Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Optimizing FPGA-based accelerators for LSTM models towards low latency applications

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Γεράσιμος Γ. Καναρόπουλος

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής

Αθήνα, Οκτώβριος 2022

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Optimizing FPGA-based accelerators for LSTM models towards low latency applications

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Γεράσιμος Γ. Καναρόπουλος

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Οκτωβρίου 2022.

....	....	....
Δ. Σούντρης	Π. Τσανάκας	Σ. Ξύδης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκ. Καθηγητής
		Χαροκόπειο Παναεπιστήμιο

Αθήνα, Οκτώβριος 2022

....

Ιωάννης Γεράσιμος Γ. Καναρόπουλος

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

Copyright © Ιωάννης Γεράσιμος Γ. Καναρόπουλος, 2022.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η ανίχνευση ανωμαλιών είναι ένα σημαντικό πεδίο ερευνών για μεγάλο χρονικό διάστημα. Η μηχανική μάθηση έχει επιτύχει σημαντική πρόοδο τα τελευταία χρόνια και έχει βρει εφαρμογές σε πολλούς διαφορετικούς τομείς. Ιδιαίτερα τα δίκτυα μακράς βραχείας μνήμης (LSTMs) έχουν να επιδείξουν υποσχόμενα αποτελέσματα όταν ασχολούνται με δεδομένα εξαρτώμενα από το χρόνο, όπως χρονοσειρές. Αν και αυτές οι εξελίξεις παρέχουν όλο και πιο αποτελεσματικά εργαλεία, δεν καλύπτουν επαρκώς την ανάγκη για χαμηλή καθυστέρηση στις προβλέψεις, η οποία γίνεται πιο απαιτητική καθώς η ποσότητα των δεδομένων για επεξεργασία αυξάνεται με την πάροδο του χρόνου, ούτε την απαίτηση για χαμηλή κατανάλωση ενέργειας. Οι επιταχυντές που βασίζονται σε FPGA έχουν τραβήξει την προσοχή λόγω της καλής τους επίδοσης, της υψηλής ενεργειακής απόδοσης και της μεγάλης ευελιξίας τους. Σε αυτή τη διπλωματική, παρουσιάζουμε μια προσέγγιση για έναν επιταχυντή που βασίζεται σε FPGA για μοντέλα ανίχνευσης ανωμαλιών με LSTM. Ο γενικός στόχος είναι να αναπτυχθεί αρχιτεκτονική και κυκλώματα σε FPGA για την υποστήριξη σχετικών εφαρμογών με επίκεντρο τη χαμηλή καθυστέρηση, ενώ ταυτόχρονα να μην κάνει παραχωρήσεις όσον αφορά την ακρίβεια. Ο επιταχυντής FPGA έχει συντεθεί χρησιμοποιώντας σύνθεση υψηλού επιπέδου στην πλατφόρμα Vitis Unified Software και στοχεύει την πλακέτα Xilinx MPSoC ZCU104 και την κάρτα επιτάχυνσης Xilinx Alveo U280. Η αξιολόγηση δείχνει ότι ο επιταχυντής μπορεί να επιτύχει επιτάχυνση περίπου 1.5 έως 2.5, σε σχέση με τον προεπιλεγμένο συμπερασμό του TensorFlow, χωρίς σημαντική πτώση της ακρίβειας.

Λέξεις κλειδιά: μηχανική μάθηση, ανίχνευση ανωμαλιών, LSTM, FPGA, σύνθεση υψηλού επιπέδου, επιτάχυνση υλικού

Abstract

Anomaly detection has been an important topic of research for a long time. Machine learning has achieved major breakthroughs in recent years and has found applications in many different fields. Long Short-Term Memory networks (LSTMs) in particular have shown promising results when dealing with time dependent data like time series. While these developments provide increasingly efficient tools, they do not sufficiently address the need for low prediction latency, which becomes more demanding as the amount of data for processing increases over time, nor the requirement for low energy consumption. FPGA-based accelerators have attracted attention due to their good performance, high energy efficiency and great flexibility. In this thesis, we present an approach for an FPGA-based accelerator for LSTM anomaly detection models. The overall goal is to develop an application with focus on low latency, while at the same time not making concessions in terms of accuracy. The FPGA-accelerator has been synthesized using high level synthesis on Vitis Unified Software Platform and is targeting the Xilinx MPSoC ZCU104 board and Xilinx Alveo U280 acceleration card. The evaluation shows that the accelerator can achieve a speedup of about 1.5 to 2.5, when compared with the default TensorFlow inference, without any significant accuracy drop.

Keywords: machine learning, anomaly detection, LSTM, FPGA, high level synthesis, hardware acceleration

Ευχαριστίες

Θα ήθελα, εν πρώτοις, να ευχαριστήσω τον επιβλέποντα καθηγητή κ. Δημήτρη Σούντρη της σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του ΕΜΠ για την εμπιστοσύνη που μου έδειξε για την εκπόνηση της παρούσας διπλωματικής αλλά και την ξεχωριστή εκπαιδευτική διαδικασία που αυτή μου προσέφερε. Επιπλέον θέλω να ευχαριστήσω τους Δημήτρη Δανόπουλο, Ιωάννη Σταμούλια και Γιώργο Λεντάρη για τις συμβουλές τους και την καθοδήγησή τους σε όλα τα στάδια εκπόνησης της παρούσας εργασίας κατά τις εβδομαδιαίες μας συναντήσεις. Κλείνοντας, θα ήθελα να ευχαριστήσω τους φίλους μου, συναδέλφους και μη, χάρη στους οποίους τα φοιτητικά χρόνια θα μου μείνουν αξέχαστα αλλά και τους γονείς μου και τον αδερφό μου για την αμέριστη στήριξη που μου έδειξαν κατά τη διάρκεια των σπουδών μου.

Contents

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Contents 11

Εκτενής Περίληψη 13
- Εισαγωγή - Κίνητρο 13
- Θεωρητικό Υπόβαθρο 14
  - Μηχανική μάθηση 14
  - Τεχνητά νευρωνικά δίκτυα 15
  - Δίκτυα μακράς βραχείας μνήμης 16
  - Συστοιχία επιτόπια προγραμματιζόμενων πυλών 18
  - Σύνθεση υψηλού επιπέδου 18
  - TensorFlow και Keras 19
- Ανάπτυξη και επιτάχυνση LSTM σε FPGA 20
  - Μεταφορά μοντέλου python σε C++ 20
  - Δημιουργία πυρήνα για το FPGA 20
  - Βελτιστοποιήσεις πυρήνα 21
- Αποτελέσματα - Αξιολόγηση 22
  - Συσκευές 22
  - Σύνολα δεδομένων και μοντέλα που χρησιμοποιήθηκαν 22
  - Απόδοση 23
- Συμπεράσματα 27

Chapter 1: Introduction 29
- 1.1 Motivation 29
- 1.2 Thesis outline 30

Chapter 2: Background 31
- 2.1 Machine Learning 31
- 2.2 Artificial Neural Networks 32
  - 2.2.1 Modeling a neuron 32
  - 2.2.2 Neural Network Architectures 34
- 2.3 Long Short-Term Memory Networks 35
  - 2.3.1 Function of an LSTM cell 36
  - 2.3.2 Number of parameters of an LSTM cell 39
- 2.4 Field-Programmable Gate Arrays 40
  - 2.4.1 FPGA Architecture 41
  - 2.4.2 FPGA Components 42
  - 2.4.3 Comparison with other architectures 44
  - 2.4.4 FPGA Applications 47
- 2.5 High Level Synthesis 48

2.5.1 HLS Phases 49
2.5.2 Vitis Unified Software Platform 50
2.6 TensorFlow and Keras 52
2.6.1 Model Life Cycle 53

Chapter 3: LSTM deployment and acceleration on FPGA 55
3.1 Network creation and training in Python 55
3.2 Transferring the model to C++ 57
3.3 Creating a kernel for the FPGA 58
3.4 Kernel Optimizations 62
3.4.1 Code transformations 62
3.4.2 Optimization pragmas 62
3.4.3 Fixed point numbers 65

Chapter 4: Results - Evaluation 68
4.1 Device setup 68
4.1.1 Alveo U280 68
4.1.2 MPSoC ZCU104 70
4.2 Datasets and models used 72
4.2.1 Johnson & Johnson stock price 72
4.2.2 Numenta Anomaly Benchmark 74
4.3 Performance 74

Chapter 5: Conclusion 83
5.1 Summary 83
5.2 Future directions 83

Appendix 84
Transferring weights from python to C++ 84
Example of an LSTM layer in C++ 87
Example of Dense layer in C++ 88
Quantizing tanh and sigmoid 88
Python code for J&J dataset 89
Python code for NAB dataset 92
C++ FPGA code for J&J dataset 96
host code 96
kernel code 106
C++ FPGA code for NAB dataset 112
host code 112
kernel code 121

References 128

Εκτενής Περίληψη

Εισαγωγή - Κίνητρο

Η ανίχνευση ανωμαλιών αποτελεί αντικείμενο έρευνας για μεγάλο χρονικό διάστημα. Σε έναν κόσμο ψηφιοποίησης, ο όγκος των δεδομένων που μεταφέρονται υπερβαίνει την ανθρώπινη ικανότητα να τα μελετήσει χειροκίνητα. Ως εκ τούτου, η αυτοματοποιημένη ανάλυση δεδομένων καθίσταται αναγκαία. Μία από τις πιο σημαντικές εργασίες ανάλυσης δεδομένων είναι ο εντοπισμός ανωμαλιών στα δεδομένα. Οι ανωμαλίες είναι σημεία δεδομένων που αποκλίνουν από την κανονική κατανομή ολόκληρου του συνόλου δεδομένων και η ανίχνευση ανωμαλιών είναι η τεχνική για την εύρεση τους [1]. Επομένως, η ανίχνευση ανωμαλιών είναι η αναγνώριση σπάνιων γεγονότων, στοιχείων ή παρατηρήσεων που είναι ύποπτα επειδή διαφέρουν σημαντικά από τις τυπικές συμπεριφορές ή πρότυπα. Οι ανωμαλίες στα δεδομένα ονομάζονται επίσης τυπικές αποκλίσεις, ακραίες τιμές, θόρυβος, καινοτομίες και εξαιρέσεις [2]. Οι μέθοδοι ανίχνευσης ανωμαλιών είναι συγκεκριμένες για τον τύπο των δεδομένων. Για παράδειγμα, οι αλγόριθμοι που χρησιμοποιούνται για την ανίχνευση ανωμαλιών στις εικόνες είναι διαφορετικοί από τις προσεγγίσεις που χρησιμοποιούνται στις ροές δεδομένων [1]. Εδώ εστιάζουμε στην ανίχνευση ανωμαλιών σε δεδομένα χρονοσειρών.

Οι ανωμαλίες μπορούν να ταξινομηθούν γενικά με διάφορους τρόπους, μερικοί από τους οποίους είναι [2]:

- Ανωμαλίες δικτύου: Οι ανωμαλίες στη συμπεριφορά του δικτύου αποκλίνουν από αυτό που είναι φυσιολογικό, τυπικό ή αναμενόμενο. Για τον εντοπισμό ανωμαλιών δικτύου, οι διαχειριστές του δικτύου πρέπει να έχουν μια ιδέα της αναμενόμενης ή κανονικής συμπεριφοράς. Η ανίχνευση ανωμαλιών στη συμπεριφορά του δικτύου απαιτεί τη συνεχή παρακολούθηση ενός δικτύου για απροσδόκητες τάσεις ή γεγονότα.
- Ανωμαλίες απόδοσης εφαρμογής: Πρόκειται απλώς για ανωμαλίες που εντοπίζονται από την παρακολούθηση της απόδοσης της εφαρμογής από άκρο σε άκρο. Αυτά τα συστήματα παρατηρούν τη λειτουργία της εφαρμογής, συλλέγοντας δεδομένα για όλα τα προβλήματα, συμπεριλαμβανομένων των υποδομών υποστήριξης και των εξαρτήσεων εφαρμογών. Όταν εντοπίζονται ανωμαλίες, ενεργοποιείται ο περιορισμός ρυθμού λειτουργίας και οι διαχειριστές ειδοποιούνται για την πηγή του προβλήματος με τα προβληματικά δεδομένα.
- Ανωμαλίες ασφαλείας εφαρμογών ιστού: Αυτές περιλαμβάνουν οποιαδήποτε άλλη ύποπτη συμπεριφορά εφαρμογής ιστού που μπορεί να επηρεάσει την ασφάλεια, όπως επιθέσεις CSS ή επιθέσεις DDOS.

Είναι σημαντικό για τους διαχειριστές του δικτύου να είναι σε θέση να αναγνωρίζουν και να αντιδρούν στις μεταβαλλόμενες συνθήκες λειτουργίας. Οποιεσδήποτε μεταβολές στις συνθήκες λειτουργίας των κέντρων δεδομένων ή των εφαρμογών cloud μπορούν να σηματοδοτήσουν μη αποδεκτά επίπεδα επιχειρηματικού κινδύνου. Από την άλλη πλευρά, ορισμένες αποκλίσεις μπορεί να υποδηλώνουν θετική ανάπτυξη. Επομένως, ο

εντοπισμός ανωμαλιών είναι κεντρικός για την εξαγωγή βασικών επιχειρηματικών πληροφοριών και τη διατήρηση των βασικών λειτουργιών [2]. Την τελευταία δεκαετία, οι προσεγγίσεις βαθιάς μάθησης σημείωσαν τεράστια πρόοδο στις εργασίες όρασης υπολογιστή. Αυτή η επιτυχία παρακίνησε τους ερευνητές να αξιοποιήσουν αυτές τις μεθόδους για τον εντοπισμό ανωμαλιών. Διάφορες προσεγγίσεις βαθιάς μάθησης όπως τα πολυεπίπεδα Perceptrons (MLPs), τα συνελικτικά νευρωνικά δίκτυα (CNNs) και τα δίκτυα μακράς βραχείας μνήμης (LSTMs) προτάθηκαν ως τεχνικές ανίχνευσης ανωμαλιών. Είναι παρόμοιες με τις κλασικές προσεγγίσεις μηχανικής μάθησης όσον αφορά το γεγονός ότι δεν προϋποθέτουν καμία γνώση της υποκείμενης διαδικασίας παραγωγής δεδομένων. Η δημοτικότητά τους βασίζεται στα εμπειρικά τους αποτελέσματα [1].

Τα αναδρομικά νευρωνικά δίκτυα (RNNs), ένας πολύ γνωστός αλγόριθμος βαθιάς μάθησης, έχουν εφαρμοστεί εκτενώς σε διάφορες εφαρμογές. Εκμεταλλευόμενα τις προηγούμενες εξόδους ως εισόδους για την τρέχουσα πρόβλεψη, τα RNNs δείχνουν μια ισχυρή ικανότητα εκμάθησης και πρόβλεψης ακολουθιακών δεδομένων. Για περαιτέρω βελτίωση της ακρίβειας πρόβλεψης των RNN, τα δίκτυα μακράς βραχείας μνήμης (LSTMs), ένας ελεγκτής μνήμης με ικανότητα μάθησης, συνδυάζεται με τυπικά σχέδια RNN. Τα τελευταία χρόνια, η έρευνα για τα LSTM-RNN έχει αναπτυχθεί πολύ γρήγορα λόγω της ταχείας ανάπτυξης σύγχρονων εφαρμογών που βασίζονται σε αλγόριθμους βαθιάς μάθησης. Αν και ο συνδυασμός LSTM και τυπικών RNN βελτιώνει την ακρίβεια πρόβλεψης, καθιστά επίσης το μοτίβο υπολογισμού και το πρότυπο πρόσβασης δεδομένων πιο πολύπλοκα. Λόγω της αναδρομικής φύσης των LSTM-RNN, είναι αρκετά δύσκολο για τις CPU να πραγματοποιήσουν τον υπολογισμό LSTM-RNN παράλληλα. Οι GPU μπορούν να εξερευνήσουν λίγο παραλληλισμό λόγω των λειτουργιών διακλάδωσης και του σχετικά μικρού μεγέθους μοντέλου των LSTM-RNN. Η απογοητευτική απόδοση του LSTM-RNN σε επεξεργαστές γενικής χρήσης δεν μπορεί να καλύψει τις απαιτήσεις συμπερασμού σε πραγματικό χρόνο στις σύγχρονες εφαρμογές. Σαν αποτέλεσμα, ένας επιταχυντής υψηλής απόδοσης είναι ιδιαίτερα επιθυμητός. Λαμβάνοντας υπόψη την απόδοση, την ενεργειακή απόδοση και την ευελιξία, ένας επιταχυντής που βασίζεται σε FPGA είναι μια καλή επιλογή. Τυπικά στην πράξη, ένα μοντέλο LSTM-RNN πρέπει να εκπαιδεύεται off-line μέχρι να πετύχει μια αρκετά καλή ακρίβεια πρόβλεψης και, στη συνέχεια, μπορεί να εφαρμοστεί σε διάφορες εφαρμογές της πραγματικής ζωής. Ως αποτέλεσμα, η ταχύτητα επεξεργασίας του on-line συμπερασμού είναι το βασικό σημείο της ανάπτυξης LSTM-RNN, επομένως εστιάζουμε στην επιτάχυνση της φάσης συμπερασμού των LSTM-RNN [3].

Θεωρητικό Υπόβαθρο

Μηχανική μάθηση

Η Μηχανική Μάθηση (Machine Learning) είναι ένας κλάδος της Τεχνητής Νοημοσύνης που επιτρέπει στα συστήματα να μαθαίνουν και να βελτιώνονται χωρίς να προγραμματίζονται ρητά [4]. Επικεντρώνεται στη χρήση δεδομένων και αλγορίθμων από προγράμματα για να μιμηθούν τον τρόπο που μαθαίνουν οι άνθρωποι και να κάνουν προβλέψεις για κάτι στον κόσμο, βελτιώνοντας σταδιακά την ακρίβειά τους [5]. Ο κύριος

στόχος είναι η γενίκευση από την αποκτηθείσα εμπειρία και η όσο το δυνατόν ακριβέστερη επίδοση σε εργασίες που δεν είχαν δοθεί προηγουμένως.

Τεχνητά νευρωνικά δίκτυα

Τα τεχνητά νευρωνικά δίκτυα ή απλά νευρωνικά δίκτυα εμπνέονται από τον τρόπο λειτουργίας των νευρώνων στον ανθρώπινο εγκέφαλο. Ο εγκέφαλός μας είναι ένας εξαιρετικά περίπλοκος, μη γραμμικός, παράλληλος υπολογιστής (σύστημα επεξεργασίας πληροφοριών). Τα δομικά στοιχεία του, γνωστά ως νευρώνες, είναι οργανωμένα με τέτοιο τρόπο ώστε οι ανθρώπινες λειτουργίες όπως η αντίληψη και ο έλεγχος της κίνησης, η αντιστοίχιση προτύπων και άλλες εκτελούνται πολλές φορές πιο γρήγορα από αυτό που μπορούν να πετύχουν οι πιο γρήγοροι ψηφιακοί υπολογιστές που υπάρχουν σήμερα [7].

Χαρακτηριστικό παράδειγμα είναι η αίσθηση της όρασης που είναι μια διαδικασία συλλογής πληροφοριών που επιτυγχάνεται μέσω της συνεργασίας πολλαπλών νευρώνων. Με τον καιρό, αφού συλλέξουμε οπτική εμπειρία, είμαστε σε θέση να αναγνωρίσουμε αυτό που βλέπουμε. Ομοίως, τα τεχνητά νευρωνικά δίκτυα αποτελούνται από «νευρώνες», απλές μονάδες επεξεργασίας που λαμβάνουν εισόδους, αλληλεπιδρούν μεταξύ τους και είναι ικανές να αποθηκεύουν εμπειρική γνώση [7].

