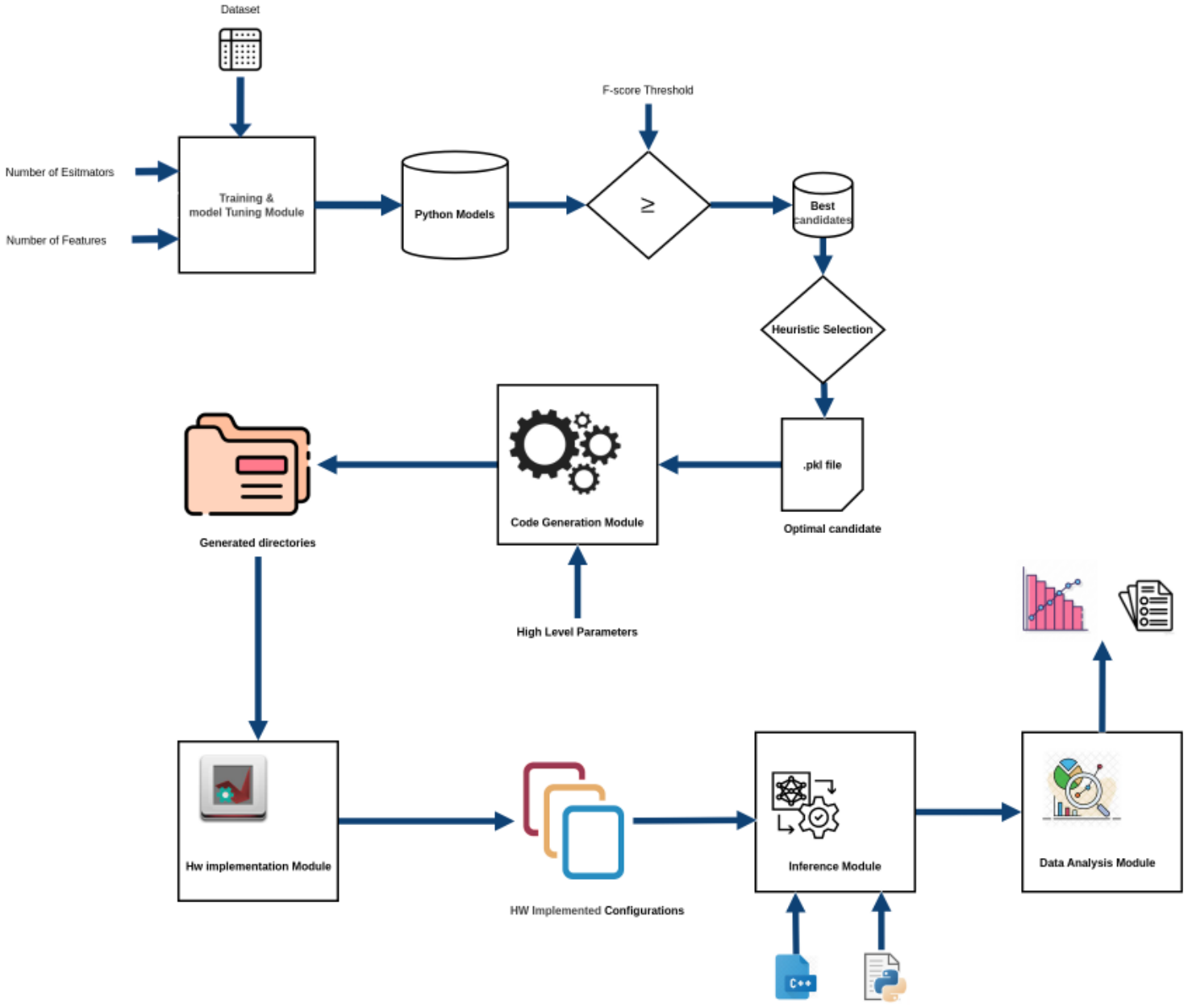

Figure 29: Components and workflow of the framework.

8 Experimental

In this chapter, we will present our findings regarding the performance and resource utilization of the pipeline and parallel architectures as presented in Chapter 4, applying precision scaling from Chapter 5. We will demonstrate the impact on resource utilization while maintaining or improving the inference time. The best-performing models in terms of inference time will be compared to their respective implementations in C++ and Python as they run natively.

8.1 Device set up

The target FPGA board for this work is the Alveo U200 Data Center Accelerator Card xilinx_u200_xdma_201830_2 . The AMD Alveo U200 accelerator card is custom-built UltraScale+ FPGA that run optimally (and exclusively) on the Alveo architecture. The Alveo U200 card uses the XCU200 FPGA also the AMD stacked silicon interconnect (SSI) technology to deliver breakthrough FPGA capacity, bandwidth, and power efficiency. This technology allows for increased density by combining multiple super logic regions (SLRs). The XCU200 comprises three SLRs. The device connect to 16 lanes of PCI Express® that can operate up to 8 GT/s (Gen3). Also connects to four DDR4 16 GB, 2400 MT/s, 64-bit with error correcting code (ECC) DIMMs for a total of 64 GB of DDR4. Furhtermore the device connects to two QSFP28 connectors with associated clocks generated on board. The following figure 30 show the SLR regions along with the PCIe, DDR4 and QSFP28 connections for the Alveo U200 . The U200 card has three SLRs [26].