Η μίμηση του ανθρώπινου εγκεφάλου επιδιώκεται με δύο τρόπους [7]:

1. Το δίκτυο λαμβάνει γνώση από το περιβάλλον του μέσω μιας διαδικασίας μάθησης.
2. Η ισχύς της σύνδεσης μεταξύ των νευρώνων, που ονομάζεται συναπτικό βάρος, χρησιμοποιείται για την αποθήκευση της αποκτηθείσας γνώσης.

Η διαδικασία εκπαίδευσης ενός τεχνητού νευρωνικού δικτύου επικεντρώνεται στην τροποποίηση των τιμών των συναπτικών βαρών χρησιμοποιώντας μια ποικιλία αλγορίθμων προκειμένου να επιτευχθεί το επιθυμητό αποτέλεσμα.

Ένας τεχνητός νευρώνας αποτελείται από τρία βασικά στοιχεία:

1. Ένα σύνολο συνάψεων η καθεμία με το δικό της βάρος. Συγκεκριμένα ένα σήμα x_j που δίνεται ως είσοδος στη σύναψη j πολλαπλασιάζεται με το βάρος της w_j. Τα συναπτικά βάρη μπορούν να λάβουν τόσο θετικές όσο και αρνητικές τιμές.
2. Έναν αθροιστή για τον υπολογισμό του αθροίσματος των σημάτων εισόδου πολλαπλασιασμένο με το αντίστοιχο βάρος. Αυτή η λειτουργία περιγράφει έναν γραμμικό συνδυαστή.
3. Μια συνάρτηση ενεργοποίησης για τον περιορισμό του σήματος εξόδου σε ένα συγκεκριμένο διάστημα. Είναι σύνηθες αυτό το διάστημα να είναι [-1, 1] ή [0, 1].

Ένα άλλο χαρακτηριστικό είναι η πόλωση που προστίθεται στην έξοδο του αθροιστή και έχει σκοπό να την προσαρμόσει στη συνάρτηση ενεργοποίησης.

Με μαθηματικούς όρους μπορούμε να ορίσουμε έναν νευρώνα χρησιμοποιώντας τις ακόλουθες εξισώσεις:

$$u = \sum_{i=1}^{m} w_i x_i$$

και

$$y = \varphi(u + b)$$

Στις παραπάνω εξισώσεις τα x_1, x_2, …, x_m αναπαριστούν τα σήματα εισόδου και τα w_1, w_2, …, w_m αναπαριστούν τα αντίστοιχα συναπτικά βάρη. Η έξοδος του γραμμικού συνδυαστή είναι u, η πόλωση είναι b, η συνάρτηση ενεργοποίησης είναι φ και τέλος το σήμα εξόδου του νευρώνα είναι y.

Μερικές συχνά χρησιμοποιούμενες συναρτήσεις ενεργοποίησης είναι:

1. Διορθωμένη Γραμμική Μονάδα (Rectified Linear Unit - ReLU)
$$f(x) = max(0, x)$$
2. Σιγμοειδής
$$\sigma(x) = \frac{1}{1+e^{-x}}$$
3. Υπερβολική εφαπτομένη
$$tanh(x) = 2\sigma(2x) - 1 = \frac{e^{2x}-1}{e^{2x}+1}$$

Δίκτυα μακράς βραχείας μνήμης

Τα αναδρομικά νευρωνικά δίκτυα (RNN) χρησιμοποιούν τις συνδέσεις ανάδρασής τους για να αποθηκεύουν αναπαραστάσεις πρόσφατων γεγονότων εισόδου με τη μορφή της εξόδου της συνάρτησης ενεργοποίησης. Αυτό το χαρακτηριστικό τους παρέχει βραχυπρόθεσμη μνήμη σε αντίθεση με τη μακροπρόθεσμη μνήμη που παρέχεται από τα αργά μεταβαλλόμενα βάρη τους [10]. Ένα αναδρομικό νευρωνικό δίκτυο μπορεί να θεωρηθεί ως μια σειρά από αντίγραφα του ίδιου δικτύου με το καθένα να περνά την έξοδο του στο επόμενο. Η αλυσιδωτή φύση τους αποκαλύπτει ότι σχετίζονται στενά με λίστες και πράγματι έχουν υπάρξει επιτυχημένες εφαρμογές των RNN σε τομείς όπως η αναγνώριση ομιλίας, η μοντελοποίηση γλώσσας και άλλους. Παρά αυτές τις προόδους, τα παραδοσιακά RNN έχει αποδειχθεί ότι μπορούν να μάθουν από προηγούμενες πληροφορίες μόνο σε περιπτώσεις όπου το χάσμα μεταξύ των σχετικών πληροφοριών και του χρόνου που είναι αναγκαίες είναι μικρό [9].

Τα δίκτυα μακράς βραχείας μνήμης (LSTM) είναι ένα ειδικό είδος RNN που δεν υποφέρει από το μειονέκτημα των τυπικών RNN. Μπορούν να μάθουν από τις μακροπρόθεσμες εξαρτήσεις, καθώς σχεδιάστηκαν ακριβώς για να αντιμετωπίσουν αυτό το πρόβλημα. Αυτό επιτυγχάνεται με έναν αποτελεσματικό αλγόριθμο για μια αρχιτεκτονική που επιβάλλει σταθερή ροή σφάλματος μέσω των εσωτερικών καταστάσεων των ειδικών μονάδων [10].

Το κλειδί για τη λειτουργία ενός δικτύου LSTM είναι η κατάσταση του κελιού (cell state) η οποία διατρέχει ολόκληρη την αλυσίδα με μικρές μόνο αλληλεπιδράσεις, διασφαλίζοντας

την ανεμπόδιστη ροή πληροφοριών επιτρέποντας έτσι στο δίκτυο να φέρει μακροπρόθεσμες εξαρτήσεις.

Η προσθήκη ή η αφαίρεση πληροφοριών από την κατάσταση του κελιού ρυθμίζεται προσεκτικά από δομές που ονομάζονται πύλες. Αποτελούνται από ένα νευρωνικό δίκτυο ενός επιπέδου που λαμβάνει τις τιμές της προηγούμενης κρυφής κατάστασης και την τρέχουσα είσοδο συνενωμένη και έχει ως συνάρτηση ενεργοποίησης τη σιγμοειδή.

Πρώτα έχουμε την πύλη επιλεκτικής συγκράτησης (forget gate). Χρησιμοποιείται για να αποφασίσει ποιες τιμές της κατάστασης κελιού θα πρέπει να διατηρηθούν. Η έξοδός της είναι ένας αριθμός μεταξύ 0 και 1 (λόγω της σιγμοειδούς συνάρτησης ενεργοποίησης) για κάθε στοιχείο της κατάστασης κελιού με το 0 να σημαίνει την πλήρη απόρριψη αυτού του στοιχείου και το 1 για την πλήρη διατήρησή του.

Το επόμενο βήμα είναι να αποφασιστεί ποιες νέες πληροφορίες θα αποθηκευτούν στην κατάσταση κελιού. Αυτό γίνεται σε δύο μέρη. Αρχικά η πύλη εισόδου αποφασίζει ποιες τιμές θα ενημερωθούν. Στη συνέχεια, ένα επίπεδο υπερβολικής εφαπτομένης παράγει ένα διάνυσμα υποψήφιων τιμών που πρέπει να προστεθούν στην κατάσταση. Τέλος, αυτά τα δύο σύνολα τιμών συνδυάζονται για να δημιουργήσουν μια ενημέρωση για την κατάσταση. Σε αυτό το σημείο μπορεί να ενημερωθεί η προηγούμενη κατάσταση κελιού στη νέα. Η παλιά κατάσταση πολλαπλασιάζεται με την έξοδο της πύλης επιλεκτικής συγκράτησης, «ξεχνώντας» έτσι άχρηστες πληροφορίες. Στη συνέχεια προστίθενται το γινόμενο της πύλης εισόδου και του επιπέδου υπερβολικής εφαπτομένης που δείχνουν ποια ενημέρωση πρέπει να γίνει σε κάθε τιμή της κατάστασης.

Τέλος, το μόνο που απομένει είναι ο υπολογισμός της εξόδου, η οποία είναι μια φιλτραρισμένη έκδοση της κατάστασης κελιού. Η πύλη εξόδου αποφασίζει ποια μέρη της κατάστασης θα διατηρηθούν και η τελική έξοδος υπολογίζεται ως το γινόμενο της πύλης εξόδου και της κατάστασης κελιού αφού περάσει από ένα επίπεδο υπερβολικής εφαπτομένης για να περιοριστούν οι τιμές μεταξύ -1 και 1.

Για να συνοψίσουμε χρησιμοποιώντας μαθηματικούς όρους, η έξοδος (h_t) και η κατάσταση κελιού (C_t) ενός δικτύου LSTM υπολογίζονται χρησιμοποιώντας τις ακόλουθες εξισώσεις:

$$f_t = \sigma\left(W_f[h_{t-1}, x_t] + b_f\right)$$

$$i_t = \sigma\left(W_i[h_{t-1}, x_t] + b_i\right)$$

$$c_t = tanh\left(W_c[h_{t-1}, x_t] + b_c\right)$$

$$o_t = \sigma\left(W_o[h_{t-1}, x_t] + b_o\right)$$

$$C_t = f_t C_{t-1} + i_t c_t$$

$$h_t = o_t tanh\left(C_t\right)$$

όπου:

- t το χρονικό βήμα
- h το διάνυσμα κρυφής κατάστασης

- x το διάνυσμα εισόδου
- [h, x] η συνένωση της προηγούμενης κρυφής κατάστασης και της τρέχουσας εισόδου
- W ο πίνακας των συναπτικών βαρών για κάθε εσωτερικό δίκτυο ενός επιπέδου
- b το διάνυσμα πόλωσης για κάθε εσωτερικό δίκτυο ενός επιπέδου
- f, i, c, o οι έξοδοι κάθε δικτύου ενός επιπέδου

Συστοιχία επιτόπια προγραμματιζόμενων πυλών

Μια συστοιχία επιτόπια προγραμματιζόμενων πυλών (FPGA) είναι ένα ψηφιακό ολοκληρωμένο κύκλωμα (IC) που περιέχει διαμορφώσιμα (προγραμματιζόμενα) μπλοκ λογικής μαζί με διαμορφώσιμες διασυνδέσεις μεταξύ αυτών των μπλοκ [12], εξ ου και το όνομα προγραμματιζόμενων πυλών. Οι σύγχρονες συσκευές FPGA αποτελούνται από έως και δύο εκατομμύρια λογικά κελιά που μπορούν να ρυθμιστούν ώστε να υλοποιούν μια ποικιλία αλγορίθμων λογισμικού. Αν και η παραδοσιακή ροή σχεδίασης σε FPGA μοιάζει περισσότερο με ένα κανονικό ολοκληρωμένο κύκλωμα παρά με έναν επεξεργαστή, ένα FPGA παρέχει σημαντικά πλεονεκτήματα κόστους σε σύγκριση με μια προσπάθεια ανάπτυξης ολοκληρωμένου κυκλώματος και προσφέρει το ίδιο επίπεδο απόδοσης στις περισσότερες περιπτώσεις. Ένα άλλο πλεονέκτημα του FPGA σε σύγκριση με το ολοκληρωμένο κύκλωμα είναι η ικανότητά του να αναδιαμορφώνεται δυναμικά. Αυτή η διαδικασία, η οποία είναι ίδια με τη φόρτωση ενός προγράμματος σε έναν επεξεργαστή, μπορεί να επηρεάσει μέρος ή το σύνολο των πόρων που είναι διαθέσιμοι στο FPGA [13].

Τα FPGA είναι κατάλληλα για διαφορετικές εφαρμογές όπως επεξεργασία ήχου και βίντεο, κρυπτογραφία, επεξεργασία σήματος, επεξεργασία εικόνας, δημιουργία τυχαίων αριθμών και διάφορες εφαρμογές αλγορίθμων. Αναμένεται ότι θα χρησιμοποιηθούν σε διάφορους τομείς όπως του πετρελαίου και του φυσικού αερίου, τα χρηματοοικονομικά και πολλούς άλλους. Οι τρέχουσες περιπτώσεις χρήσης περιλαμβάνουν, αλλά δεν περιορίζονται σε, διακομιστές cloud, τεχνητή νοημοσύνη, διαστημική τεχνολογία, αμυντικά συστήματα, συστήματα ανανεώσιμων πηγών ενέργειας, μετάφραση, τιμολόγηση παραγώγων και φυσική σωματιδίων [14].

Σύνθεση υψηλού επιπέδου

Ο σχεδιασμός μιας εφαρμογής που στοχεύει ένα FPGA ήταν στο παρελθόν μια δύσκολη διαδικασία, κυρίως λόγω των διαθέσιμων εργαλείων χαμηλού επιπέδου. Παραδοσιακά ο προγραμματισμός και η διαμόρφωση ενός FPGA επιτυγχανόταν μέσω της χρήσης μιας γλώσσας περιγραφής υλικού (HDL) όπως η Verilog ή η VHDL, μια διαδικασία παρόμοια με αυτή που χρησιμοποιείται για ένα ολοκληρωμένο κύκλωμα ειδικής εφαρμογής (ASIC) [19]. Η ολοκλήρωση αυτής της εργασίας ήταν συχνά αρκετά απαιτητική, για παράδειγμα στον εντοπισμό σφαλμάτων του κώδικα, σε σύγκριση με άλλες πλατφόρμες όπως μια CPU όπου χρησιμοποιήθηκαν γλώσσες υψηλού επιπέδου. Για να ξεπεραστεί αυτό το εμπόδιο τα τελευταία χρόνια έχουν αναπτυχθεί νέες, πιο προηγμένες μέθοδοι και έχουν δημιουργηθεί εργαλεία για σύνθεση υψηλού επιπέδου (HLS) που μετατρέπουν τον

κώδικα μιας γλώσσας υψηλού επιπέδου όπως η C/C++ σε μια γλώσσα περιγραφής υλικού.

Η σύνθεση υψηλού επιπέδου γεφυρώνει τους τομείς υλικού και λογισμικού, παρέχοντας τα ακόλουθα οφέλη [20]:

- Βελτιωμένη παραγωγικότητα για σχεδιαστές υλικού: οι σχεδιαστές υλικού μπορούν να εργαστούν σε υψηλότερο επίπεδο αφαίρεσης, ενώ δημιουργούν υλικό υψηλής απόδοσης.
- Βελτιωμένη απόδοση συστήματος για σχεδιαστές λογισμικού: οι προγραμματιστές λογισμικού μπορούν να επιταχύνουν τα υπολογιστικά εντατικά τμήματα των αλγορίθμων τους σε έναν νέο στόχο μεταγλώττισης, το FPGA.

Αυτά επιτυγχάνονται μέσω χαρακτηριστικών του HLS όπως[20]:

- Ανάπτυξη αλγορίθμων σε γλώσσα C, μειώνοντας τον χρόνο ανάπτυξης.
- Επαλήθευση σε γλώσσα C. Η επικύρωση της λειτουργικής ορθότητας του σχεδιασμού και η διόρθωση σφαλμάτων είναι πολύ πιο γρήγορη σε μια γλώσσα υψηλού επιπέδου από ό,τι με τις παραδοσιακές γλώσσες περιγραφής υλικού.
- Έλεγχος της διαδικασίας σύνθεσης μέσω οδηγιών βελτιστοποίησης. Η δημιουργία συγκεκριμένων εφαρμογών υλικού υψηλής απόδοσης και επιταχυντών αλγορίθμων είναι ευκολότερη.
- Δημιουργία πολλαπλών υλοποιήσεων από τον πηγαίο κώδικα C με χρήση οδηγιών βελτιστοποίησης για εξερεύνηση του χώρου σχεδιασμού. Αυτό επιτρέπει στον σχεδιαστή να εξετάσει τις ανταλλαγές μεταξύ των πόρων, της ταχύτητας, της κατανάλωσης ενέργειας και του μεγέθους του κυκλώματος, γεγονός που αυξάνει την πιθανότητα εύρεσης μιας βέλτιστης υλοποίησης ανάλογα με τις ανάγκες της εφαρμογής.
- Δημιουργία ευανάγνωστου και φορητού πηγαίου κώδικα C. Η επαναστόχευση του πηγαίου κώδικα C σε διαφορετικές συσκευές FPGA καθώς και η ενσωμάτωση του πηγαίου κώδικα C σε νέες εφαρμογές γίνεται μόνο με την αλλαγή λίγων παραμέτρων στο εργαλείο HLS.

TensorFlow και Keras

Το TensorFlow είναι μια πλατφόρμα ανοιχτού κώδικα για μηχανική μάθηση. Διαθέτει ένα ολοκληρωμένο, ευέλικτο σύστημα εργαλείων, βιβλιοθηκών και κοινοτικών πόρων που επιτρέπει στους ερευνητές να χρησιμοποιήσουν την τελευταία λέξη της τεχνολογίας στη μηχανική μάθηση και στους προγραμματιστές να δημιουργούν και να αναπτύσσουν εύκολα εφαρμογές που υποστηρίζονται από μηχανική μάθηση [24]. Πιο συγκεκριμένα, το TensorFlow είναι μια βιβλιοθήκη ανοιχτού κώδικα για τη δημιουργία μοντέλων μηχανικής μάθησης σε μεγάλη κλίμακα. Είναι μακράν η πιο δημοφιλής βιβλιοθήκη για τη δημιουργία μοντέλων βαθιάς μάθησης. Έχει επίσης την ισχυρότερη και τεράστια κοινότητα προγραμματιστών, ερευνητών και συντελεστών [25].

Το Keras είναι ένα API βαθιάς μάθησης γραμμένο σε Python, που τρέχει πάνω από την πλατφόρμα μηχανικής μάθησης TensorFlow. Αναπτύχθηκε με έμφαση στην παροχή δυνατότητας γρήγορου πειραματισμού. Παρέχει βασικές αφαιρέσεις και δομικά στοιχεία για την ανάπτυξη λύσεων μηχανικής εκμάθησης με υψηλή ταχύτητα επανάληψης. Το

Keras δίνει τη δυνατότητα στους μηχανικούς και τους ερευνητές να επωφεληθούν πλήρως από την επεκτασιμότητα και τις δυνατότητες πολλαπλών πλατφορμών του TensorFlow 2, μπορείτε να εκτελέσετε το Keras σε TPU ή σε μεγάλα συμπλέγματα GPU και μπορείτε να εξάγετε τα μοντέλα Keras για να εκτελεστούν στο πρόγραμμα περιήγησης ή σε μια κινητή συσκευή. [27]. Είναι πολύ δημοφιλές στην κοινότητα έρευνας και ανάπτυξης επειδή υποστηρίζει γρήγορο πειραματισμό, δημιουργία πρωτοτύπων και φιλικό προς τον χρήστη API.

Ανάπτυξη και επιτάχυνση LSTM σε FPGA

Μεταφορά μοντέλου python σε C++

Για να δημιουργήσουμε το μοντέλο χρησιμοποιούμε το Sequential API του TensorFlow το οποίο ομαδοποιεί μια γραμμική ακολουθία επιπέδων νευρωνικού δικτύου σε ένα μοντέλο [35], η έξοδος κάθε επιπέδου δίνεται ως είσοδος στο επόμενο. Αυτό κάνει την όλη διαδικασία πολύ εύκολη και με λίγες γραμμές κώδικα μπορούμε να ορίσουμε και να εκπαιδεύσουμε ένα πολύ περίπλοκο μοντέλο.