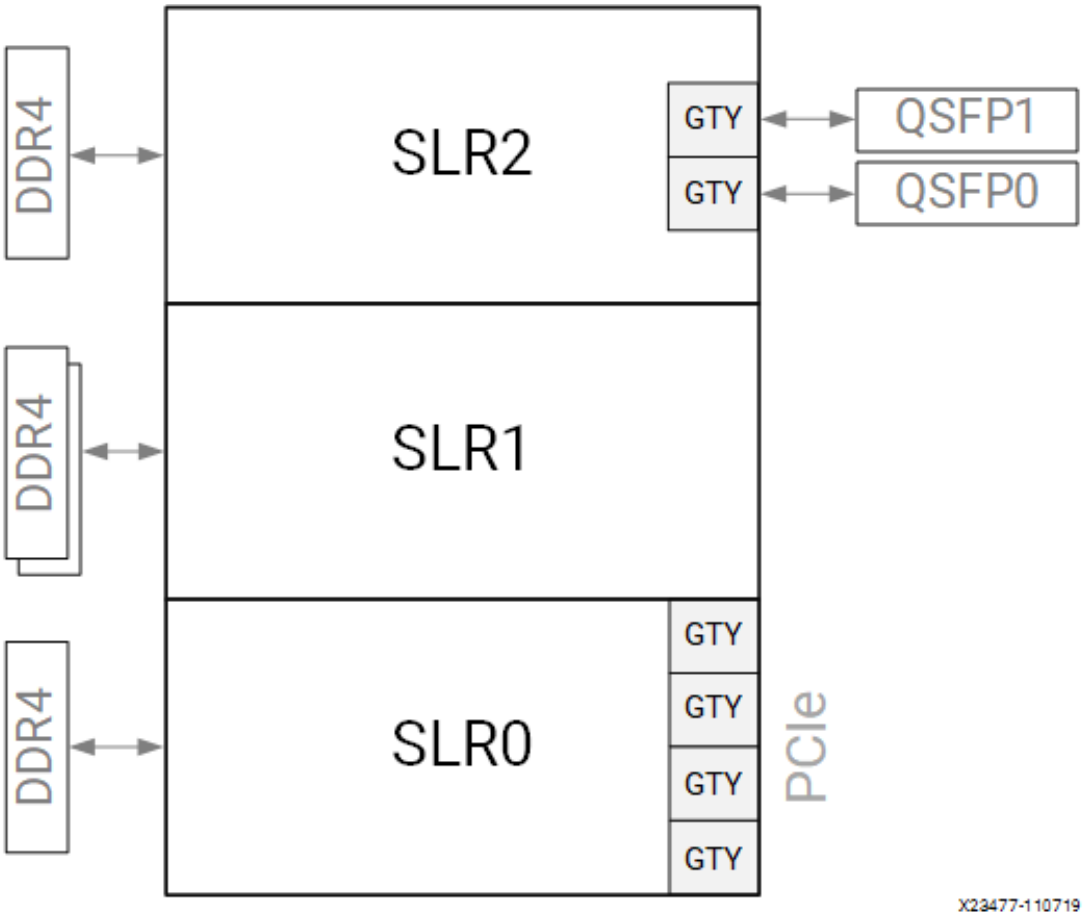

Figure 30: Floorplan of the XCU200 Device [26].

The resources available as presented through the Vitis IDE are shown at figure 31.

Resources

Resource	Total Count
BRAM	2160
DSP	6840
LUT	1182240
FF	2364480
REG	2186301

Figure 31: Resources of U200 card.

8.2 Results - Evaluation

Before we proceed with the experimental evaluation, we must explain how we selected the model, specifically the combination of estimators and the number of features, for which we conducted the hardware exploration as described in the previous chapters. We used the heuristic presented in the Framework chapter 7. From the models we proposed in the training section 4.1 we examined only those that had an F-score greater than 99%. We performed hardware emulation using parameters for the coarse-grained parallel RF architecture with RF_SIZE=1 and without precision scaling. Out of the 37 generated configurations, we identified which ones were on the Pareto front, as shown in the figure 32.

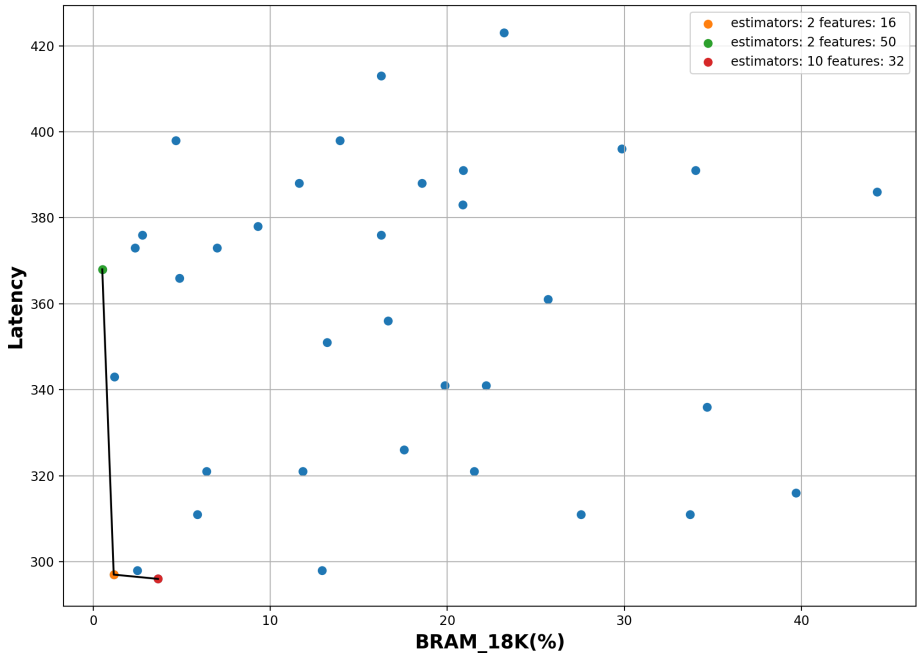

Figure 32: Selection of optimal model using hw emulation heuristic.

The heuristic we employed to determine the optimal configuration was to choose the one with the highest F-score from those on the Pareto front. In this case, the optimal model consisted of 10 estimators trained over 32 features. This choice is consistent with the analysis we conducted in the previous chapters.

In the figures 33, 34 & 35, we display bar plots illustrating the impact of precision scaling over the BRAM, LUT and FF utilization for both the pipeline and parallel architectures. The proposed configurations have both pipeline and parallel architectures with the later consisting of 1, 2, 4, 8, 16, and 24 parallel RFs. Additionally, we applied the conversion from float 32 bits to fixed_float for 32 and 24 bits, as well as to fixed_int for 32, 24, and 16 bits. This accumulative reduction can have a significant impact, especially for large designs like Parallel RF SIZE=24, resulting in a decrease from 70.69% to 39.88% BRAM usage.