Το επόμενο βήμα είναι η μεταφορά του εκπαιδευμένου μοντέλου σε C++. Το πρώτο βήμα για να εκτελέσουμε το εκπαιδευμένο μας μοντέλο σε C++ είναι να κατανοήσουμε πώς λειτουργεί ένα ακολουθιακό μοντέλο. Κάθε επίπεδο είναι ανεξάρτητο από τα άλλα ως προς τους υπολογισμούς που γίνονται, η μόνη τους αλληλεπίδραση είναι ότι κάθε ένα περνάει την έξοδο του ως είσοδο στο επόμενο. Το επόμενο πράγμα που πρέπει να κάνουμε είναι να κατανοήσουμε πώς λειτουργεί κάθε διαφορετικό επίπεδο. Αναλύοντας το πρόβλημα καθαρά μαθηματικά παρατηρούμε πως έχουμε να κάνουμε κυρίως με πολλαπλασιασμούς πίνακα-διανύσματος και πράξεις διανυσμάτων. Το μόνο που απομένει είναι να εξαγάγουμε τα βάρη από το εκπαιδευμένο μοντέλο python και να τα χρησιμοποιήσουμε στον κώδικα C++. Αυτό μπορεί να γίνει εύκολα με μερικές απλές συναρτήσεις python. Ο κώδικας για την εξαγωγή των βαρών, καθώς και παραδείγματα LSTM και Dense layers, είναι διαθέσιμος στο Παράρτημα.

Δημιουργία πυρήνα για το FPGA

Χρησιμοποιούμε την ενιαία πλατφόρμα λογισμικού Vitis (Vitis Unified Software Platform) για να δημιουργήσουμε το τελικό εκτελέσιμο αρχείο για τα FPGA. Η ροή ανάπτυξης επιτάχυνσης εφαρμογών Vitis παρέχει ένα πλαίσιο για την ανάπτυξη εφαρμογών με επιτάχυνση σε FPGA χρησιμοποιώντας τυπικές γλώσσες προγραμματισμού τόσο για στοιχεία λογισμικού όσο και για στοιχεία υλικού. Το στοιχείο λογισμικού, ή το πρόγραμμα υποδοχής (host), αναπτύσσεται χρησιμοποιώντας C/C++ για εκτέλεση σε επεξεργαστές x86 ή ενσωματωμένους επεξεργαστές, με κλήσεις OpenCL API για τη διαχείριση των αλληλεπιδράσεων χρόνου εκτέλεσης με τον επιταχυντή. Το στοιχείο υλικού, ή ο πυρήνας, μπορεί να αναπτυχθεί χρησιμοποιώντας C/C++, OpenCL C ή RTL [23]. Στην περίπτωσή μας χρησιμοποιούμε C++ τόσο για το πρόγραμμα υποδοχής όσο και για τον πυρήνα.

Έχοντας γράψει πρώτα το μοντέλο μας σε C++, η ανάπτυξη ενός λειτουργικού πυρήνα χωρίς να επιτύχουμε επιτάχυνση της εφαρμογής μας, μπορεί να γίνει αρκετά εύκολα και μόνο με μερικές τροποποιήσεις στον κώδικα C++ μας. Πρώτα από όλα χωρίζουμε τον κώδικα σε δύο μέρη, τον κώδικα που εκτελείται στον επεξεργαστή και τον κώδικα πυρήνα (που εκτελείται στο FPGA, αυτόν που θέλουμε να επιταχύνουμε). Το επόμενο βήμα είναι η τροποποίηση της main συνάρτησης της C++. Προσθέτουμε μερικές εντολές OpenCL για τη διαχείριση της εκτέλεσης του πυρήνα. Συνοπτικά αυτό που πρέπει να κάνουμε είναι να βρούμε την πλατφόρμα και να φορτώσουμε το εκτελέσιμο αρχείο, να εκχωρήσουμε μνήμη στη συσκευή, να μεταφέρουμε τα δεδομένα εισόδου, να εκκινήσουμε τον πυρήνα και να ανακτήσουμε το αποτέλεσμα μόλις ολοκληρωθούν οι υπολογισμοί. Ο πυρήνας μας έχει μια συνάρτηση ανώτατου επιπέδου ορατή στον κώδικα που εκτελείται στον επεξεργαστή και αυτή είναι με την οποία αλληλεπιδρούμε. Όσο για τον κώδικα του πυρήνα μπορούμε να τον διατηρήσουμε ως έχει με ελάχιστες τροποποιήσεις. Είναι δυνατή η χρήση τύπων δεδομένων της C++ και τυπικών κεφαλίδων βιβλιοθήκης C++ και το εργαλείο θα παράγει το εκτελέσιμο αρχείο χωρίς προβλήματα. Μια σημαντική εξαίρεση είναι ότι η δυναμική εκχώρηση μνήμης δεν είναι διαθέσιμη. Αξίζει επίσης να σημειωθεί ότι κρατώντας αυτόν τον κωδικό τον ίδιο πιθανότατα δεν θα επιτύχουμε καμία επιτάχυνση. Θα καλύψουμε τις βελτιστοποιήσεις που χρησιμοποιούμε αργότερα.

Βελτιστοποιήσεις πυρήνα

Μετασχηματισμοί κώδικα

Στα μοντέλα που υλοποιήσαμε χρησιμοποιούμε δύο κελιά LSTM τα οποία στον αρχικό κώδικα C++ αντιπροσωπεύονται από διαφορετικές συναρτήσεις. Αυτό σημαίνει ότι καθεμία από αυτές τις συναρτήσεις απαιτεί ξεχωριστούς πόρους όταν εφαρμόζεται στο FPGA, παρότι εκτελούν τους ίδιους υπολογισμούς και δεν είναι δυνατή η ταυτόχρονη εκτέλεση (το δεύτερο LSTM περιμένει την έξοδο του πρώτου). Μπορούμε να εξαλείψουμε την ανάγκη για ξεχωριστές συναρτήσεις εισάγοντας μερικές απλές εντολές if-else στα σημεία που χρησιμοποιούνται τα βάρη και χειριζόμενοι σωστά το τελικό αποτέλεσμα. Ένα άλλο πράγμα που πρέπει να τονιστεί είναι ότι οι τέσσερις πολλαπλασιασμοί διανυσμάτων-πίνακα σε κάθε κελί LSTM δεν έχουν εξαρτήσεις δεδομένων και μπορούν να εκτελεστούν ταυτόχρονα. Γράφουμε ξεχωριστή συνάρτηση για κάθε μία και το εργαλείο τις υλοποιεί αυτόματα παράλληλα. Τέλος, για να εφαρμοστούν σωστά οι οδηγίες βελτιστοποίησης στον εμφωλευμένο βρόχο πολλαπλασιασμού πίνακα-διανύσματος χρησιμοποιήσαμε μια ξεχωριστή συνάρτηση για τον εσωτερικό βρόχο.

Οδηγίες βελτιστοποίησης

Το εργαλείο HLS παρέχει οδηγίες βελτιστοποίησης που μπορούν να χρησιμοποιηθούν για τη βελτιστοποίηση του σχεδιασμού, τη μείωση του χρόνου εκτέλεσης και τη μείωση της χρήσης πόρων συσκευής του προκύπτοντος κώδικα RTL. Αυτές οι οδηγίες βελτιστοποίησης μπορούν να προστεθούν απευθείας στον πηγαίο κώδικα του πυρήνα [38].

Αριθμοί σταθερής υποδιαστολής
Αρκετές μελέτες έχουν δείξει ότι ο συμπερασμός των νευρωνικών δικτύων μπορεί να επιτευχθεί με μειωμένη ακρίβεια των τελεστών. Τα βάρη κβαντίζονται χρησιμοποιώντας ένα σχήμα αναπαράστασης σταθερής υποδιαστολής. Στην απλούστερη έκδοση αυτής της μορφής, οι αριθμοί κωδικοποιούνται με το ίδιο πλάτος bit που ορίζεται σύμφωνα με το αριθμητικό εύρος και την επιθυμητή ακρίβεια. Όλοι οι τελεστές μοιράζονται τον ίδιο εκθέτη (δηλ. συντελεστή κλίμακας) που μπορεί να θεωρηθεί ως η θέση του σημείου υποδιαστολής. Σε σύγκριση με το κινητή υποδιαστολή, ο υπολογισμός με αριθμούς σταθερής υποδιαστολής με σταθερό πλάτος bit είναι γνωστό ότι είναι πιο αποτελεσματικός όσον αφορά τη χρήση του υλικού και την κατανάλωση ενέργειας [39].

Αποτελέσματα - Αξιολόγηση

Συσκευές

Alveo U280
Οι κάρτες επιτάχυνσης Alveo της Xilinx είναι κάρτες συμβατές με το PCI Express που έχουν σχεδιαστεί για την επιτάχυνση εφαρμογών με εντατικούς υπολογισμούς, όπως η μηχανική μάθηση, η ανάλυση δεδομένων και η επεξεργασία βίντεο σε διακομιστή ή σταθμό εργασίας. Στη συσκευή Xilinx, η πλατφόρμα αποτελείται από μια στατική περιοχή και μια δυναμική περιοχή. Η στατική περιοχή της πλατφόρμας παρέχει τη βασική υποδομή για την επικοινωνία της κάρτας με τον κεντρικό υπολογιστή και την υποστήριξη υλικού για τον πυρήνα. Η δυναμική περιοχή είναι το μέρος της συσκευής όπου αποθηκεύονται και εκτελούνται επιταχυνόμενοι πυρήνες.

MPSoC ZCU104
Το Zynq UltraScale+ MPSoC είναι η πλατφόρμα Zynq δεύτερης γενιάς Xilinx, που συνδυάζει ένα ισχυρό σύστημα επεξεργασίας (PS) και προγραμματιζόμενη από το χρήστη λογική (PL) στην ίδια συσκευή. Το σύστημα επεξεργασίας διαθέτει τον κορυφαίο επεξεργαστή Arm Cortex-A53 64-bit τετραπύρηνο ή διπύρηνο και διπύρηνο επεξεργαστή Cortex-R5F πραγματικού χρόνου. Αυτές οι νέες συσκευές προσφέρουν την ευελιξία και την επεκτασιμότητα ενός FPGA, ενώ παρέχουν την απόδοση, την ισχύ και την ευκολία χρήσης που συνήθως σχετίζονται με ASIC και ASSP. Η γκάμα της οικογένειας Zynq UltraScale+ δίνει τη δυνατότητα στους σχεδιαστές να στοχεύουν σε ευαίσθητες στο κόστος εφαρμογές και εφαρμογές υψηλής απόδοσης από μια ενιαία πλατφόρμα χρησιμοποιώντας εργαλεία βιομηχανικών προτύπων. Τα χαρακτηριστικά του PL διαφέρουν από τον έναν τύπο συσκευής στον άλλο. Ως αποτέλεσμα, τα Zynq UltraScale+ MPSoC μπορούν να εξυπηρετήσουν ένα ευρύ φάσμα εφαρμογών [33].

Σύνολα δεδομένων και μοντέλα που χρησιμοποιήθηκαν

Στην πρώτη μας εφαρμογή ανιχνεύουμε ανωμαλίες στα ιστορικά δεδομένα τιμών μετοχών της Johnson & Johnson (J&J). Η χρονική περίοδος που επιλέχθηκε ήταν από το 1985-09-04 έως το 2020-09-03 και αυτά τα δεδομένα είναι διαθέσιμα μέσω του Yahoo Finance [41]. Το δίκτυο που χρησιμοποιείται για αυτήν την εργασία είναι ένας autoencoder. Οι autoencoders είναι ένας συγκεκριμένος τύπος νευρωνικών δικτύων

όπου η είσοδος είναι ίδια με την έξοδο. Συμπιέζουν την είσοδο σε μια αναπαράσταση χαμηλότερης διαστατικότητας και στη συνέχεια ανασυνθέτουν την έξοδο από αυτήν την αναπαράσταση. Η αναπαράσταση αυτή είναι μια συμπαγής "σύνοψη" ή "συμπίεση" της εισόδου, που ονομάζεται επίσης αναπαράσταση λανθάνοντος χώρου. Ένας autoencoder αποτελείται από 3 στοιχεία: κωδικοποιητή, κώδικα και αποκωδικοποιητή. Ο κωδικοποιητής συμπιέζει την είσοδο και παράγει τον κώδικα, ο αποκωδικοποιητής στη συνέχεια αναδομεί την είσοδο χρησιμοποιώντας μόνο αυτόν τον κώδικα [42].

Για τη δεύτερη εφαρμογή μας χρησιμοποιήσαμε το σύνολο δεδομένων Numenta Anomaly Benchmark (NAB) που είναι δημόσια διαθέσιμο στο Kaggle. Είναι ένα σημείο αναφοράς για την αξιολόγηση αλγορίθμων για την ανίχνευση ανωμαλιών σε streaming, διαδικτυακές εφαρμογές. Αποτελείται από περισσότερα από 50 επισημασμένα αρχεία δεδομένων πραγματικού κόσμου και τεχνητών χρονοσειρών συν έναν νέο μηχανισμό βαθμολόγησης σχεδιασμένο για εφαρμογές σε πραγματικό χρόνο [44]. Αυτό το σύνολο δεδομένων περιλαμβάνει δεδομένα σε μορφή χρονοσειράς που ονομάζονται 'ambient_temperature_system_failure' σε μορφή CSV. Περιλαμβάνει δεδομένα αισθητήρα θερμοκρασίας της θερμοκρασίας περιβάλλοντος σε ένα περιβάλλον γραφείου. Αυτά είναι τα δεδομένα που χρησιμοποιήσαμε. Κατά τη διάρκεια της προεπεξεργασίας κάναμε ορισμένες προσθήκες στα χαρακτηριστικά των δεδομένων. Κάθε χρονικό βήμα περιλάμβανε μόνο τη θερμοκρασία (την οποία μετατρέψαμε από Φαρενάιτ σε Κελσίου). Προσθέσαμε τέσσερα νέα χαρακτηριστικά, την ώρα του χρονικού βήματος, εάν υπήρχε φως ημέρας εκείνη την ώρα (υποθέσαμε ότι το φως της ημέρας είναι παρόν μετά τις 7 π.μ. και πριν τις 10 μ.μ.), την ημέρα της εβδομάδας και αν ήταν καθημερινή ή σαββατοκύριακο.

Ο πλήρης κωδικός και για τα δύο μοντέλα είναι διαθέσιμος στο Παράρτημα.

Απόδοση

Μετά την εκπαίδευση των μοντέλων python και τη λήψη ικανοποιητικά χαμηλού σφάλματος, το επόμενο βήμα είναι να μεταφερθούν και τα δύο στο FPGA για να ξεκινήσει η διαδικασία επιτάχυνσης. Πρώτα από όλα χρησιμοποιήσαμε τυπικούς δεκαδικούς αριθμούς κινητής υποδιαστολής για να επαληθεύσουμε την ορθότητα του σχεδίου μας, δηλαδή ότι παράγει τα ίδια αποτελέσματα με τον κώδικα της python. Η μεταφορά από τη C++ που εκτελείται στη CPU στο FPGA απαιτεί ορισμένους μετασχηματισμούς κώδικα, επομένως είναι πιθανό να προκύψουν κάποια σφάλματα. Το επόμενο βήμα μας είναι να μετατρέψουμε την εφαρμογή μας ώστε να χρήση δεκαδικών αριθμών σταθερής υποδιαστολής. Εκτός του ότι οι πράξεις τους είναι ταχύτερες από εκείνες των αριθμών κινητής υποδιαστολής, παρόμοιες με τους ακέραιους, οι αριθμοί σταθερής υποδιαστολής απαιτούν λιγότερους πόρους επιτρέποντας έτσι περαιτέρω βελτιστοποιήσεις. Το κύριο μειονέκτημά τους είναι η απώλεια ακρίβειας, επομένως πριν αποφασίσουμε για έναν συγκεκριμένο αριθμό δεκαδικών bit είναι απαραίτητο να κάνουμε κάποιες δοκιμές για να βρούμε αυτό που ανταποκρίνεται καλύτερα στις απαιτήσεις μας. Η χρήση ενός ή δύο bit λιγότερων δεν έχει σοβαρό αντίκτυπο στην απόδοση, αλλά μειώνει τους πόρους που απαιτούνται, καθιστώντας δυνατή την πραγματοποίηση πιο επιθετικών βελτιστοποιήσεων με τη χρήση των οδηγιών βελτιστοποίησης.

Στα μοντέλα μας χρησιμοποιήσαμε δύο διαφορετικούς αριθμούς σταθερής υποδιαστολής, έναν για τα βάρη των εκπαιδευμένων μοντέλων και έναν για τους υπολογισμούς των διαφορετικών επιπέδων, για παράδειγμα έναν πολλαπλασιασμό διανύσματος-πίνακα. Χρησιμοποιούν τον ίδιο αριθμό δεκαδικών ψηφίων, αλλά τα βάρη δεν απαιτούν πολλά bit για το ακέραιο μέρος τους. Οι τιμές των βαρών είναι μικρότερες από δύο και μεγαλύτερες από μείον ένα, επομένως χρειάζονται μόνο δύο bit για το ακέραιο μέρος τους, ένα για το πρόσημο και ένα για την τιμή. Για το ακέραιο τμήμα του μεγαλύτερου αριθμού σταθερής υποδιαστολής εξετάσαμε το εύρος των τιμών που παρήχθησαν στους υπολογισμούς για το σύνολο εκπαίδευσης (training set) και υποθέσαμε ότι η χρήση αρκετών bits για την κάλυψη αυτών των αριθμών θα ήταν επαρκής, κάτι που αποδείχθηκε σωστό. Είναι προφανές ότι στην εξερεύνηση μας για τον καλύτερα ταιριαστό τύπο δεδομένων σταθερής υποδιαστολής διατηρήσαμε σταθερά τα ακέραια bit και αλλάξαμε μόνο τον αριθμό των δεκαδικών bit.

Η διαδικασία που ακολουθήσαμε είναι απλή. Κατασκευάσαμε πολλές διαφορετικές εφαρμογές με κάθε μία να έχει διαφορετικό αριθμό δεκαδικών ψηφίων. Στη συνέχεια, εκτελούμε την κάθε μια εξετάζοντας τις διαφορές μεταξύ της υλοποίησης λογισμικού σε python και της υλοποίησης υλικού και επίσης παρακολουθούμε ορισμένες μετρικές ακρίβειας. Οι μετρικές που χρησιμοποιήσαμε είναι:

$$Precision = \frac{TP}{TP+FP},\ Recall = \frac{TP}{TP+FN},\ F_measure = 2\frac{Precision*Recall}{Precision+Recall} = \frac{TP}{TP+\frac{1}{2}(FP+FN)}$$

όπου TP=True Positives, FP=False Positives, TN=True Negatives, FN=False Negatives

Δεδομένου ότι ο στόχος μας είναι να δημιουργήσουμε αποτελέσματα όσο το δυνατόν όμοια με το μοντέλο python, η σύγκριση γίνεται μεταξύ των προβλέψεων της python και εκείνων που έγιναν μετά την εκτέλεση της εφαρμογής στο FPGA. Έτσι, true positive είναι τα ανώμαλα σημεία που βρέθηκαν τόσο από το FPGA όσο και από το μοντέλο python, τα false positive αυτά που βρέθηκαν μόνο από το FPGA κ.λπ. Αυτά τα πειράματα έγιναν χρησιμοποιώντας την κάρτα επιτάχυνσης Alveo u280 λόγω του γεγονότος ότι ήταν πιο εύκολα διαθέσιμη όλη την ώρα και δεν απαιτούσε την επανεγγραφή μιας κάρτας SD και την επανεκκίνηση για την εκτέλεση της εφαρμογής. Τα αποτελέσματα αυτών των πειραμάτων συνοψίζονται στα γραφήματα στο κεφάλαιο 4.3.

Αφού μελετήσαμε τα αποτελέσματα για όλες τις διαφορετικές διαμορφώσεις, παρατηρούμε ότι από τα εννέα δεκαδικά bit και πάνω, όλες οι μετρικές σφάλματος που παρακολουθούμε γίνονται πολύ μικρές, τα αποτελέσματα είναι σχεδόν ίδια με εκείνα που έγιναν από τα αρχικά μας μοντέλα python. Το μέσο σφάλμα ανακατασκευής (για το σύνολο δεδομένων J&J) και το μέσο σφάλμα πρόβλεψης (για το σύνολο δεδομένων NAB) δεν παρουσιάζουν μεγάλη βελτίωση κατά την αύξηση του αριθμού των δεκαδικών bit και είναι σχεδόν ίσα με την υλοποίηση του λογισμικού. Επιπλέον, η τιμή των μετρικών ακρίβειας (precision, recall, F-measure) πλησιάζει το ένα. Έτσι καταλήγουμε στο συμπέρασμα ότι εννέα bit είναι αρκετά για τις ανάγκες μας και προχωράμε χρησιμοποιώντας αυτήν την υλοποίηση για περαιτέρω επιτάχυνση.