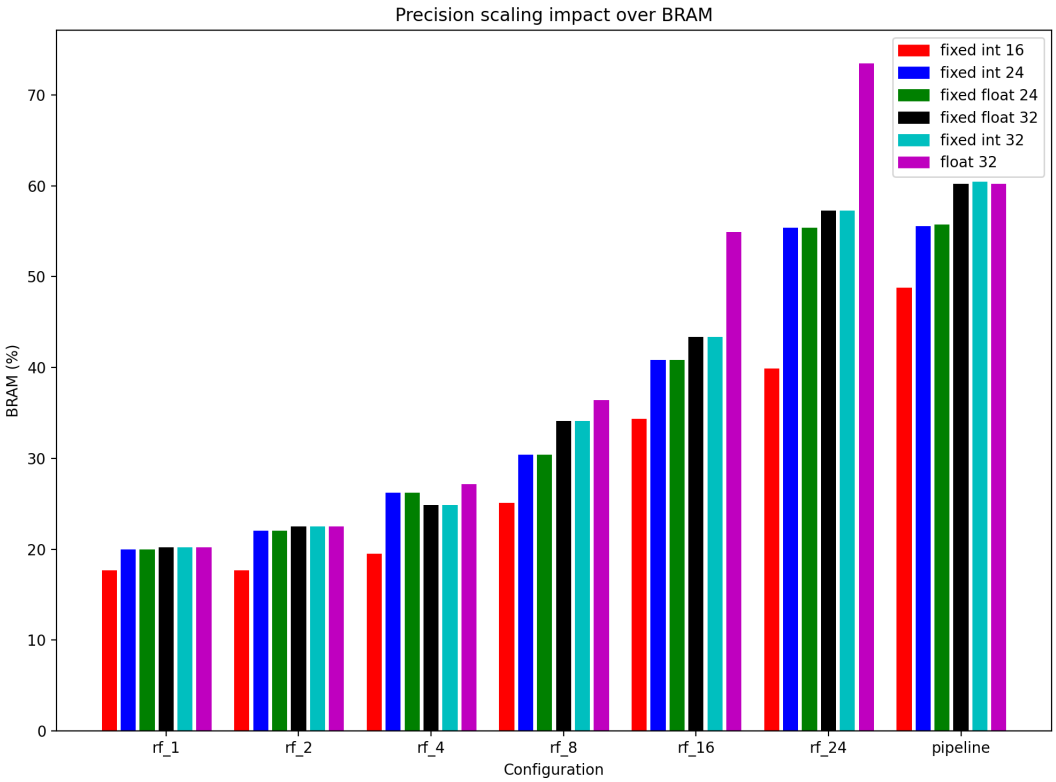

Figure 33: Impact of precision scaling over BRAM Utilization(%).

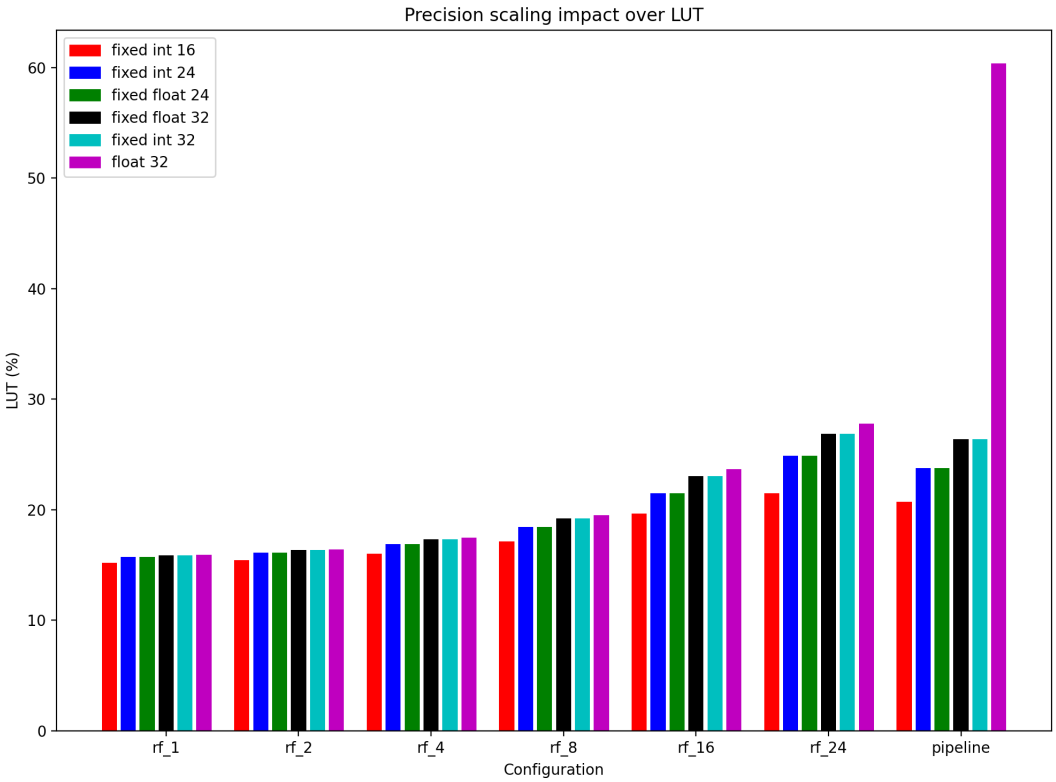

Figure 34: Impact of precision scaling over LUT Utilization(%).

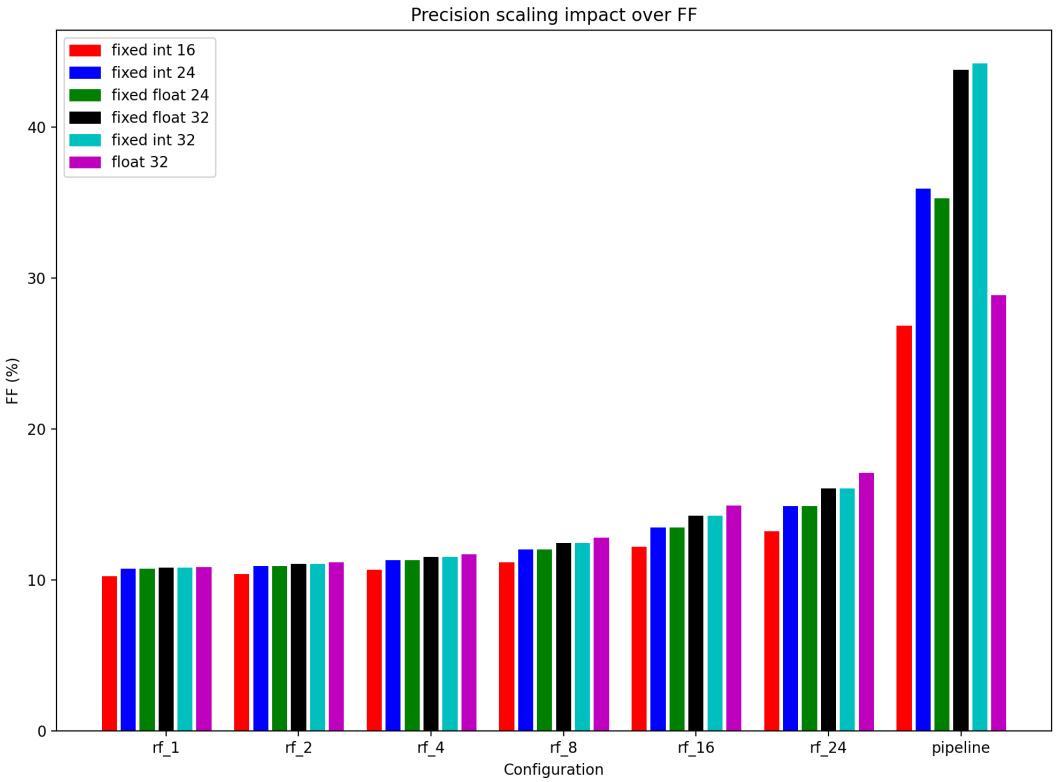

Figure 35: Impact of precision scaling over FF Utilization(%).

In order to determine the best configurations, taking into account both inference time and resource utilization, we designed Pareto plots. The yaxis provides the inference time for each configuration, and the xaxis represents resource utilization. We chose to create plots specifically for BRAM since, as indicated by the previous box plots, it is the most dominant resource and depletes faster.We generated different plots for each input size, enabling us to identify the best configurations for both small and large test datasets. As mentioned in Chapter 5.2.2, the input sizes are 480, 4800, 48000, and 288334 (the entire dataset).