Συνεχίζουμε κάνοντας μερικούς μετασχηματισμούς κώδικα στους πυρήνες σταθερής υποδιαστολής 9-bit. Οι δύο ξεχωριστές συναρτήσεις για τα διαφορετικά κελιά LSTM

συγχωνεύονται και χρησιμοποιείται μια διακριτή συνάρτηση για κάθε πύλη (επιτρέποντας την ταυτόχρονη εκτέλεση). Η οδηγία βελτιστοποίησης array_partition εφαρμόζεται σε πίνακες όπου απαιτείται ταυτόχρονη πρόσβαση σε πολλά στοιχεία, οι βρόχοι όπου δεν υπάρχουν εξαρτήσεις δεδομένων μεταξύ των επαναλήψεων γίνονται pipelined (χρησιμοποιώντας την αντίστοιχη οδηγία βελτιστοποίησης) και ο εσωτερικός βρόχος των πολλαπλασιασμών πίνακα-διανύσματος γράφεται ως ξεχωριστή συνάρτηση έτσι ώστε ο εξωτερικός βρόχος μπορεί να γίνει pipelined και ο εσωτερικός βρόχος μπορεί να γίνει unroll με το εργαλείο Vitis.

Παρουσιάζουμε δύο ξεχωριστές εκδόσεις για τις επιταχυνόμενες εφαρμογές μας, η μία για ένα ενσωματωμένο σύστημα και η άλλη πιο κατάλληλη για διακομιστή ή κέντρο δεδομένων. Για το ενσωματωμένο σύστημα χρησιμοποιούμε την πλακέτα ZCU104. Ο επεξεργαστής ARM του είναι τυπικός εκπρόσωπος ενός επεξεργαστή ενσωματωμένου συστήματος. Το μοντέλο python μετατρέπεται σε TensorFlow Lite (το οποίο είναι ένα σύνολο εργαλείων που επιτρέπει τη μηχανική μάθηση στη συσκευή βοηθώντας τους προγραμματιστές να εκτελούν τα μοντέλα τους σε κινητές και ενσωματωμένες συσκευές) προκειμένου να εκτελεστεί στην πλακέτα. Για το κέντρο δεδομένων χρησιμοποιούμε ένα σύστημα εξοπλισμένο με δύο επεξεργαστές Intel(R) Xeon(R) Silver 4210 και την κάρτα επιτάχυνσης Alveo u280. Η τελική επιτάχυνση για τις εφαρμογές μας παρουσιάζεται παρακάτω. Μετράμε τον μέσο χρόνο εκτέλεσης για τον συμπερασμό ενός σημείου δεδομένων και εμφανίζουμε χρόνους εκτέλεσης για το μοντέλο python και τη C++ χωρίς παραλληλισμό, που εκτελούνται και οι δύο στην CPU και για τον πυρήνα FPGA. Για το ενσωματωμένο σύστημα δεν παρέχουμε έκδοση κινητής υποδιαστολής για τον πυρήνα FPGA. Στα ενσωματωμένα συστήματα οι διαθέσιμοι πόροι είναι περιορισμένοι και αν δεν εκμεταλλευτούμε τους αριθμούς σταθερής υποδιαστολής, η δυνατότητα επιτάχυνσης περιορίζεται σοβαρά.

J&J dataset
ενσωματωμένη εφαρμογή (MPSoC ZCU 104)

python	C++	FPGA kernel
22ms	63.879ms	0.477ms

Ο χρόνος εκτέλεσης για τον πυρήνα FPGA περιλαμβάνει 0,133ms για τη μεταφορά δεδομένων από την CPU στο FPGA και αντίστροφα, ο καθαρός χρόνος εκτέλεσης είναι 0,344ms.

εφαρμογή κέντρου δεδομένων (Alveo U280)

python	C++	FPGA kernel (floating point)	FPGA kernel (fixed point)
1.209ms	18.514ms	1.167ms	0.477ms

Ο χρόνος εκτέλεσης για τον πυρήνα κινητής υποδιαστολής FPGA περιλαμβάνει 0,193ms για τη μεταφορά δεδομένων από την CPU στο FPGA και αντίστροφα, ο καθαρός χρόνος εκτέλεσης είναι 0,974ms. Αντίστοιχα, ο χρόνος για τον πυρήνα σταθερής υποδιαστολής

περιλαμβάνει 0,183ms για τη μεταφορά δεδομένων από την CPU στο FPGA και αντίστροφα, ο καθαρός χρόνος εκτέλεσης είναι 0,294ms.

NAB dataset
ενσωματωμένη εφαρμογή (MPSoC ZCU 104)

python	C++	FPGA kernel
12ms	50.298ms	0.428ms

Ο χρόνος εκτέλεσης για τον πυρήνα FPGA περιλαμβάνει 0,078ms για τη μεταφορά δεδομένων από την CPU στο FPGA και αντίστροφα, ο καθαρός χρόνος εκτέλεσης είναι 0,350ms.

εφαρμογή κέντρου δεδομένων (Alveo U280)

python	C++	FPGA kernel (floating point)	FPGA kernel (fixed point)
0.658ms	13.354ms	1.018ms	0.423ms

Ο χρόνος εκτέλεσης για τον πυρήνα κινητής υποδιαστολής FPGA περιλαμβάνει 0,146ms για τη μεταφορά δεδομένων από την CPU στο FPGA και αντίστροφα, ο καθαρός χρόνος εκτέλεσης είναι 0,872ms. Αντίστοιχα, ο χρόνος για τον πυρήνα σταθερής υποδιαστολής περιλαμβάνει 0,139ms για τη μεταφορά δεδομένων από την CPU στο FPGA και αντίστροφα, ο καθαρός χρόνος εκτέλεσης είναι 0,284ms.

Εξετάζοντας τους χρόνους εκτέλεσης παραπάνω, μπορούμε να βγάλουμε κάποια συμπεράσματα. Πρώτα απ' όλα, ο πυρήνας κινητής υποδιαστολής επιτυγχάνει πολύ μικρή (J&J) έως καθόλου επιτάχυνση (NAB). Επίσης, ενώ εξετάζουμε τους χρόνους εκτέλεσης για τα μοντέλα python και C++, το μοντέλο NAB φαίνεται να είναι πολύ πιο γρήγορο, αλλά οι πυρήνες FPGA έχουν σχεδόν την ίδια καθυστέρηση. Αυτό μπορεί να εξηγηθεί από την προσέγγισή μας στη διαδικασία επιτάχυνσης. Και τα δύο μοντέλα αποτελούνται από δύο κελιά LSTM. Οι πιο χρονοβόρες λειτουργίες είναι οι τέσσερις πολλαπλασιασμοί διανυσμάτων-πίνακα κάθε κελιού. Αυτοί γίνονται παράλληλα, οπότε χρειάζεται απλώς να δούμε ένα από αυτά ως μονάδα και όχι και τους τέσσερις ως σύνολο. Στην περίπτωση του μοντέλου NAB ο πίνακας έχει διαστάσεις 100x150 και το διάνυσμα μέγεθος 150 ενώ στην περίπτωση του μοντέλου J&J τα μεγέθη είναι 128x256 και 256 αντίστοιχα. Οι πολλαπλασιασμοί γίνονται χρησιμοποιώντας έναν εμφωλευμένο βρόχο, όπου ο εσωτερικός βρόχος που εκτελεί αριθμό επαναλήψεων ίσο με τη δεύτερη διάσταση του πίνακα ξετυλίγεται (unrolls) πλήρως, οπότε χρειάζεται σχεδόν ίσος χρόνος και για τα δύο μοντέλα. Το μόνο που μένει είναι ο εξωτερικός βρόχος με έναν αριθμό επαναλήψεων ίσο με την πρώτη διάσταση του πίνακα, 100 για το NAB και 128 για το J&J. Κάθε επανάληψη και για τα δύο μοντέλα κάνει τις ίδιες λειτουργίες, επομένως χρειάζεται περίπου τον ίδιο χρόνο. Αν λάβουμε υπόψη το γεγονός ότι οι εξωτερικοί βρόχοι γίνονται pipeline με διάστημα μεταξύ επαναλήψεων (iteration interval) ενός κύκλου ρολογιού, είναι προφανές ότι οι χρόνοι εκτέλεσης για τον πολλαπλασιασμό του

πίνακα-διανύσματος δεν είναι πολύ διαφορετικοί μεταξύ των δύο μοντέλων, γεγονός που εξηγεί τους παρόμοιους συνολικούς χρόνους εκτέλεσης. Τέλος, παρατηρούμε ότι η μεταφορά δεδομένων μεταξύ της CPU και του FPGA είναι πολύ πιο γρήγορη στην πλακέτα ZCU104, κάτι που είναι λογικό αφού η CPU και το FPGA αποτελούν μέρος του ίδιου τσιπ.

Μια άλλη χρήσιμη μετρική για τη μέτρηση της απόδοσης των εφαρμογών μας είναι ο αριθμός των πράξεων ανά δευτερόλεπτο (FLOPS). Οι συνολικές πράξεις για ένα συμπερασμό του μοντέλου J&J είναι 7.861.200 πολλαπλασιασμοί, 7.875.840 προσθέσεις και 38.400 ενεργοποιήσεις ενώ για το μοντέλο NAB έχουμε 6.030.100 πολλαπλασιασμούς, 6.010.100 προσθέσεις και 50.000 ενεργοποιήσεις. Αν εξαιρέσουμε τις ενεργοποιήσεις (χρησιμοποιούμε πίνακες αναζήτησης με προϋπολογισμένες τιμές) το σύνολο είναι 15.737.040 και 12.040.200 πράξεις αντίστοιχα. Ο παρακάτω πίνακας συνοψίζει τα FLOPS για κάθε πυρήνα.

	J&J	NAB
ZCU104	32.99 GFLOPS	28.13 GFLOPS
Alveo U280	32.99 GFLOPS	28.46 GFLOPS

Τέλος, είναι χρήσιμο να αναφέρουμε τους συνολικούς πόρους που χρησιμοποιούνται από κάθε πυρήνα. Χρησιμοποιήσαμε τον ίδιο πυρήνα τόσο για το ZCU104 όσο και για το Alveo U280, επομένως αναμένεται ότι η πολύ μεγαλύτερη κάρτα επιτάχυνσης Alveo θα έχει μικρότερο ποσοστό χρησιμοποιούμενων πόρων. Οι συνολικοί πόροι βρίσκονται στις εικόνες στο τέλος του κεφαλαίου 4.3.

Συμπεράσματα

Σε αυτή τη διπλωματική εργασία παρουσιάζουμε τα αποτελέσματα της εργασίας μας σε μια προσπάθεια να επιταχύνουμε μοντέλα μηχανικής μάθησης ανίχνευσης ανωμαλιών με βάση τα LSTM σε FPGA. Η προσπάθειά μας επικεντρώθηκε στην επίτευξη όσο το δυνατόν χαμηλότερης καθυστέρησης, στοχεύοντας παράλληλα το MPSoC ZCU104 και το Alveo U280, εκπροσώπους των ενσωματωμένων εφαρμογών και των εφαρμογών κέντρων δεδομένων αντίστοιχα.

Αυτά τα μοντέλα σχεδιάστηκαν χρησιμοποιώντας τη βιβλιοθήκη TensorFlow σε python. Μετά από κατάλληλη εκπαίδευση μεταφέρθηκαν σε C++ που στη συνέχεια χρησιμοποιήθηκε για την επιτάχυνση σε FPGA με τη βοήθεια των εργαλείων για High Level Synthesis που παρέχει η Xilinx. Προκειμένου να αντιμετωπίσουμε αποτελεσματικά το πρόβλημα της επιτάχυνσης χρησιμοποιήσαμε στη συνέχεια τρεις τεχνικές, μετασχηματισμούς κώδικα, εφαρμογή οδηγιών βελτιστοποίησης και μετάβαση από αριθμούς κινητής υποδιαστολής σε αριθμούς σταθερής υποδιαστολής. Για την αξιολόγηση της εφαρμογής μας χρησιμοποιήσαμε δύο σύνολα δεδομένων, την τιμή της μετοχής της Johnson & Johnson και το Numenta Anomaly Benchmark. Τα τελικά αποτελέσματα έδειξαν ότι και τα δύο μοντέλα επιταχύνθηκαν με την προσέγγισή μας,

έχοντας ελάχιστη έως καθόλου απώλεια στην ακρίβεια, με τα δύο διαφορετικά συστήματα (ZCU105 και U280) να έχουν σχεδόν την ίδια καθυστέρηση. Η επιτάχυνση είναι περίπου 2,5 φορές για το μοντέλο J&J και 1,5 φορές για το μοντέλο NAB. Αν εστιάσουμε μόνο στην ενσωματωμένη εφαρμογή και χρησιμοποιήσουμε αυτή την καθυστέρηση ως σημείο αναφοράς, η επιτάχυνση είναι 46 και 28 φορές αντίστοιχα.

Σε μελλοντική εργασία, η εξέταση διαφορετικών στρατηγικών επιτάχυνσης μπορεί να οδηγήσει σε πιο αποτελεσματικά αποτελέσματα με δύο τρόπους. Πρώτα από όλα και τα δύο μοντέλα επιτυγχάνουν περίπου την ίδια καθυστέρηση, ενώ το NAB είναι πολύ μικρότερο όσον αφορά τις συνολικές πράξεις και είναι πολύ πιο γρήγορο σε python και C++ όταν εκτελείται στην CPU. Επομένως, μια διαφορετική προσέγγιση θα μπορούσε να διερευνηθεί όταν ασχολούμαστε με τέτοια μοντέλα, με μικρότερα κελιά LSTM. Τέλος, παρατηρούμε ότι η χρήση πόρων είναι χαμηλή στην κάρτα επιτάχυνσης Alveo U280. Η εξερεύνηση μιας διαφορετικής, πιο επιθετικής στρατηγικής επιτάχυνσης θα μπορούσε να οδηγήσει σε ακόμα καλύτερα αποτελέσματα όσον αφορά την επιτάχυνση.

Chapter 1: Introduction

1.1 Motivation

Detecting anomalies has been a research topic for a long time. In a world of digitization, the amount of data transferred exceeds the human ability to study it manually. Hence, automated data analysis becomes a necessity. One of the most important data analysis tasks is the detection of anomalies in data. Anomalies are data points which deviate from the normal distribution of the whole dataset, and anomaly detection is the technique to find them [1]. Therefore, anomaly detection is the identification of rare events, items, or observations which are suspicious because they differ significantly from standard behaviors or patterns. Anomalies in data are also called standard deviations, outliers, noise, novelties, and exceptions [2]. Anomaly detection methods are specific to the type data. For instance, the algorithms used to detect anomalies in images are different to the approaches used on data streams [1]. Here we focus on anomaly detection in time-series data.

Anomalies can classified generally in several ways, some of which are [2]:

- Network anomalies: Anomalies in network behavior deviate from what is normal, standard, or expected. To detect network anomalies, network owners must have a concept of expected or normal behavior. Detection of anomalies in network behavior demands the continuous monitoring of a network for unexpected trends or events.
- Application performance anomalies: These are simply anomalies detected by end-to-end application performance monitoring. These systems observe application function, collecting data on all problems, including supporting infrastructure and app dependencies. When anomalies are detected, rate limiting is triggered and admins are notified about the source of the issue with the problematic data.
- Web application security anomalies: These include any other anomalous or suspicious web application behavior that might impact security such as CSS attacks or DDOS attacks.

It is critical for network admins to be able to identify and react to changing operational conditions. Any nuances in the operational conditions of data centers or cloud applications can signal unacceptable levels of business risk. On the other hand, some divergences may point to positive growth. Therefore, anomaly detection is central to extracting essential business insights and maintaining core operations [2]. In the last decade, deep learning approaches achieved tremendous progress in computer vision tasks. This success motivated researchers to leverage these methods to detect anomalies. Various deep learning approaches such as Multi-Layer Perceptrons (MLPs), Convolution Neural Network (CNNs) and Long-Short Term Memory (LSTMs) were proposed as anomaly detection techniques. They are similar to classical machine learning approaches with regard to the fact that they do not presume any knowledge of

the underlying data generation process. Their popularity is based on their empirical results [1].

Recurrent neural networks (RNNs), a well-known deep learning algorithm, have been extensively applied in various applications. By taking advantage of previous outputs as inputs for current prediction, RNNs show a strong ability to learn and predict sequential data. To further improve the prediction accuracy of RNNs, Long Short-Term Memory (LSTM), a learned memory controller, is combined with standard RNN designs. In recent years, research on LSTM-RNNs has grown very fast due to the rapid development of modern applications based on deep learning algorithms. Though the combination of LSTM and standard RNNs improves the prediction accuracy, it also makes the computation pattern and data access pattern more complex. Due to the recurrent nature of LSTM-RNNs, it is quite difficult for CPUs to accomplish LSTM-RNN computation in parallel. GPUs can explore little parallelism due to the branching operations and relatively small model size of LSTM-RNNs. The disappointing performance of LSTM-RNN on general-purpose processors can not meet the requirements of real-time inference in modern applications. It means that a high-performance accelerator is highly desired. Taking performance, energy-efficiency and flexibility into consideration, an FPGA-based accelerator is a good choice. Typically in practice, an LSTM-RNN model must be trained off-line for a fairly good prediction accuracy, then it can be applied to various real-life applications. As a result, the processing speed of on-line inference is the key point of LSTM-RNN deployment, thus we focus on accelerating the inference phase of LSTM-RNNs [3].

1.2 Thesis outline

In chapter 2 we provide the required background for the reader, that is neural networks, FPGAs and some of the tools used. Chapter 3 analyzes our workflow for creating and accelerating an anomaly detection LSTM model. Our device setup as well as the experimental process and results can be found in chapter 4. Finally, chapter 5 summarizes and concludes this thesis and discusses further research directions. In the appendix we provide the code we used.

Chapter 2: Background

The purpose of this chapter is to cover the theoretical background necessary behind the topics examined. Specifically we make a brief introduction to machine learning and in particular to long short-term memory (LSTM) recurrent neural networks (RNN) and we continue with examining some basic aspects of field-programmable gate arrays (FPGA). Finally we describe some of the tools used such as the tensorflow and keras python libraries for building and training neural network models and Vitis Unified Software Platform for building the accelerated applications.

2.1 Machine Learning

Machine Learning (ML) is a branch of Artificial Intelligence which enables systems to learn and improve without being explicitly programmed [4]. It focuses on the use of data and algorithms by programs to imitate the way humans learn and to make predictions about something in the world, gradually improving accuracy [5]. The main goal is to generalize from acquired experience and perform as accurately as possible on previously unseen tasks.

This process can be broken out into three main parts [5]:

1. Decision process: Machine Learning is mostly used for making predictions or classifications. Based on some training data the algorithm infers an estimate about a pattern.
2. Error function: It is used for evaluation. If we have known examples (labeled data) this function can compare the predicted and actual result to assess the model's accuracy.
3. Optimization process: If the model can fit better to the training data, weights adjust to reduce the variation between the estimate and the already known result. This process of evaluation and optimization repeats until a desired accuracy is met or until we reach a threshold of accuracy beyond which our model cannot pass.

The type of training algorithm that data scientists choose depends on the type of data they want to predict. The training methods i.e. the way the algorithm learns to make more accurate predictions, can be classified into four broad categories [6]:

- Supervised Learning: It is defined by its use of labeled training datasets where the desired output is explicitly defined beforehand. As input data is given, the output is compared with the correct one, errors are found and the model is adjusted accordingly and after sufficient training new inputs can be evaluated.
- Unsupervised learning: It involves algorithms that train on unlabeled data. These algorithms infer a description of the input data by discovering hidden patterns or data groupings. At no point in the training process does the system know the correct output.
- Semi-supervised learning: This approach involves a medium between the two preceding categories. It uses a small labeled training dataset in order to guide classification through the discovery of data groupings in a larger, unlabeled

dataset. It can serve as the solution to the problem of not having enough labeled data needed for the training of a supervised learning algorithm.