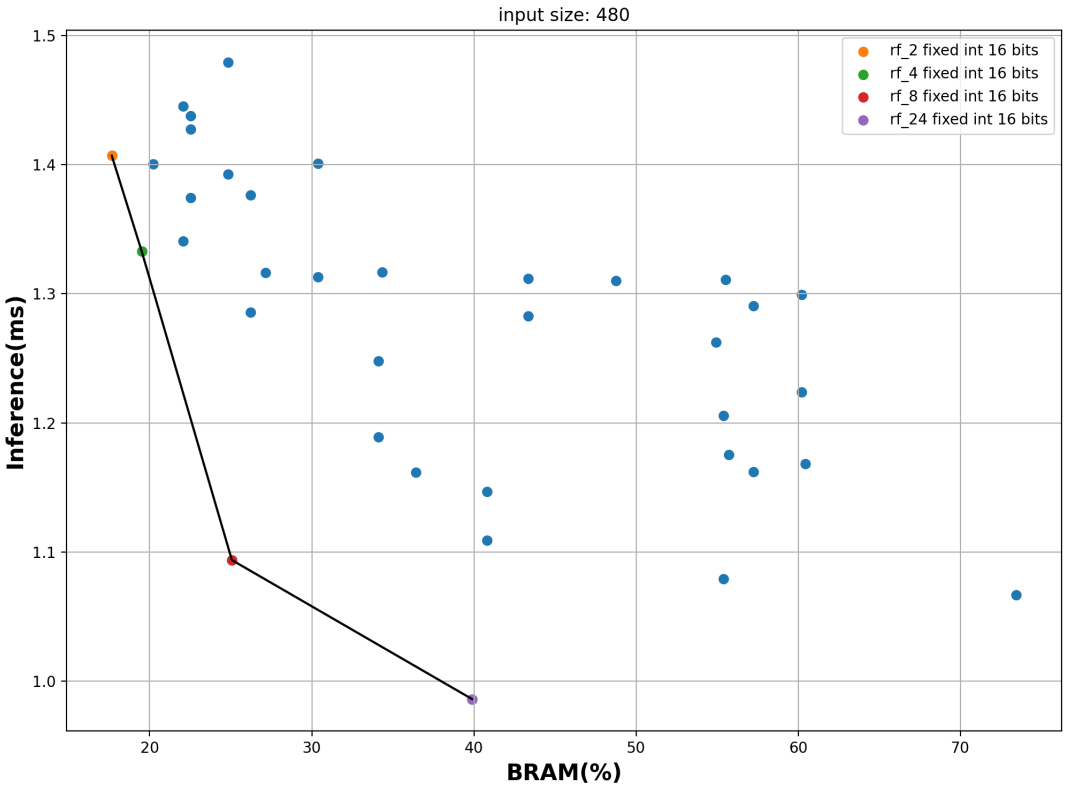

Figure 36: Pareto plot for input size 480 samples.

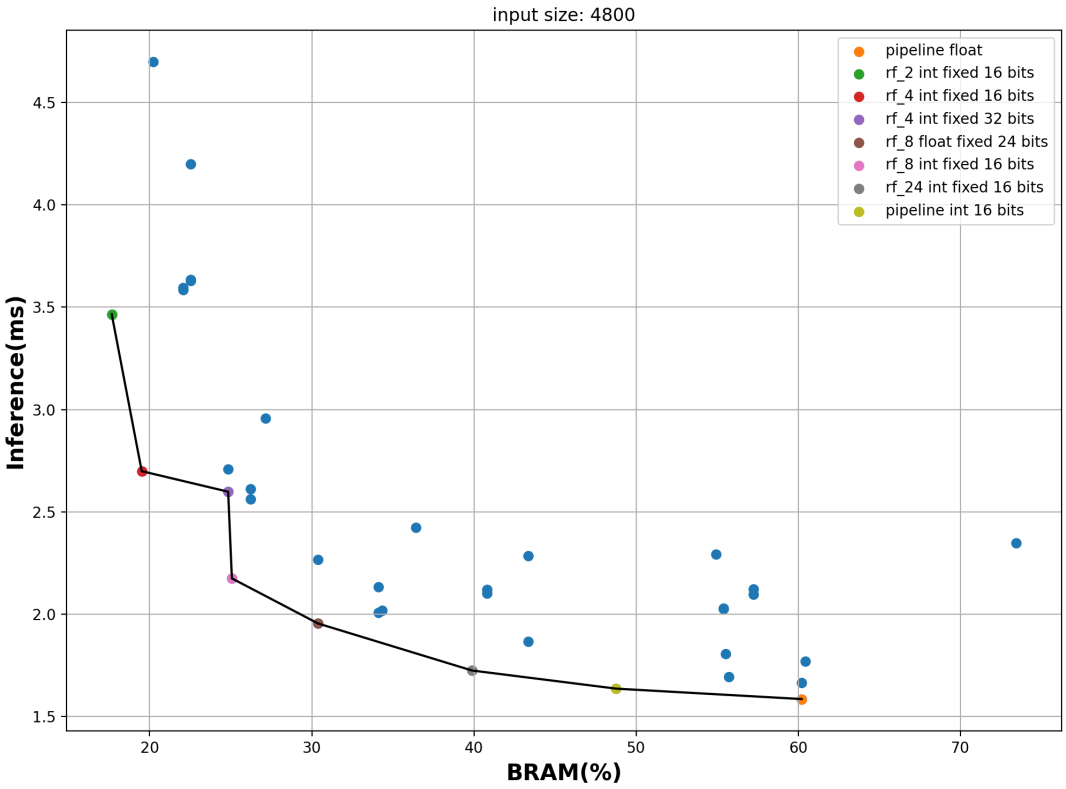

Figure 37: Pareto plot for input size 4800 samples.