- Reinforcement learning: It is typically used to train a system to complete a multi-step process with clearly defined rules. Similarly to supervised learning feedback is used during training, but not in the form of a labeled dataset. The learning process involves a trial and error search and receiving positive or negative cues as the algorithm tries to discover the best policy for a given problem.

2.2 Artificial Neural Networks

Artificial neural networks or simply neural networks are inspired by the way neurons in the human brain function. Our brain is an extremely complicated, non linear, parallel computer (information processing system). Its building blocks, known as neurons, are organized in such a way so that human functions like perception and control of movement, pattern matching and others are performed many times quicker than the faster digital computers existing today can achieve [7].

One typical example is the sense of sight which is an information gathering process achieved through the cooperation of multiple neurons. With time, after collecting visual experience, we are able to recognize what we are seeing. Likewise, artificial neural networks consist of “neurons”, simple processing units which receive inputs, interact with each other and are capable of storing empirical knowledge [7].

The imitation of the human brain is pursued in two ways [7]:

1. The network receives knowledge from its environment through a learning procedure.
2. The strength of the connection between neurons, called synaptic weight, is used for storing the acquired knowledge.

The training process of an artificial neural network focuses on modifying the values of the synaptic weights using a variety of algorithms in order to obtain the desired result.

2.2.1 Modeling a neuron

As stated before, the basic building block of an artificial neural network is called neuron, named after our brain cells or broadly speaking our brain’s computational units. The figure below shows a drawing of a biological neuron (left) and a common mathematical description (right). Each (biological) neuron receives input signals from its dendrites and produces output signals through its (single) axon. The axon eventually branches out and connects via synapses to dendrites of other neurons. In the computational model shown, the signals that travel along the axons (e.g. x_0) interact multiplicatively (e.g. x_0w_0) with the dendrites of the other neuron based on the synaptic strength at the synapse (e.g. w_0). The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence of one neuron on another [8].

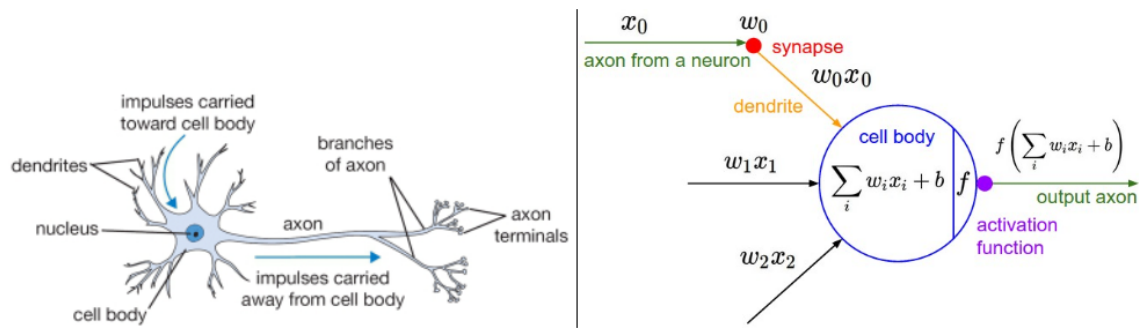

Figure 1. A drawing of a biological neuron and its mathematical model [8].

An artificial neuron is consisted of three basic elements:

1. A set of synapses each one with its own weight. Specifically a signal x_j given as input to the synapse j is multiplied with its weight w_j. Synapse weights can take both positive and negative values.
2. An adder for calculating the sum of the input signals multiplied with the corresponding weight. This operation describes a linear combiner.
3. An activation function for limiting the output signal to a specific interval. It is usual for this interval to be [-1, 1] or [0, 1].

One other feature is the bias which is added to the output of the adder and has the purpose of adjusting it to the activation function.

In mathematical terms we can define a neuron using the following equations:

$$u = \sum_{i=1}^{m} w_i x_i$$

and

$$y = \varphi(u + b)$$

In the above equations $x_1, x_2, \ldots, x_m$ represent the input signals and $w_1, w_2, \ldots, w_m$ represent the corresponding synaptic weights. The output of the linear combiner is u, the bias is b, the activation function is φ and finally the neuron's output signal is y.

Some commonly used activation functions are:

1. Rectified Linear Unit (ReLU)

$$f(x) = max(0, x)$$

2. Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

3. Hyperbolic tangent

$$tanh(x) = 2\sigma(2x) - 1 = \frac{e^{2x}-1}{e^{2x}+1}$$

2.2.2 Neural Network Architectures

The way the neurons of a network are structured is closely related with the selection of the training algorithm. Some fundamentally different categories of network architectures are the following:

1. Single-Layer Feedforward Networks: They have one input layer connected directly with a layer of output neuron (but not the other way around).
2. Multi-Layer Feedforward Networks: They include one or more hidden neuron layers without any feedback loops. These hidden layers increase the network's capacity.

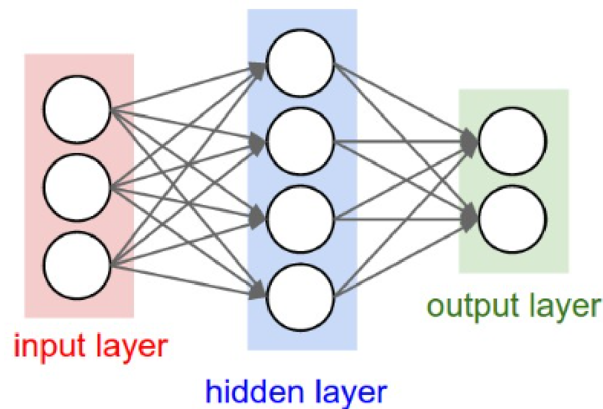

Figure 2. A Multi-Layer network with one hidden layer [8].

3. Recurrent Networks: Their difference from feedforward networks is the presence of at least one feedback loop, the connection of the output signal of a layer as an input to a previous layer. These loops greatly increase the network's learning capabilities and performance in certain tasks.

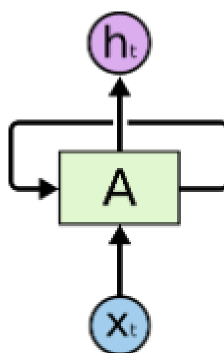

Figure 3. A Recurrent Neural Network representation [9].

2.3 Long Short-Term Memory Networks

Recurrent neural networks (RNNs) use their feedback connections to store representations of recent input events in the form of the output of the activation function. This feature provides them with short-term memory as opposed to long-term memory provided by their slowly changing weights [10].

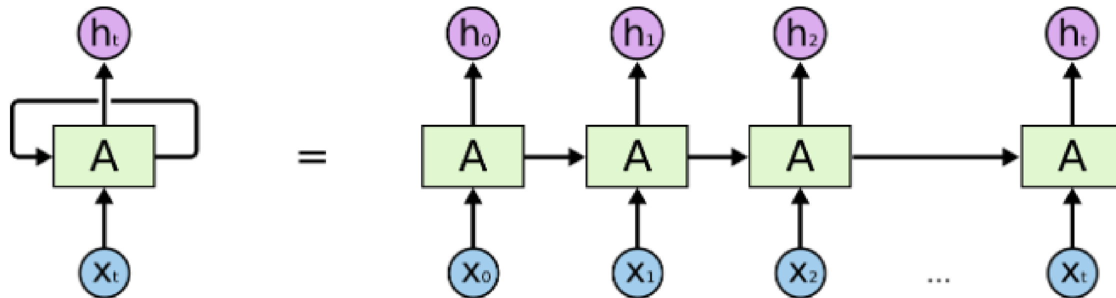

Figure 4. An unrolled recurrent neural network [9].

As shown in the figure above a recurrent neural network can be thought of as a series of copies of the same network with each one passing its output to the next one. Their chain-like nature reveals that they are closely related to lists and indeed there have been successful applications of RNNs in fields such as speech recognition, language modeling and others. Despite those breakthroughs, traditional RNNs have been shown to be able to learn from past information only in cases where the gap between relevant information and the place that it is needed is small [9].

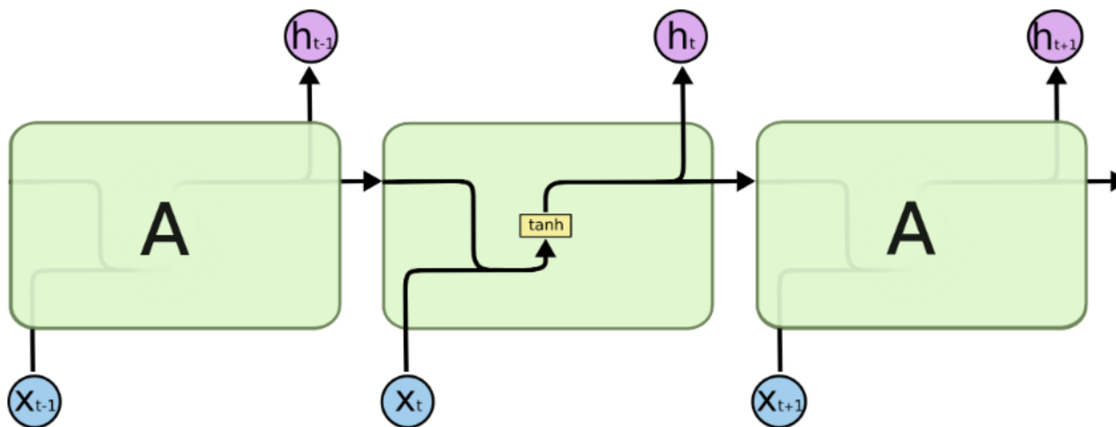

Figure 5. An unrolled standard RNN. The parallelogram represents a single-layer network using tanh as an activation function [9].

The output of a standard RNN as the one shown above is calculated by the equation:

$$h_t = tanh\left(W[h_{t-1}, x_t] + b\right)$$

where:

- t is the time step
- h is the hidden state vector
- x is the input vector

- [h, x] is the concatenation of the previous hidden state and the current input
- W is the matrix of synaptic weights
- b is the bias vector

2.3.1 Function of an LSTM cell

Long short-term memory networks (LSTMs) are a special kind of RNN not suffering from the drawback of standard RNNs. They are able to learn from long-term dependencies as they were explicitly designed to tackle this problem. This is achieved by an efficient, gradient-based algorithm for an architecture enforcing constant error flow through internal states of special units [10]. Instead of a single neural network layer there are four interacting in the way shown in the figure below.

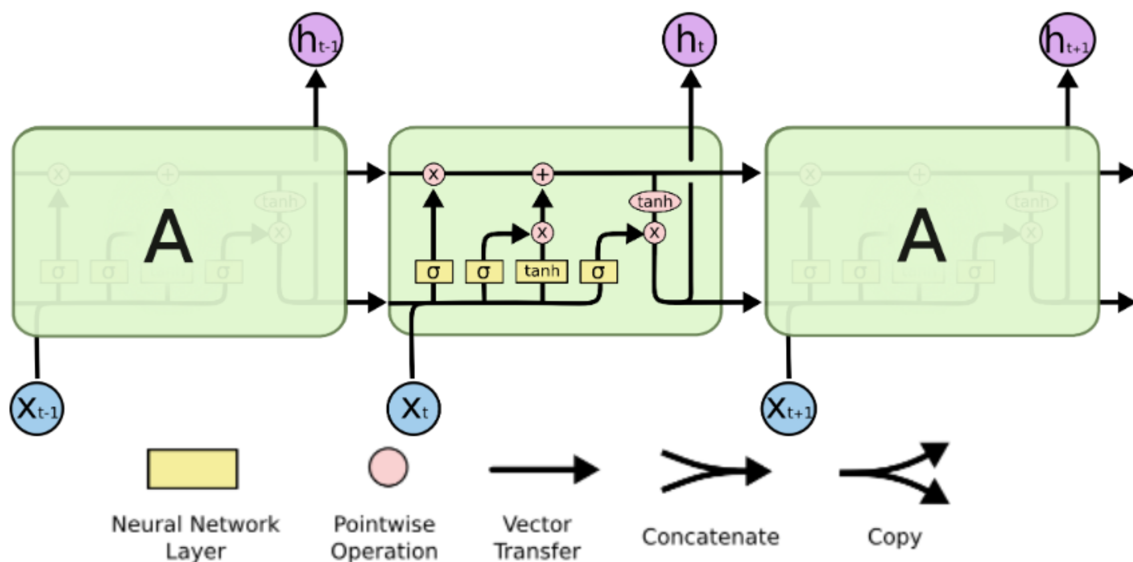

Figure 6. Structure of an LSTM network [9].

The key for the function of an LSTM network is the cell state, the horizontal line which runs through the entire chain with only minor interactions, ensuring the unhindered flow of information thus enabling the network to carry long term dependencies.

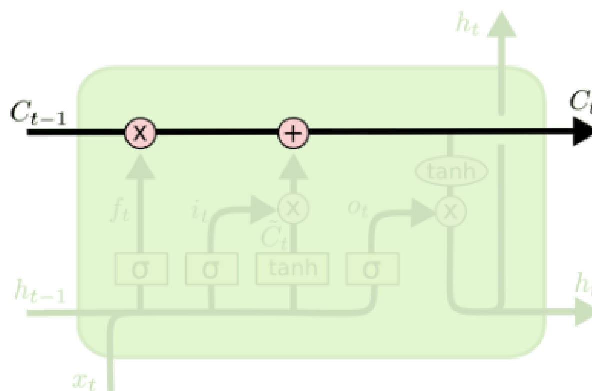

Figure 7. Cell state of LSTM [9].

The addition or removal of information from the cell state is carefully regulated by structures called gates. They are composed out of a single-layer neural network receiving the values of the previous hidden state and the current input concatenated and having sigmoid as an activation function.

First we have the “forget gate layer”. It is used to decide which values of the cell state should be kept. Its output is a number between 0 and 1 (due to the sigmoid activation function) for each element of the cell state with 0 meaning to completely discard this element and 1 to completely keep it.

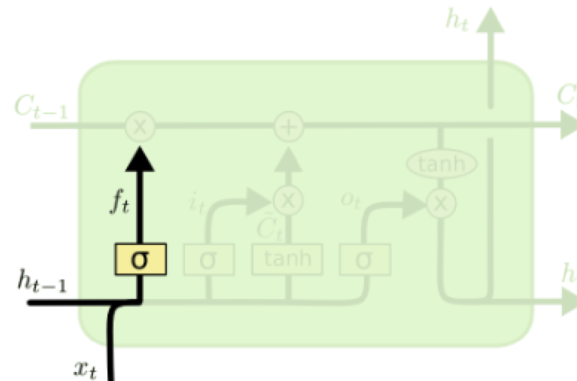

Figure 8. Forget gate [9].

The next step is to decide which new information will be stored in the cell state. This is done in two parts. Initially the “input gate layer” decides which values will be updated. Next a tanh layer produces a vector of candidate values to be added to the state. Finally these two sets of values are combined to create an update for the state.

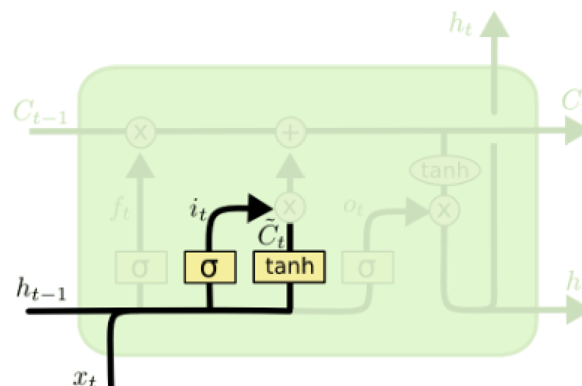

Figure 9. Input gate and tanh layer [9].

At this point we can update the previous cell state to the new one. We multiply the old state with the output of the forget gate, thus “forgetting” useless information. Then we add the product of the input gate and the tanh layer which show what update is to be done in each state value.

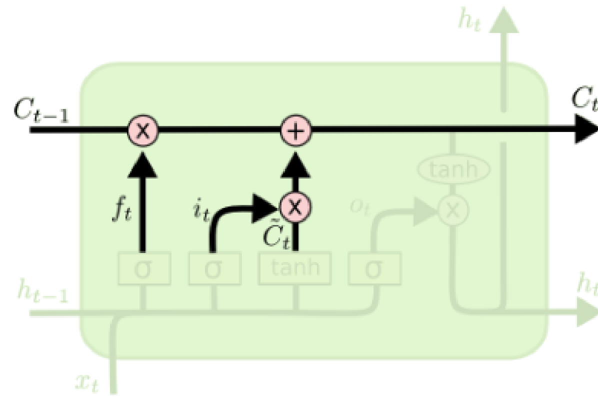

Figure 10. Cell state update [9].

Finally all that is left is the calculation of the output which will be a filtered version of the cell state. The "output gate layer" decides what parts of the state will be kept and the final output is calculated as the product of the output gate and the cell state passed through tanh to limit the values between -1 and 1.

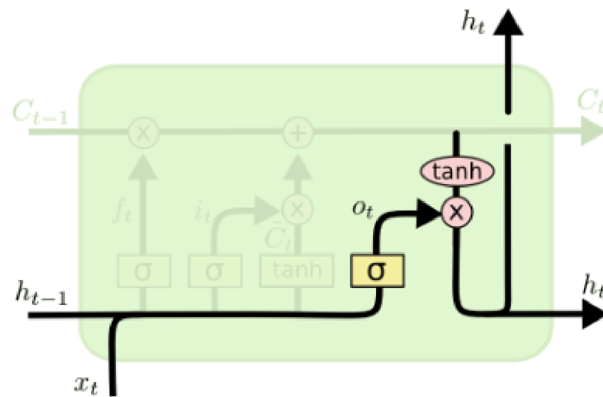

Figure 11. Output gate and output calculation [9].

To summarize using mathematical terms, the output (h_t) and cell state (C_t) of an LSTM network are calculated using the following equations:

$$f_t = \sigma\left(W_f[h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i[h_{t-1}, x_t] + b_i\right)$$
$$c_t = tanh\left(W_c[h_{t-1}, x_t] + b_c\right)$$
$$o_t = \sigma\left(W_o[h_{t-1}, x_t] + b_o\right)$$
$$C_t = f_t C_{t-1} + i_t c_t$$
$$h_t = o_t tanh\left(C_t\right)$$

where:

- t is the time step
- h is the hidden state vector
- x is the input vector
- [h, x] is the concatenation of the previous hidden state and the current input

- W is the matrix of synaptic weights for each internal single-layer network
- b is the bias vector for each internal single-layer network
- f, i, c, o are the outputs of each internal single-layer network

2.3.2 Number of parameters of an LSTM cell

After analyzing the way an LSTM cell works we continue by calculating the number of trainable parameters, that is how many values are fine tuned during training. To do this it was important to gain some insight of the way the cell handles information underneath as we will use its internal structure as a guide for our calculations. One of the reasons we attempt this is that the number of learnable parameters is an important metric for understanding the model complexity and capacity.

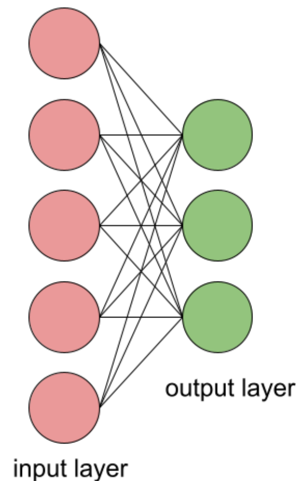

Figure 12. Single-layer feedforward network.

First of all we need to find the number of parameters in a single-layer network. These are equal with the number of connections between the two layers plus the biases for the output layer (the input layer only receives the input information and forwards it to the next layer). Putting it together we find the number of parameters to be:

$$i \cdot o + o = o(i + 1)$$

where:

- i is the size of the input vector
- o is the size of the output vector

The above formula will be used for calculating the number of parameters for each internal network of an LSTM cell.