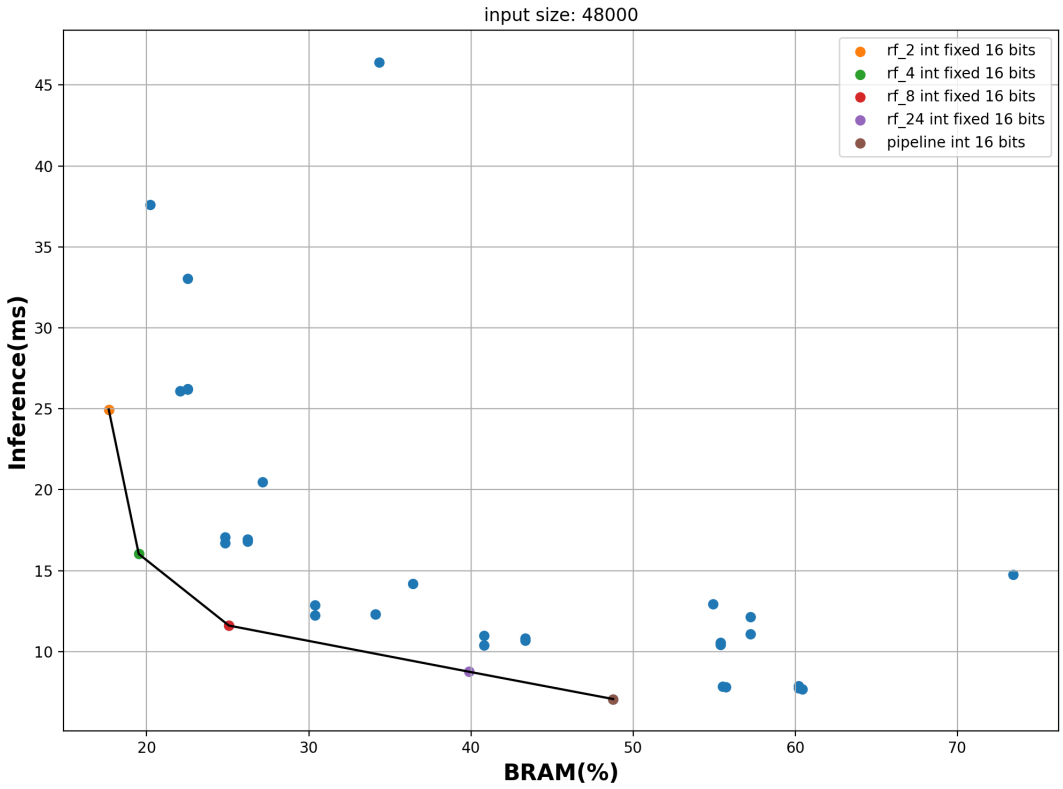

Figure 38: Pareto plot for input size 48000 samples.

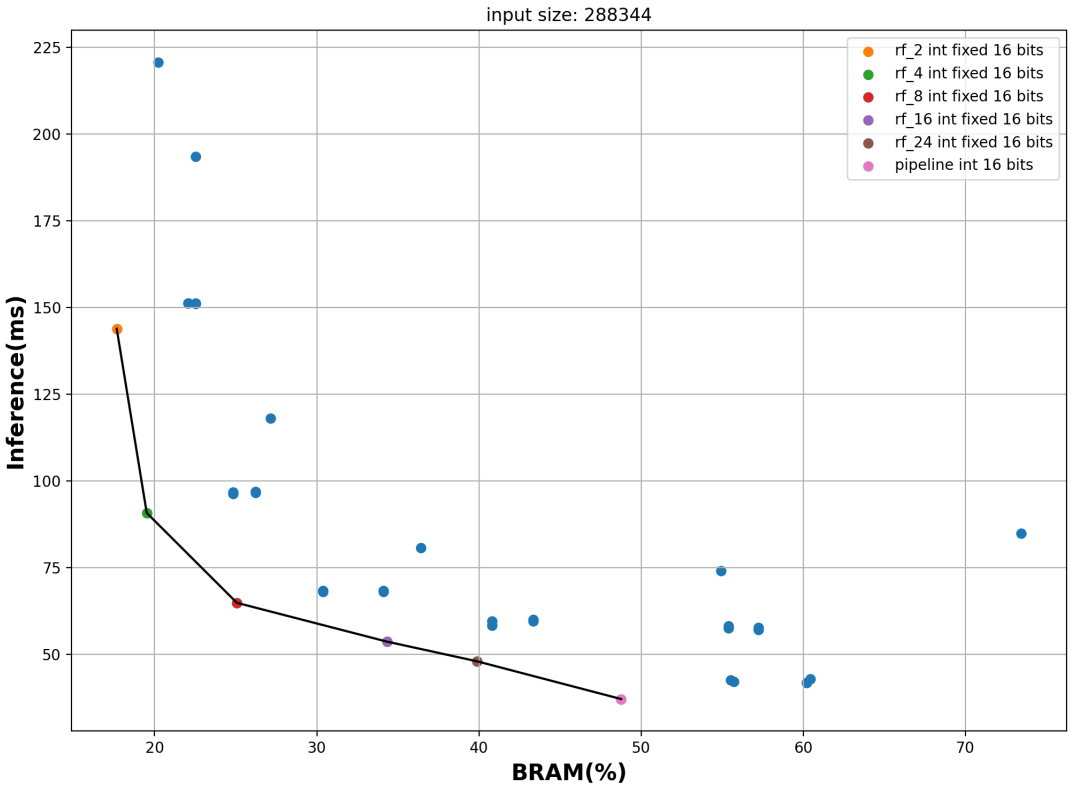

Figure 39: Pareto plot for whole test dataset.

We highlighted the configurations that are on the Pareto front in the Pareto diagrams 36, 37, 38 & 39. With the exception of the limited input amount of 480 samples, the fastest model from the Pareto fronts had a pipeline design and performed well in all circumstances. The parallel variant with 24 RFs comes in second, consolidating the hypothesis that more parallel RFs improve performance. On the Pareto fronts, practically all configurations have the trait that the ones that are faster and use less BRAM have precision scaling with integer 16 bits applied. As also seen in the aforementioned graphs 33, 34 & 35, precision scaling application significantly impacts resource reduction while also enhancing performance. We selected the two quickest configurations from the two architectures

for the comparison with C++ and Python, namely pipeline fixed integer 16 bits and parallel 24 RF integer 16 bits.We also included the unoptimized RF design that was presented in 4.2

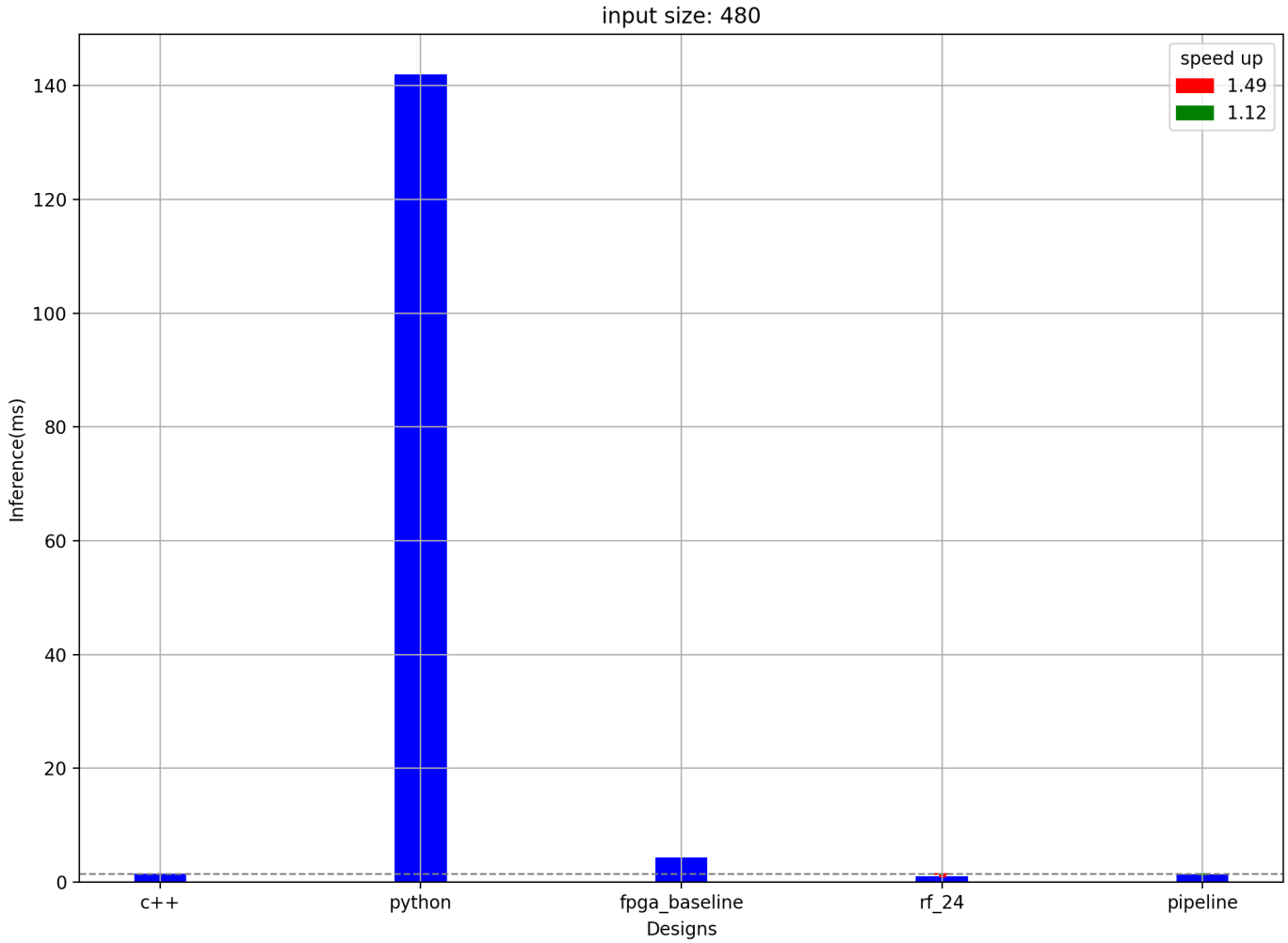

Figure 40: Inference comparison for 480 samples.

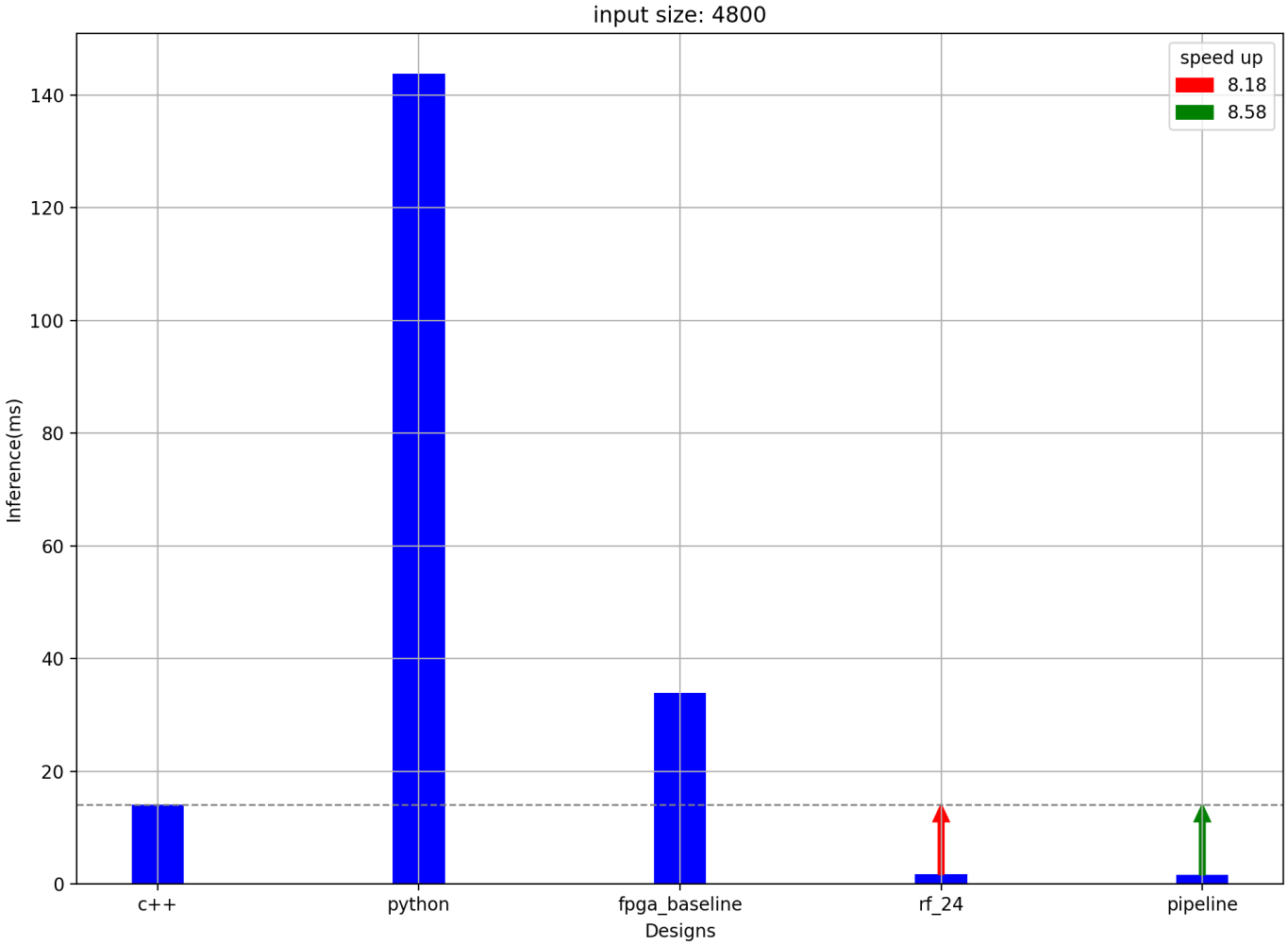

Figure 41: Inference comparison for 4800 samples.