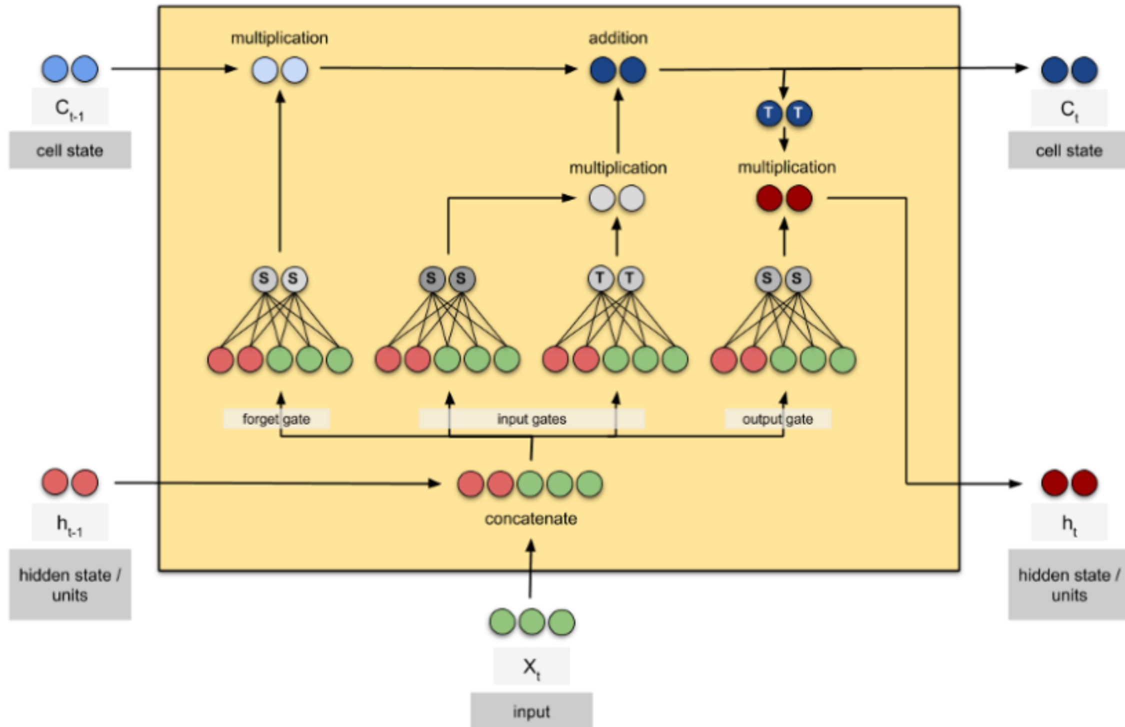

Figure 13. LSTM cell with internal structure visible [11].

As we observe in the above figure an LSTM cell receives three inputs. These are:

1. the previous timestep hidden state vector (h_{t-1})
2. the previous timestep cell state vector (C_{t-1})
3. the current timestep input vector (x_t)

It is also obvious from looking at the internal structure in the figure that the only place we can search for trainable parameters are the four internal single-layer networks. They all have the same input and output size therefore the same number of parameters. Using the formula for the number of parameters for a single-layer feedforward network calculated before, we find the total number of parameters of an LSTM cell to be:

$$4((x + h)h + h) = 4h(x + h + 1)$$

where:

- x is the size of the input vector of the LSTM cell
- h is the size of the hidden state vector, equal to the size of the cell state vector
- the size of the input vector for each internal network is x+h
- the size of the output vector for each internal network is h

2.4 Field-Programmable Gate Arrays

A field-programmable gate array (FPGA) is a digital integrated circuit (IC) that contains configurable (programmable) blocks of logic along with configurable interconnects between these blocks [12], hence the name field-programmable. Modern FPGA devices 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 advantages in comparison to an IC development effort and offers the same level of performance in most cases. Another advantage of the FPGA when compared to the IC is its ability to be dynamically reconfigured. This process, which is the same as loading a program in a processor, can affect part or all of the resources available in the FPGA fabric [13].

2.4.1 FPGA Architecture

The flexibility of the FPGA is due to its basic component which is called a configuration logic block (CLB). CLB provides the basic logic and storage capability. The configuration logic blocks are spread all over the FPGA structure [14]. These are connected through programmable interconnects and I/O pads for the transfer of data in and out of the FPGA.

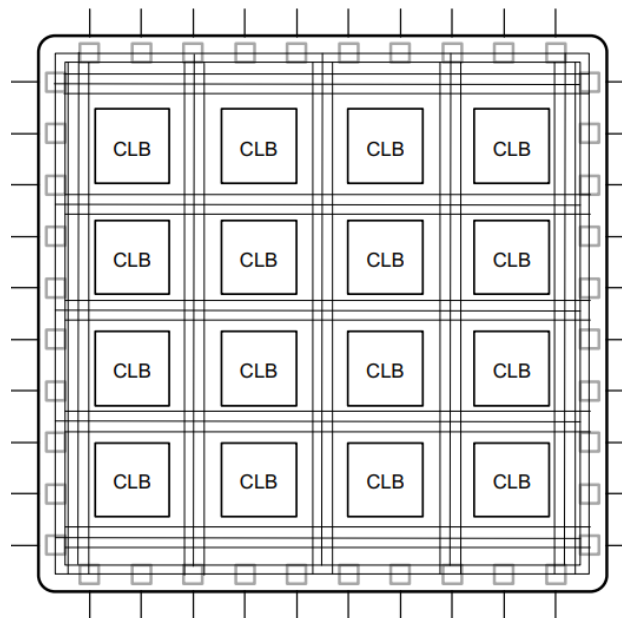

Figure 14. Basic FPGA architecture [13].

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 figure below. The LUTs in this figure are split into two 3-input LUTs. In normal mode those are combined into a 4-input LUT through the left multiplexer (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 [15].

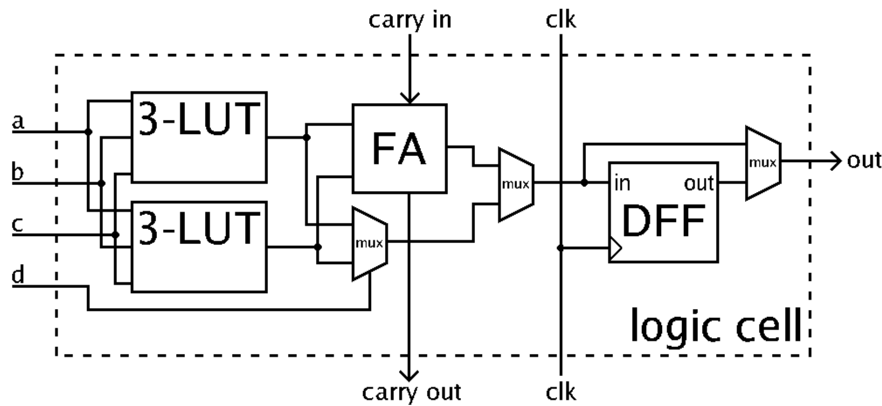

Figure 15. Simplified illustration of a logic cell [15].

2.4.2 FPGA Components

If we do not consider the CLBs as the basic building blocks and we look deeper, the basic structure of an FPGA is composed of the following elements:

- Lookup tables (LUTs): the perform logic operations
- Flip-flops (FFs): register elements that store the result of the LUT
- Wires: they connect elements to one another
- Input/Output (I/O) pads: connect the FPGA with the outside world

Contemporary FPGA architectures incorporate the basic elements along with additional computational and data storage blocks that increase the computational density and efficiency of the device [13]. Some of these additional elements are:

- Embedded memories for distributed data storage
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

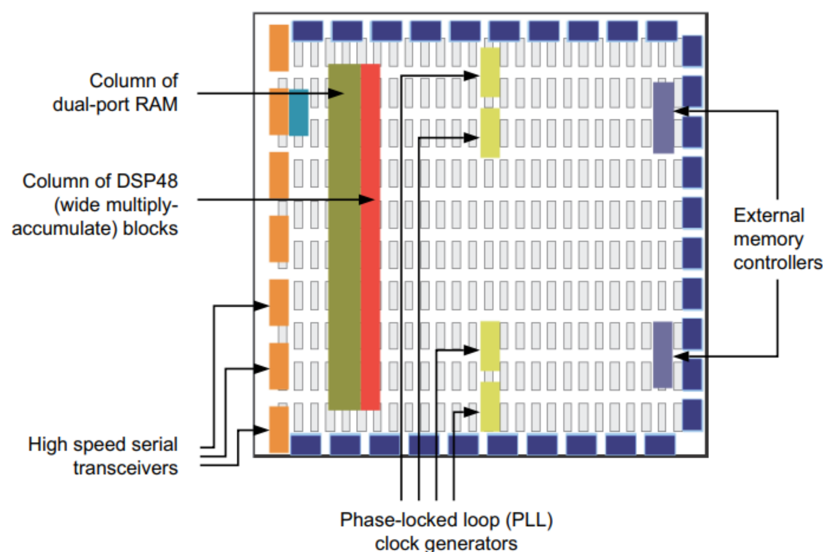

Figure 16. Contemporary FPGA Architecture [13].

Now, we are going to provide some additional information on some of the key components of an FPGA.

Lookup Table (LUT)
The LUT is the basic building block of an FPGA and is capable of implementing any logic function of N Boolean variables. Essentially, this element is a truth table in which different combinations of the inputs implement different functions to yield output values.

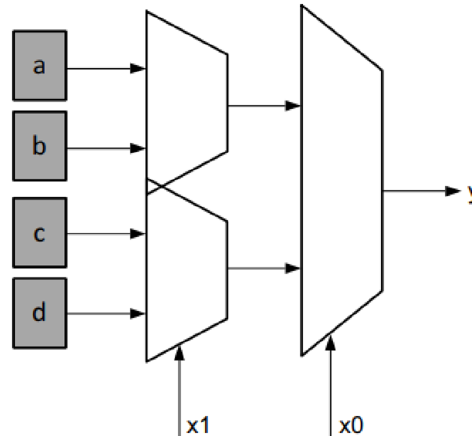

Figure 17. Functional Representation of a LUT as Collection of Memory Cells [13].

The hardware implementation of a LUT can be thought of as a collection of memory cells connected to a set of multiplexers. The inputs to the LUT act as selector bits on the multiplexer to select the result at a given point in time. An LUT can be used as both a function compute engine and a data storage element [13].

Flip-Flop (FF)
The flip-flop is the basic storage unit within the FPGA fabric. This element is always paired with a LUT to assist in logic pipelining and data storage. The basic structure of a flip-flop includes a data input, clock input, clock enable, reset, and data output. During normal operation, any value at the data input port is latched and passed to the output on every pulse of the clock. The purpose of the clock enable pin is to allow the flip-flop to hold a specific value for more than one clock pulse. New data inputs are only latched and passed to the data output port when both clock and clock enable are equal to one [13].

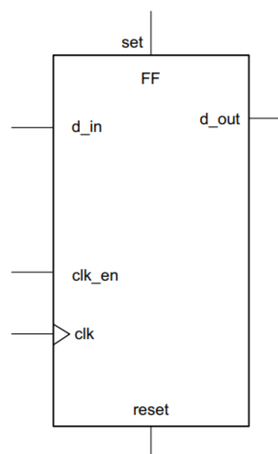

Figure 18. Structure of a Flip-Flop [13].

DSP Block

The most complex computational block available in a Xilinx FPGA is the DSP block. The DSP block is an arithmetic logic unit (ALU) embedded into the fabric of the FPGA, which is composed of a chain of three different blocks. The computational chain in the DSP is composed of an add/subtract unit connected to a multiplier connected to a final add/subtract/accumulate engine. This chain allows a single DSP unit to implement functions of the form [13]:

$$p = a(b + d) + c$$

or

$$p = a(b + d)$$

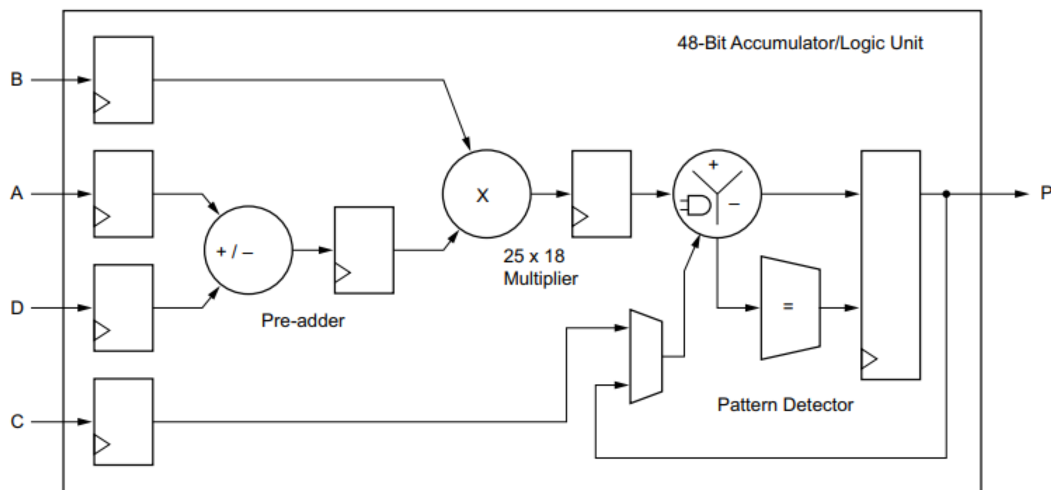

Figure 19. Structure of a DSP Block [13].

Storage Elements

The FPGA device includes embedded memory elements that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. These elements are block RAMs (BRAMs), UltraRAM blocks (URAMS), LUTs, and shift registers (SRLs) [13].

2.4.3 Comparison with other architectures

CPU

A scalar pipelined CPU core can execute instructions, divided into stages, at the rate of up to one instruction per clock cycle (IPC) when there are no dependencies. To boost performance, modern CPUs cores are multithreaded superscalar processors with sophisticated mechanisms used to find instruction-level parallelism and execute multiple out-of-order instructions per clock cycle. They fetch many instructions at once, find the dependency graph of those instructions, utilize sophisticated branch prediction mechanisms, and execute those instructions in parallel (typically at 10x the performance of scalar processors in terms of IPC). For parts of the algorithm that can be vector-parallelized, modern CPUs support single instruction, multiple data (SIMD) instructions [16].

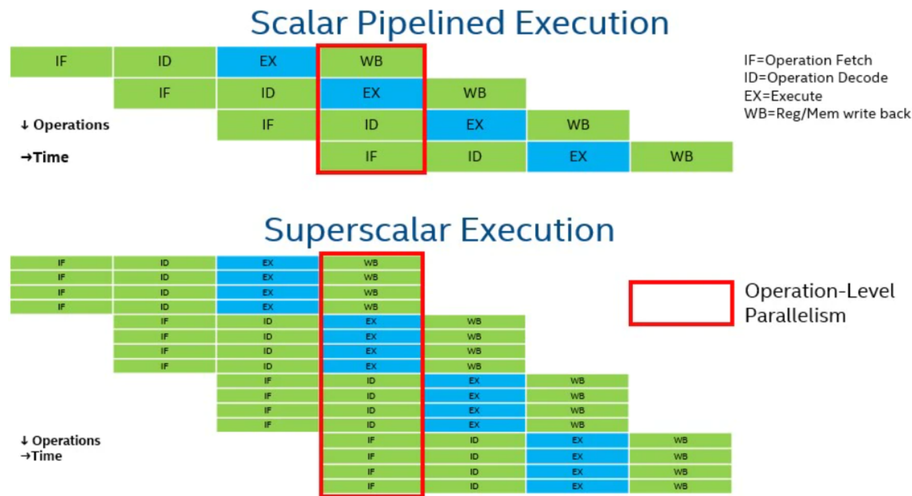

Figure 20. Simplified execution of a scalar pipelined CPU and a superscalar CPU [16].

GPU

GPUs are processors made of massively parallel, smaller, and more specialized cores than those generally found in high-performance CPUs. GPU architecture is optimized for aggregate throughput across all cores, deemphasizing individual thread latency and performance. GPU architecture efficiently processes vector data (an array of numbers) and is often referred to as vector architecture. GPUs dedicate more silicon space to compute and less to cache and control. As a result, GPU hardware explores less instruction-level parallelism and relies on software-given parallelism to achieve performance and efficiency. GPUs are in-order processors and do not support sophisticated branch prediction. Instead, they have a plethora of arithmetic logic units (ALUs) and deep pipelines. Performance is achieved through multithreaded execution of large and independent data, which amortizes the cost of simpler control and smaller caches. GPUs employ a single instruction, multiple threads (SIMT) execution model where multithreading and SIMD are leveraged together. In the SIMT model, multiple threads (work-items, or sequence of SIMD lane operations) are processed in lockstep in the same SIMD instruction stream. Multiple SIMD instruction streams are mapped to a single execution unit (EU), and the GPU EU can context switch among those SIMD instruction streams when one stream is stalled [16].

Figure 21. Each EU can process multiple SIMD instruction streams. Multiple EUs combine to form a compute unit with shared local memory and synchronization mechanisms [16].

FPGA

Unlike CPUs and GPUs, which are software-programmable fixed architectures, FPGAs are reconfigurable and their compute engines are defined by the user. When writing software targeting an FPGA, compiled instructions become hardware components that are laid out on the FPGA fabric in space, and those components can all execute in parallel. Because of this, FPGA architecture is sometimes referred to as spatial architecture. When software is “executed” on the FPGA, it is not executing in the same sense that compiled and assembled instructions execute on CPUs and GPUs. Instead, data flows through customized deep pipelines on the FPGA that match the operations expressed in the software. Because the dataflow pipeline hardware matches the software, control overhead is eliminated, which results in improved performance and efficiency. With CPUs and GPUs, instruction stages are pipelined and new instructions start executing every clock cycle. With FPGAs, operations are pipelined so new instruction streams operating on different data start executing every clock cycle [16].

Some key advantages of FPGAs include [17]:

- Excellent performance with reduced latency advantages: FPGAs provide low latency as well as deterministic latency (DL). DL as a model will continuously produce the same output from an initial state or given starting condition. The DL provides a known response time which is critical for many applications with hard deadlines. This enables faster execution of real-time applications like speech recognition, video streaming and motion recognition.
- Cost effectiveness: FPGAs can be reprogrammed after manufacturing for different data types and capabilities, delivering real value over having to replace the application with new hardware. By integrating additional capabilities, like an image processing pipeline, onto the same chip, designers can reduce costs and save board space by using the FPGA for more than just AI. The long product lifecycle of FPGAs can deliver increased utility for an application that can be measured in years or even decades. This characteristic makes them ideal for use in industrial, aerospace, defense, medical and transportation markets.
- Energy efficiency: FPGAs give designers the ability to fine-tune the hardware to the match application needs. Utilizing development tools like INT8 quantization is a successful method for optimizing machine learning frameworks like TensorFlow and PyTorch. INT8 quantization also delivers favorable results for hardware toolchains like NVIDIA® TensorRT and Xilinx® DNNDK. This is because INT8 uses 8-bit integers instead of floating-point numbers and integer math instead of floating-point math. Proper utilization of INT8 can reduce both memory and computing requirements, which can shrink memory and bandwidth usage by as much as 75%. This can prove critical in meeting power efficiency requirements in demanding applications.

In the figures following we present a performance comparison of FPGA, CPU and GPU in some image processing applications, namely two-dimensional filters, stereo-vision and k-means clustering.