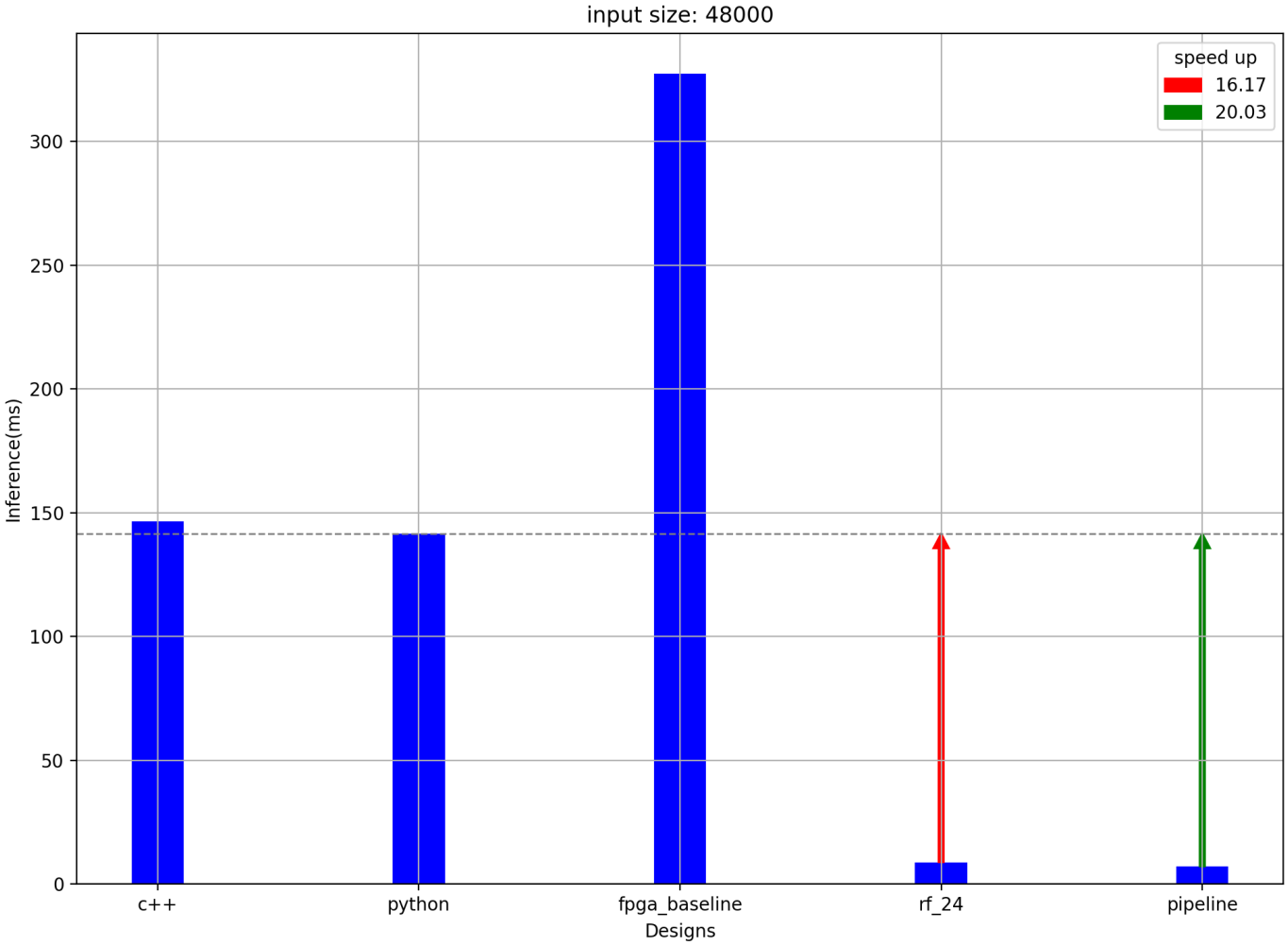

Figure 42: Inference comparison for 4800 samples.

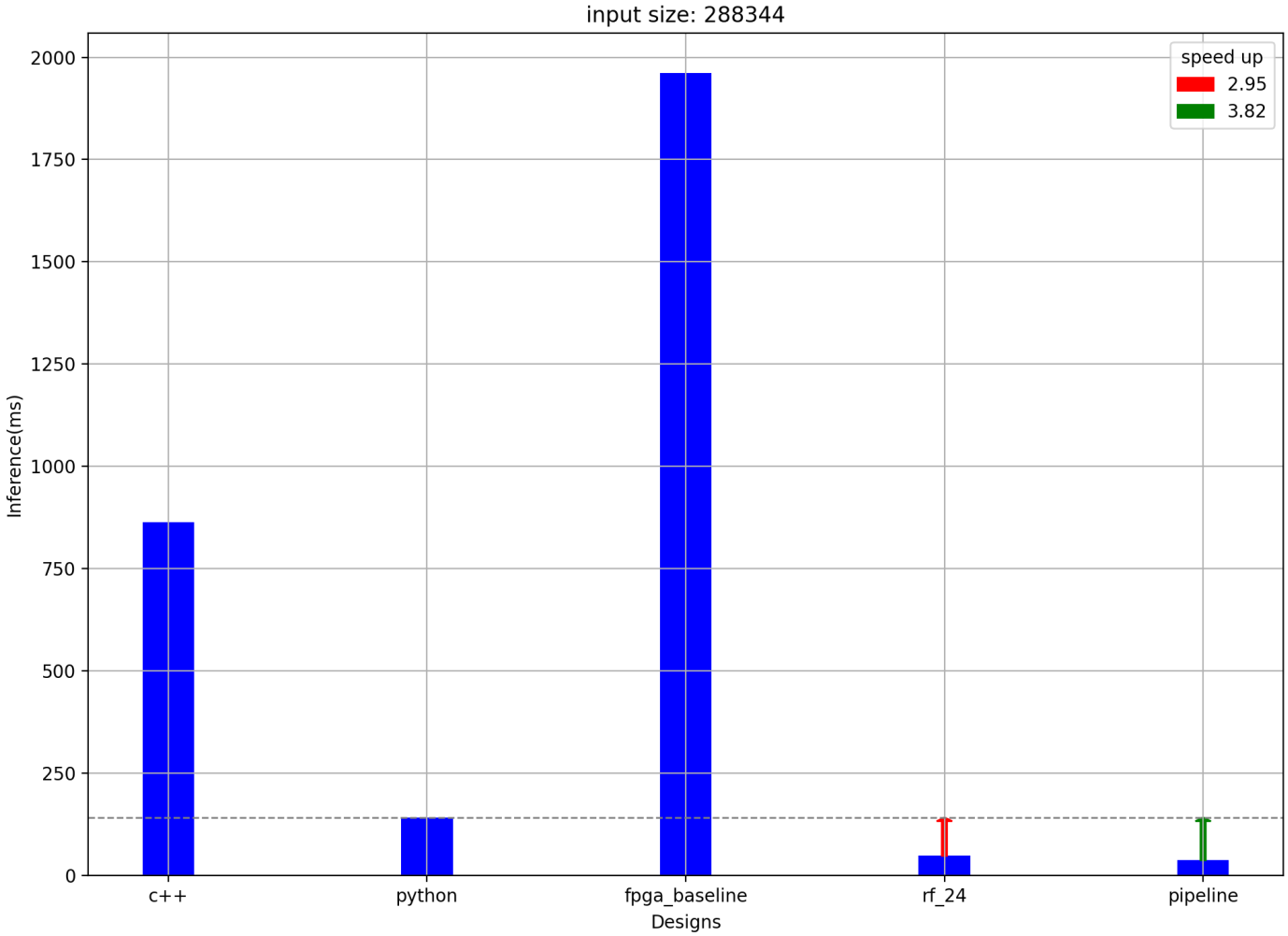

Figure 43: Inference comparison for whole test dataset.