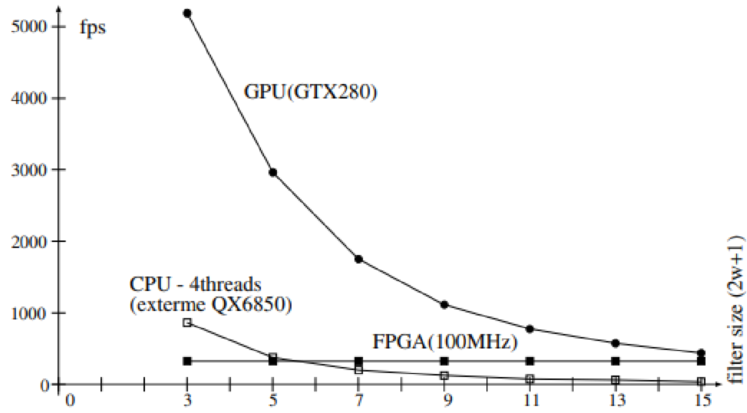

Figure 22. Performance of two-dimensional filters [18].

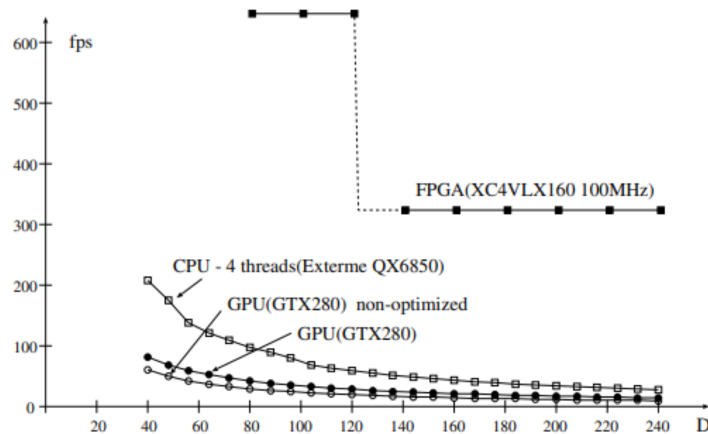

Figure 23. Performance of the stereo-vision [18].

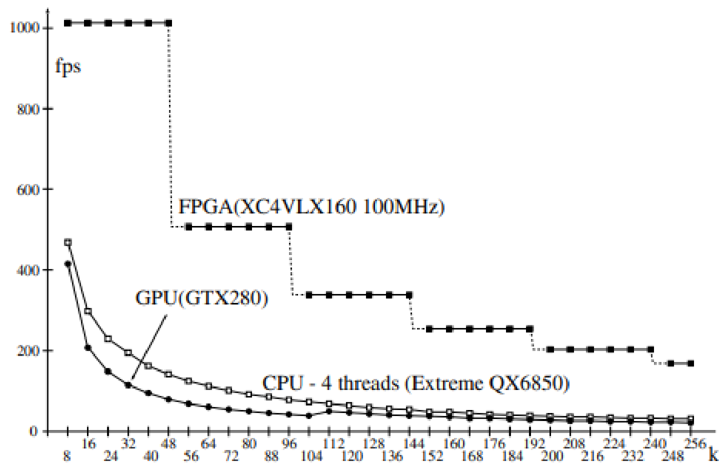

Figure 24. Performance of the k-means clustering algorithm [18].

2.4.4 FPGA Applications

FPGAs are suitable for very diverse applications like audio and video processing, cryptography, signal processing, image processing, random number generation and various algorithm implementations. It is expected that they will be used in various sectors like oil and gas, finance and many more. Current use cases include, but are not

limited to, cloud servers, artificial intelligence, space technology, defense systems, renewable energy systems, translation, derivative pricing and particle physics [14].

2.5 High Level Synthesis

The design of an application targeting an FPGA was in the past a challenging process, mostly due to the low level tools available. Traditionally programming and configuration of an FPGA was achieved through the use of a hardware description language (HDL) like Verilog or VHDL, a procedure similar to the one used for an application specific integrated circuit (ASIC) [19]. Completing this task was often quite demanding, for example in debugging the code, in comparison to other platforms like a CPU where high level languages were used. In order to overcome this obstacle in the last years there has been a development of new, more advanced methods and a creation of tools for high level synthesis (HLS) which transform the code of a high level language like C/C++ to a hardware description language.

High-level synthesis bridges hardware and software domains, providing the following primary benefits [20]:

- Improved productivity for hardware designers: hardware designers can work at a higher level of abstraction while creating high-performance hardware.
- Improved system performance for software designers: software developers can accelerate the computationally intensive parts of their algorithms on a new compilation target, the FPGA.

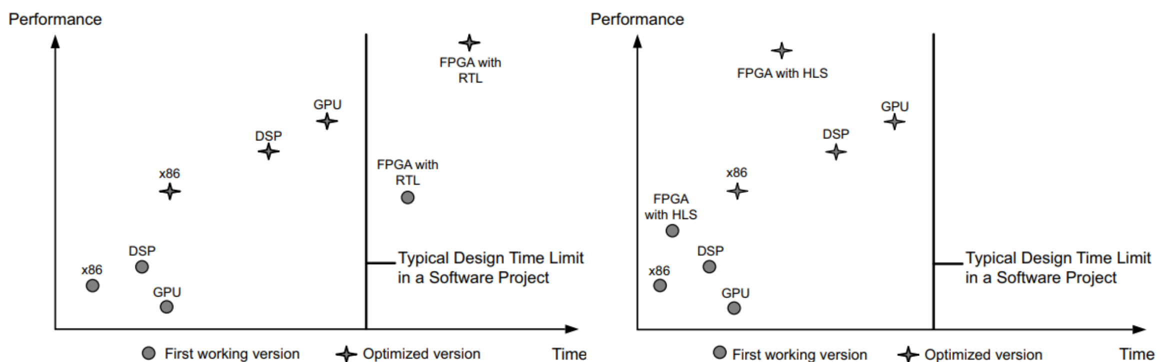

Figure 25. Design Time vs. Application Performance with RTL Design Entry (left) and Design Time vs. Application Performance with Vivado HLS Compiler (right) [13].

These are achieved through features of HLS such as[20]:

- Developing algorithms at the C-level, decreasing development time.
- Verification at the C-level. Validation of the functional correctness of the design and bug correction is way quicker in a high level language than with traditional hardware description languages.
- Control of the C synthesis process through optimization directives. Creation of specific high-performance hardware implementations and algorithm accelerators is easier.

- Creation of multiple implementations from the C source code using optimization directives for design space exploration. This enables the designer to look into trade-offs between resources, speed, power consumption and area of the circuit which increases the likelihood of finding an optimal implementation depending on the needs of the application.
- Creation of readable and portable C source code. Retarget of the C source into different FPGA devices as well as incorporation of C source into new projects is done only by changing a few parameters in the HLS tool.

2.5.1 HLS Phases

High-level synthesis includes the following phases [20]:

- Scheduling: Determines which operations occur during each clock cycle. It is based on the length of the clock cycle (clock frequency), the time it takes for the operation to complete, as defined by the target device and the user-specified optimization directives. If the clock period is longer or a faster FPGA is targeted, more operations are completed within a single clock cycle, and all operations might complete in one clock cycle. Conversely, if the clock period is shorter or a slower FPGA is targeted, high-level synthesis automatically schedules the operations over more clock cycles, and some operations might need to be implemented as multicycle resources.
- Binding: Determines which hardware resource implements each scheduled operation. To implement the optimal solution, high-level synthesis uses information about the target device.
- Control logic extraction: Extracts the control logic to create a finite state machine (FSM) that sequences the operations in the register transfer level (RTL) design.

High-level synthesis creates an optimized implementation based on default behavior, constraints, and any optimization directives the designers specify. They can use optimization directives to modify and control the default behavior of the internal logic and I/O ports. This allows them to generate variations of the hardware implementation from the same C code. To determine if the design meets their requirements, they can review the performance metrics in the synthesis report generated by high level synthesis. After analyzing the report, they can use optimization directives to refine the implementation. 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 DSPs.
- 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.
- Loop iteration latency: Number of clock cycles it takes to complete one iteration of the loop.
- Loop initiation interval: Number of clock cycles before the next iteration of the loop starts to process data.
- Loop latency: Number of cycles to execute all iterations of the loop.

2.5.2 Vitis Unified Software Platform

Vitis is a compilation of existing Xilinx tools into a single environment. While the tools may look the same, bringing all Xilinx devices under a single design environment should improve ease of use, development flow, and validation. This defines the core of Vitis' value – enhanced productivity. With it, application developers can continue to innovate on algorithm design and leverage compute acceleration without waiting for the prohibitively long design cycle of custom ASICs [21].

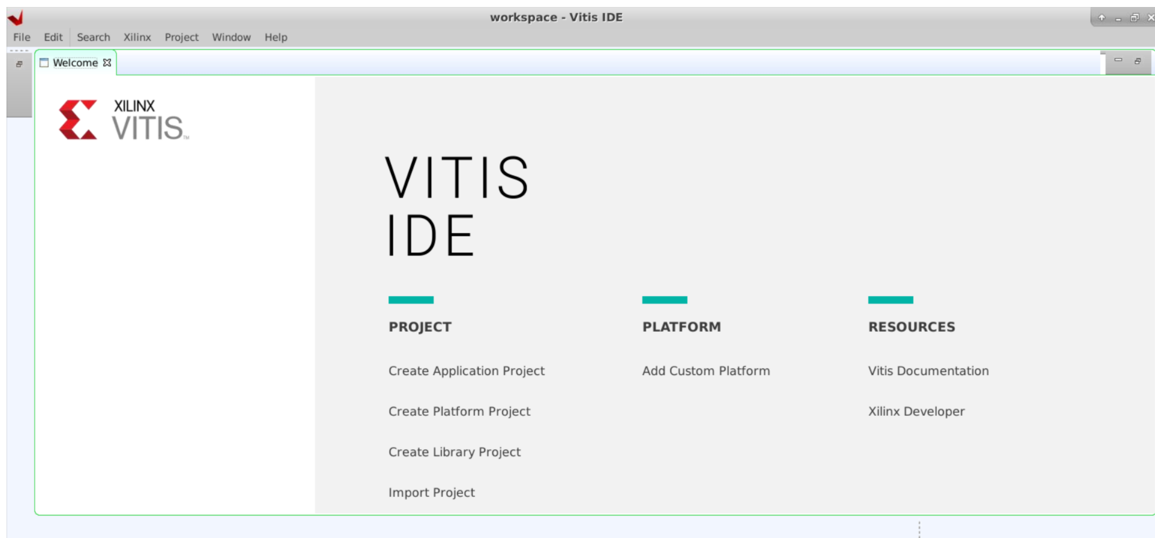

Figure 26. Vitis IDE welcome screen.

The Vitis development enables portability from platform to platform whether you are porting from development board or custom board. Vitis Platform-Based design methodology provides many productivity advantages [22]:

- Platform Reuse: Swap different acceleration applications with the same platform.
- Application Portability: Port applications across different platforms with minimum efforts.
- Simulation Time: Speed up co-simulation with kernels.
- Run Time: Open source runtime that takes care of host-device communication through PCIe or embedded for you.
- System debug: Save the full hardware compile by co-simulating the full system.

In the Vitis core development kit, an application program is split between a host application and hardware accelerated kernels with a communication channel between them. The host program, written in C/C++ and using API abstractions like OpenCL, is compiled into an executable that runs on a host processor (such as an x86 server or an ARM processor for embedded platforms); while hardware accelerated kernels are compiled into an executable device binary (.xclbin) that runs within the programmable logic (PL) region of a Xilinx device. The API calls, managed by XRT, are used to process transactions between the host program and the hardware accelerators. Communication between the host and the kernel, including control and data transfers, occurs across the PCIe bus or an AXI bus for embedded platforms. While control information is transferred between specific memory locations in the hardware, global memory is used to transfer data between the host program and the kernels. Global memory is accessible by both

the host processor and hardware accelerators, while host memory is only accessible by the host application.

For instance, in a typical application, the host first transfers data to be operated on by the kernel from host memory into global memory. The kernel subsequently operates on the data, storing results back to the global memory. Upon kernel completion, the host transfers the results back into the host memory. Data transfers between the host and global memory introduce latency, which can be costly to the overall application. To achieve acceleration in a real system, the benefits achieved by the hardware acceleration kernels must outweigh the added latency of the data transfers.

The target platform contains the FPGA accelerated kernels, global memory, and the direct memory access (DMA) for memory transfers. Kernels can have one or more global memory interfaces and are programmable. The Vitis core development kit execution model can be broken down into the following steps [23]:

1. 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, or through the AXI bus on an embedded platform.
2. The host program sets up the kernel with its input parameters.
3. The host program triggers the execution of the kernel function on the FPGA.
4. The kernel performs the required computation while reading data from global memory, as necessary.
5. The kernel writes data back to global memory and notifies the host that it has completed its task.
6. The host program reads data back from global memory into the host memory and continues processing as needed.

The FPGA can accommodate multiple kernel instances on the accelerator, both different types of kernels, and multiple instances of the same kernel. XRT transparently orchestrates the interactions between the host program and kernels in the accelerator.

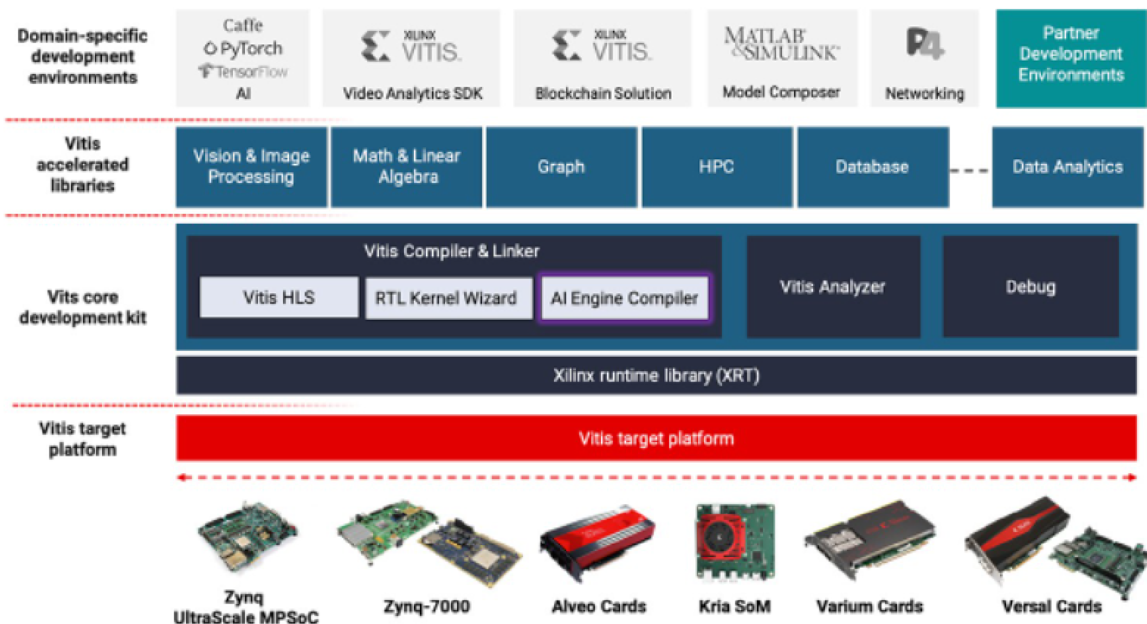

Figure 27. Vitis unified software platform overview [22].

2.6 TensorFlow and Keras

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in machine learning and developers easily build and deploy machine learning powered applications [24]. Humanly speaking, TensorFlow is an open-source library for building machine learning models at large scale. It is by far the most popular library for building deep learning models. It also has the strongest and a huge community of developers, researchers, and contributors [25]. Some of the advantages of tensorflow are [26]:

- Easy model building. TensorFlow offers multiple levels of abstraction so the developers can choose the right one for their needs. Building and training models by using the high-level Keras API makes getting started with TensorFlow and machine learning easy. If more flexibility is needed, eager execution allows for immediate iteration and intuitive debugging. For large machine learning training tasks, the Distribution Strategy API for distributed training can be used on different hardware configurations without changing the model definition.
- Robust machine learning production anywhere. TensorFlow has always provided a direct path to production. Whether it's on servers, edge devices, or the web, TensorFlow enables training and deploying models easily, no matter what language or platform is used. TensorFlow Extended (TFX) is used if there is a requirement for a full production machine learning pipeline. For running inference on mobile and edge devices there is TensorFlow Lite. Training and deploying models in JavaScript environments is done using TensorFlow.js.
- Powerful experimentation for research. Building and training state-of-the-art models without sacrificing speed or performance. TensorFlow gives this flexibility and control with features like the Keras Functional API and Model Subclassing API for creation of complex topologies. For easy prototyping and fast debugging, eager execution is used. TensorFlow also supports an ecosystem of powerful add-on libraries and models to experiment with, including Ragged Tensors, TensorFlow Probability, Tensor2Tensor and BERT.

Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity. Keras empowers engineers and researchers to take full advantage of the scalability and cross-platform capabilities of TensorFlow 2, you can run Keras on TPU or on large clusters of GPUs, and you can export your Keras models to run in the browser or on a mobile device. [27]. It is very popular in the research and development community because it supports rapid experimentation, prototyping, and user-friendly API. Being user-friendly comes with the cost of losing access to the inner details of TensorFlow, but a reasonable number of complex things can still be done [25]. The core data structures of Keras are layers and models. The simplest type of model is the Sequential model, a linear stack of layers. For more complex architectures, the Keras functional API should be used, which allows to build arbitrary graphs of layers, or write models entirely from scratch via subclassing. Keras is also a highly-flexible framework suitable to iterate on state-of-the-art research ideas. Keras follows the principle of progressive disclosure of complexity, it makes it

easy to get started, yet it makes it possible to handle arbitrarily advanced use cases, only requiring incremental learning at each step. In much the same way that training and evaluating a simple neural network is possible in a few code lines, Keras can be used to quickly develop new training procedures or exotic model architectures. Some of the features that make Keras so widely used is that is [27]:

- Simple, but not simplistic. Keras reduces developers' cognitive load to free them to focus on the parts of the problem that really matter.
- Flexible. Keras adopts the principle of progressive disclosure of complexity, simple workflows should be quick and easy, while arbitrarily advanced workflows should be possible via a clear path that builds upon what has already been learned.
- Powerful. Keras provides industry-strength performance and scalability, it is used by organizations and companies including NASA, YouTube, or Waymo.

In 2019, Google released a new version of their TensorFlow deep learning library (TensorFlow 2) that integrated the Keras API directly and promoted this interface as the default or standard interface for deep learning development on the platform. This integration is commonly referred to as the tf.keras interface or API ("tf" is short for "TensorFlow"). This is to distinguish it from the so-called standalone Keras open source project. Standalone Keras is the standalone open source project that supports TensorFlow, Theano and CNTK backends while tf.keras is the Keras API integrated into TensorFlow 2 [28].

2.6.1 Model Life Cycle

A model has a life-cycle, and this very simple knowledge provides the backbone for both modeling a dataset and understanding the tf.keras API. The five steps in the life-cycle are as follows [28]:

- Definition of the model. Defining the model first requires selecting the type of model that is needed and then choosing the network topology. From an API perspective, this involves defining the layers of the model, configuring each layer with a number of nodes and activation function, and connecting the layers together into a cohesive model.
- Compilation of the model. Compiling the model requires selecting a loss function for optimization, such as mean squared error or cross-entropy. It also requires the selection of an algorithm to perform the optimization procedure, typically stochastic gradient descent, or a modern variation, such as Adam. It may also require the definition of any performance metrics to keep track of during the model training process. From an API perspective, this involves calling a function to compile the model with the chosen configuration, which will prepare the appropriate data structures required for the efficient use of the model that has been defined.
- Fitting of the model. Fitting the model requires first selecting the training configuration, such as the number of epochs (iterations through the training dataset) and the batch size (number of samples in each epoch used to estimate the model error). Training applies the optimization algorithm to minimize the loss function and updates the model using the backpropagation of error algorithm.

Fitting the model is the slow part of the whole process and can take from seconds to days, depending on the complexity of the model, the hardware used and the size of the training dataset. From an API perspective, this involves calling a function to perform the training process. This function will block (not return) until the training process has finished.