From the above diagrams, we can draw valuable conclusions about the utility of hardware acceleration. For all input sizes, we manage to achieve acceleration. The lowest speedup, 1.5, is observed for the smallest dataset size (480 samples) when compared to inference in C++. This picture changes as the input size increases. For all

other cases, 4800, 48000 samples, and the whole dataset, we achieve much better acceleration. The best configuration for these sizes is the pipeline architecture with integer precision scaling to 16 bits, with the exception of 480 samples, where the parallel implementation with 24 RF achieves better performance. Specifically, the speedup achieved is shown in the table 8. It's interesting to note that for inference in Python, the time remains practically constant at around 140 ms. Because of this, as the test size increases, the Python inference time approaches that of C++, with the case of more than 48000 samples showing better performance. Our implementations are compared each time with the faster implementation in both C++ and Python, and the corresponding speedup is calculated. The best speedup achieved is 20.03 for 48000 samples.

input size	compare to	native time(ms)	best fpga time(ms)	speed up
480	c++	1.47	0.99	1.5
4800	c++	14.07	1.64	8.6
48000	python	141.61	7.07	20
288344	python	141.8	37.15	3.8

Table 8: Achieved speed up.

All of the above lead us to the following conclusions.For small input sizes, around 500 samples, there is little benefit from hardware acceleration. The size is small enough that the transfer time overhead from the host to the FPGA outweighs any acceleration achieved through parallelism in our implementations. In this case, inference can be done natively in C++ without a significant difference. As the input size increases, it becomes clear that the above overhead is overshadowed by the acceleration achieved. The ideal input size is around 50,000 samples, where our models demonstrate the best performance. For input sizes multiple times larger than the whole dataset, it's possible that the Python implementation achieves better results due to batch processing. However, it's hard to imagine cases where inference on such a large dataset is needed for security-critical intrusion detection applications, such as the one

examined in this thesis.

9 Conclusions

9.1 Summary

In the context of this thesis, we examined the Random Forest machine learning algorithm for anomaly detection in industrial facilities. We researched ways to achieve hardware acceleration in an FPGA for RF during inference. We identified the parts of the algorithm that can be accelerated and parallelized, leveraging the FPGA's capabilities. The decision trees that make up the RF are independent of each other and can be parallelized. Based on this property, we proposed two different architectures: Coarse-Grained Parallel RFs and Pipeline RFs. In both of these architectures, estimators are executed in parallel. In the first approach, we have multiple copies of the same RF, introducing a second level of parallelism for concurrent processing of multiple input samples, equal to the number of copies. In the second architecture, a pipeline approach was followed where, after an initial latency, an iteration interval (II) of 1 was achieved, meaning a new classification was produced in each new cycle. Additionally, we introduced the precision scaling optimization technique to reduce resource utilization and inference time. By comparing each time with the fastest RF implementation running natively in C++ and Python, we managed to achieve acceleration for all tested sample sizes. The best speedup we measured was 20 for 48,000 samples. A sub product of this thesis was the creation of a tool for automating the design exploration process on the FPGA, requiring minimal user interaction.

9.2 Future Work

In future work, we aim to explore additional techniques for acceleration on FPGAs or even different hardware platforms, such as embedded FPGAs. The research conducted in this thesis can be easily transferred to eFPGAs, which are gaining popularity in various applications closely related to our work, such as edge computing.Another interesting study could involve the acceleration of another tree-structured machine learning algorithm suitable for anomaly detection, such as Isolation Forests (IF). IF bear several similarities to RF, with a key difference being their ability to perform unsupervised anomaly detection. It would be worthwhile to compare the

two algorithms in terms of hardware acceleration performance and predictive capabilities.

The conclusion of this thesis work is that the fields of hardware acceleration and anomaly detection are continually evolving, and they will certainly continue to be of interest in the future.

References

[1] Varun Chandola & Arindam Banerjee & Vipin Kumar (2009) *Anomaly Detection: A Survey*

[2] Sridhar Adepu & Khurum Nazir Junejo & Aditya Mathur (2016) *A Dataset to Support Research in the Design of Secure Water Treatment Systems.*

[3] Sklearn Normalizer class Link

[4] Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E. *Scikit-learn: Machine Learning in Python*

[5] Random Forest essembler. Link.

[6] Decision Tree structure. Link.

[7] Simon D. Duque Anton & Sapna Sinha & Hans Dieter Schotten (2019) *Anomaly-based Intrusion Detection in Industrial Data with SVM and Random Forests.*

[8] Simon D. Duque Anton & Sapna Sinha & Hans Dieter Schotten (2019)*Anomaly-based Intrusion Detection in Industrial Data with SVM and Random Forests*

[9] Md. Al Mehedi Hasan & Mohammed Nasser & Biprodip Pal & Shamim Ahmad*Support Vector Machine and Random Forest Modeling for Intrusion Detection System (IDS)*

[10] Mahmudul Hasan & Md. Milon Islam & Md Ishrak Islam Zarif & M.M.A Hashem(2019) *Attack and anomaly detection in IoT sensors in IoT sites using machine learning approaches*

[11] Brian Van Essen & Chris Macaraeg & Maya Gokhale & Ryan Prenger (2012)*Accelerating a random forest classifier: multi-core, GP-GPU, or FPGA?*

[12] Roger J.Lewis(2000) *An Introduction to Classification and Regression Tree(CART) Analysis*

[13] Akshay Valsaraj & Ithihas Madala & Nikhil Garg & Mohit Patil & Veeky Baths *Motor Imagery Based Multimodal Biometric User Authentication System Using EEG*

[14] José MARTIN Flores AlbinoRigoberto Vizcaya & Rigoberto Vizcaya (January 2018) *Deep Learning para la Detección de Peatones y Vehículos*

[15] Introduction to FPGA Design with Vivado High-Level Synthesis (UG998) Link.

[16] Logic Block Link.

[17] Andrew Boutros & Vaughn Betz(2021) *FPGA Architecture:Principles and Progression*

[18] AXI Basics 1 - Introduction to AXI(2023) Link

[19] Vitis High-Level Synthesis User Guide (UG1399) Link

[20] Vivado Design Suite User Guide Link

[21] Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393) Link

[22] Confusion Matrix figure Link

[23] Floating point representation Link

[24] Quantization definition Link

[25] Asymmetric vs Symmetric Link

[26] Alveo U200 and U250 Data Center Accelerator Cards Data Sheet (DS962) Link

[27] Alveo U200 Data Center Accelerator Card Link