- Evaluation of the model. Evaluating the model requires first choosing a holdout dataset used to evaluate the model. This should be data not used in the training process so that an unbiased estimate of the performance of the model can be acquired when making predictions on new data. The speed of model evaluation is proportional to the amount of data used, although it is much faster than training as the model is not changed. From an API perspective, this involves calling a function with the holdout dataset and getting a loss and perhaps other metrics that can be reported.
- Making predictions. It requires new data for which a prediction is needed, e.g. where target values are not available. From an API perspective, a simple function call is used to make a prediction of a class label, probability, or numerical value, whatever the model is designed to predict.

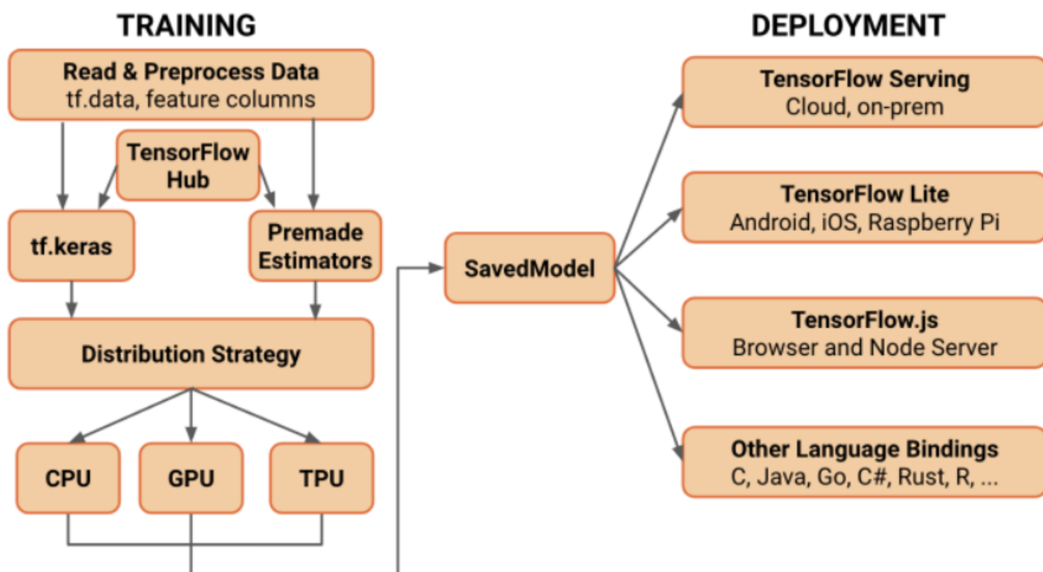

Figure 28. TensorFlow 2 architecture [25].

Chapter 3: LSTM deployment and acceleration on FPGA

In this section we describe the process followed in order to be able to accelerate our application on the FPGA. We look at all the steps needed, starting from the model creation and training in python. We continue by transferring the trained model in C++ and using this code to create our first working kernel to run on the FPGA. Finally we use optimizations provided by the tool (Vitis) to reduce the latency of the network.

3.1 Network creation and training in Python

As stated before we use the Tensorflow library for this purpose, specifically version 2.3.0 is used. To create the model we use the Sequential API which groups a linear stack of neural network layers into a model [35], the output of each layer is given as input to the next one. This makes the whole process very easy and with a few lines of code we can define and train a very complex model. The following code gives an example of a model configuration.

```
model = Sequential()
model.add(LSTM(units=128, input_shape=(30, 1)))
model.add(Dropout(rate=0.2))
model.add(RepeatVector(n=30))
model.add(LSTM(units=128, return_sequences=True))
model.add(Dropout(rate=0.2))
model.add(TimeDistributed(layer=Dense(units=1)))
model.compile(optimizer='adam', loss='mae')
```

The above snippet is all we need to do to create a model. The first line initializes a Sequential model. The next lines define the layers used and their order (the first one takes the input and passes its output to the next etc). Notice that the first layer takes the argument `input_shape`. This is the shape of the input data (not including the batch size, how many different data points we pass to the model). In our example the first dimension is 30 and the second is 1. The last line configures the model for training. The argument `optimizer`, as the name suggests, takes a string of the name of the optimizer used during training (here the optimizer is the Adam algorithm) and the argument `loss` takes a string of the name of the loss function used (or error function, this is the value we aim to minimize during training, here we have the mean absolute error -mae- between the input and the output). All the different layers we use in our application are seen in the previous example, these are the LSTM, Dropout, RepeatVector, TimeDistributed and Dense. Next we will describe the function of each one.

LSTM layer
This is an LSTM cell like the one described in chapter 2. The argument `units` refers to the dimensionality (size) of the output. Since the size of the input can be inferred from the output of the previous layer (or is explicitly defined in the first layer) we do not need any other input to the layer to specify the cell's size. The input is given in the shape (batch size, timesteps, features) so the number of timesteps is already known. The argument `return_sequences` determines whether the output contains only the result for the last timestep or for all the timesteps for the cell.

Dropout layer
The Dropout layer randomly sets input units to 0 with a frequency of `rate` at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by 1/(1 - `rate`) such that the sum over all inputs is unchanged. Note that the Dropout layer only applies during training such that no values are dropped during inference [36].

RepeatVector layer
This layer repeats its input in the output `n` times.

TimeDistributed layer
This wrapper allows to apply a layer to every temporal slice of an input. Every input should be at least 3D, and the dimension of index one of the first input will be considered to be the temporal dimension [37]. For example, given an input of shape (150, 30, 20) it will apply the `layer` given as argument to each of the 30 timesteps of 20 features independently.

Dense layer
This is a regular densely connected neural network. The argument `units` refers to the dimensionality (size) of the output.

The next step is to train the model. It is very simple, requiring just one line of code.

```
model.fit(x=x_train, y=y_train, epochs=100, batch_size=32,
validation_split=0.1,
callbacks=[keras.callbacks.EarlyStopping(monitor='val_loss',
patience=3, mode='min')], shuffle=False)
```

In order to train the model the only thing we need to do is call the fit method. The argument `x` is the input (training) data and `y` the corresponding target data. The `epochs` and `batch_size` arguments, as their names suggest, define the number of epochs to train the model and the number of samples per update. An epoch is an iteration over the entire x and y data provided. The `validation_split` is the fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The `callbacks` hook into the various stages of the model training and inference lifecycle. For example the EarlyStopping callback stops the training when

a monitored metric has stopped improving. Finally the shuffle specifies whether to shuffle the training data before each epoch.

All that is left is to make predictions using our test data. This is done by calling the predict method.

```
model.predict(x_test)
```

Before building and training the model there was some preprocessing to do with the training and test data. We will not go into much detail. In summary we used a standard scaler to standardize our data (make their mean 0 and standard deviation 1), we splitted into a train and test set and properly adjusted the shape of each set to be (batch size, timesteps, features).

3.2 Transferring the model to C++

The first step in order to execute our trained model in C++ is to understand how a Sequential model works. Each layer is independent from the others in terms of calculations made, their only interaction is that each one passes its output as input to the next. The next thing we have to do is to understand how each different layer works.

For the LSTM cell we went into much detail in chapter 2 so we are not going to repeat. It is useful to notice that a cell takes one data input with size equal to the features for each timestep, and also receives the cell state and the hidden state of the previous timestep (for the first timestep we consider them both to be 0) both with size equal to the `units` parameter. Internally we have four single-layer feedforward networks, some applications of activation functions and some pointwise vector calculations. These networks can be modeled as an array-vector multiplication between the array of weights and the input plus a vector addition between the result vector and the bias vector. For the activations we just need to apply the function pointwise to a vector.

The Dropout layer is only relevant during training and since we will use the C++ model only for inference we do not need to deal with it.

The RepeatVector is only used before an LSTM layer in order to copy the previous result. It is not necessary to copy our entire result vector, we can just give the same input to the next layer for all the timesteps.

The TimeDistributed layer applies the same layer to each timestep so it is simple to loop through every timestep and do the same.

Finally the Dense layer is a single-layer feedforward network and as stated before is modeled by an array-vector multiplication and a vector addition.

Taking all this into consideration an obvious approach is to use a different function for each layer and pass each function's output as input to the next one. The main loop for

the inference of the example model shown in chapter 4.1 can look like the following snippet of code:

```
float test_data[BATCH_SIZE][TIMESTEPS][FEATURES];
float res[BATCH_SIZE][TIMESTEPS][UNITS_DENSE];
for(int j=0; j<BATCH_SIZE; ++j)
{
	float res1[UNITS_LSTM1];
	float res2[TIMESTEPS][UNITS_LSTM2];
	LSTM_1(test_data[j], res1);
	LSTM_2(res1, res2);
	for(int i=0; i<TIMESTEPS; ++i)
	{
		Dense(res2[i], res[j][i]);
	}
}
```

Regarding the internal structure of the different layers both the Dense and LSTM layers do array-vector multiplications which can be implemented using a simple nested loop. As for the LSTM after the four multiplications all that is left is some pointwise vector calculations and the update of the hidden and cell states that are needed in the next timestep.

All that is left is to extract the weights from the trained python model and use them in our C++ code. This can be easily done with some simple python functions. The code for extracting the weights, as well as examples of LSTM and Dense layers, is available at the Appendix.

3.3 Creating a kernel for the FPGA

As mentioned before, we will use the Vitis Unified Software Platform in order to create the final executable for the FPGAs. The Vitis application acceleration development flow provides a framework for developing and delivering FPGA accelerated applications using standard programming languages for both software and hardware components. The software component, or host program, is developed using C/C++ to run on x86 or embedded processors, with OpenCL API calls to manage runtime interactions with the accelerator. The hardware component, or kernel, can be developed using C/C++, OpenCL C, or RTL [23]. In our case we use C++ for both the host program and the kernel.

Having first written our model in C++, developing a working kernel without achieving acceleration of our application, can be done quite easily and only with a few modifications to our C++ code. First of all we divide the code into two parts, the host code (executed on the processor) and the kernel code (executed on the FPGA, the one we want to accelerate). The next step is the modification of our `main` C++ function. We add some OpenCL commands to manage the kernel execution. In summary what we

need to do is find the platform and load the executable device binary, allocate memory on the device, transfer the input data, launch the kernel and retrieve the result once calculations are finished. Our kernel has a top level function visible to the host code and this is the one we interact with. As for the kernel code we can keep it as it is with minimal modifications. It is possible to use C++ data types and C++ standard library headers and the tool will produce the executable with no issues. One major exception is that dynamic memory allocation is not available. It is also worth noticing that by keeping this code the same we probably will not achieve any acceleration. We will cover optimizations we use later.

Vitis offers three different building options for our application. We begin with Software emulation. It runs fast and after completing we can simulate the execution of our kernel to check its correctness. Both the host application code and the kernel code are compiled to run on the host processor. This allows iterative algorithm refinement through fast build-and-run loops. This target is useful for identifying syntax errors, performing source-level debugging of the kernel code running together with the application, and verifying the behavior of the system. Next, there is Hardware emulation. It is slower than Software (but still pretty fast). We used it to get metrics like estimated execution time and resources used for our specific device and locate possible acceleration targets. The kernel code is compiled into a hardware model (RTL), which is run in a dedicated simulator. This build-and-run loop takes longer but provides a detailed, cycle-accurate view of kernel activity. This target is useful for testing the functionality of the logic that will go in the FPGA and getting initial performance estimates. Finally in the Hardware build the kernel code is compiled into a hardware model (RTL) and then implemented on the FPGA, resulting in a binary that will run on the actual FPGA [23]. Depending on how many optimizations we use, it can take many hours to complete.

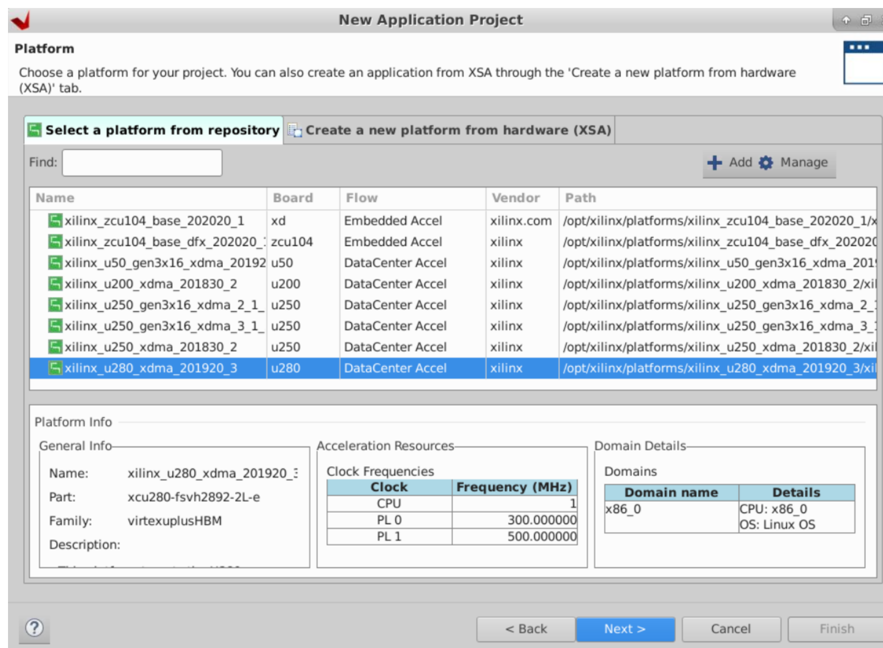

Figure 29. Platform selection using the Vitis IDE.

One major advantage of the Vitis tool is that we can use virtually the same code while targeting different devices, we just need to select our target platform when initializing the

project (see figure above). For embedded platforms we need to additionally specify the sysroot, which is part of the platform where the basic system root file structure is defined and is used to cross-compile the host application, a Linux kernel image and a rootfs (root file system) with integrated Xilinx Runtime.

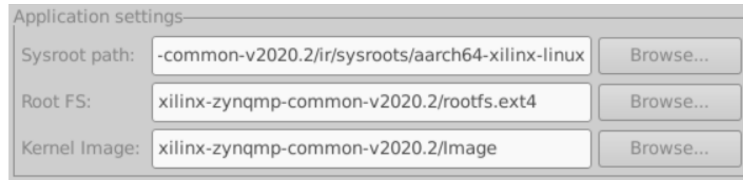

Figure 30. Sysroot, rootfs and image definition example in Vitis IDE.

We use the Vitis tool for two different development flows, data center application acceleration (for the Alveo u280) and embedded processor application acceleration (for the MPSoC ZCU104).

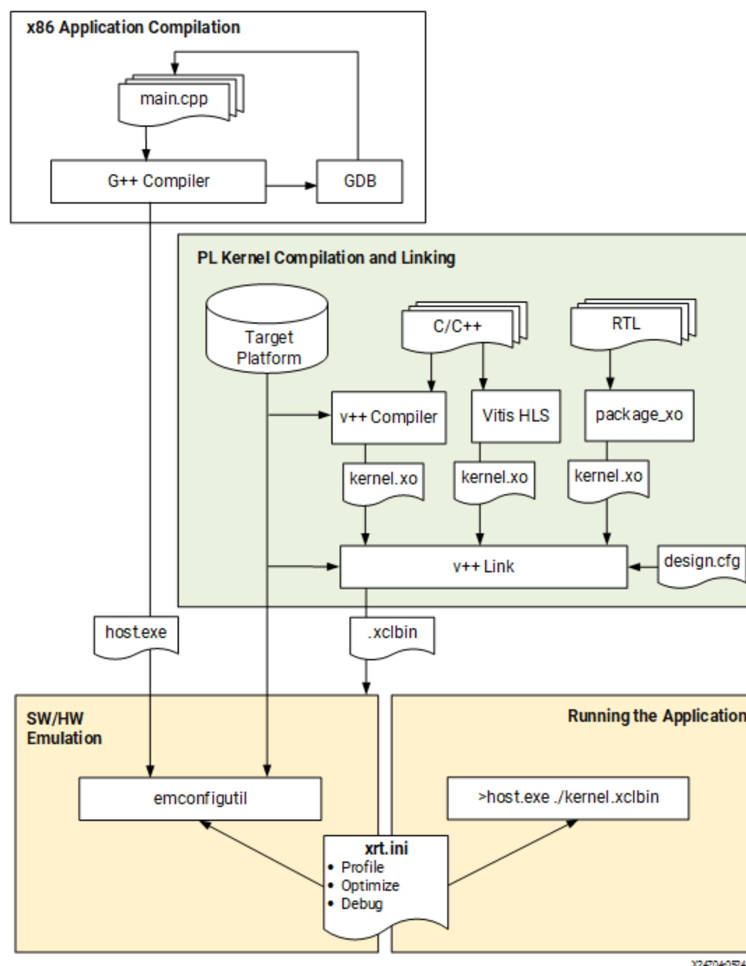

Figure 31. Application Development Flow for Data Center Accelerator Cards [23].

For data center accelerator cards the host application is compiled to run on the x86 processor using the G++ compiler to create a host executable file. The host program interacts with kernels in the PL (programmable logic) region. PL kernels are compiled for implementation in the PL region of the target platform. PL kernels are compiled into a Xilinx object form (XO) file using the Vitis compiler (v++). The Vitis compiler also links the

kernel XO files with the hardware platform to create a device executable (.xclbin) for the application. Xilinx object (XO) files are linked with the target hardware platform by the v++ --link command to create a device binary file (.xclbin) that is loaded into the Xilinx device on the target platform [23]. Finally we can run our application from the command line, for example with the following command:

```
./host.exe kernel.xclbin
```

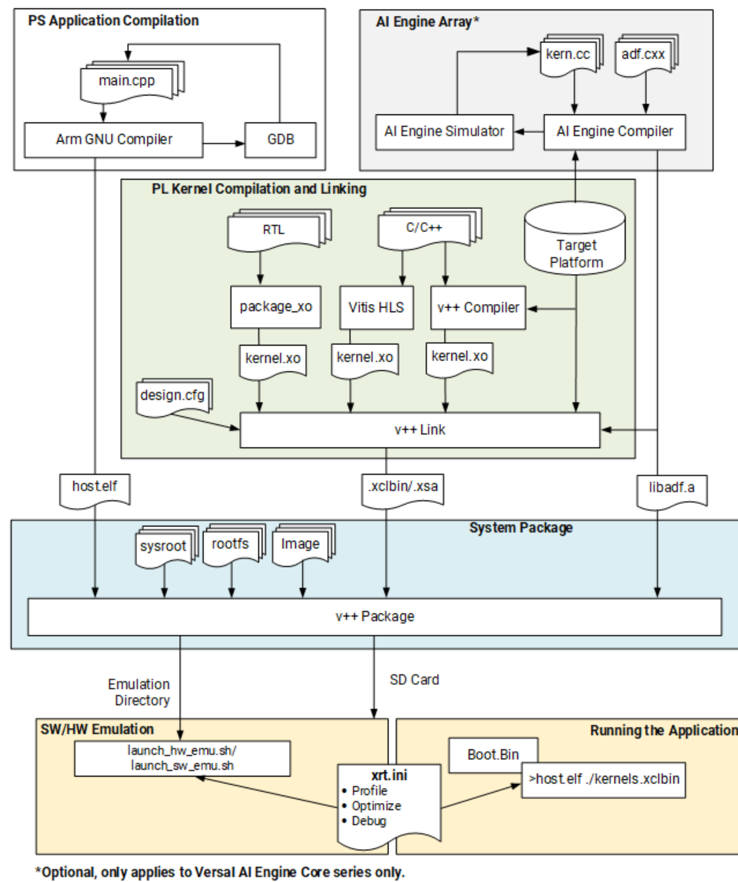

Figure 32. Application Development Flow for Versal ACAP and Zynq UltraScale+ MPSoC Devices [23].

For embedded devices the host application is compiled to run on the Cortex-A72 or Cortex-A53 core processor using the GNU ARM cross-compiler to create an ELF file. The host program interacts with kernels in the PL and AI Engine regions of the device. PL kernels are compiled for implementation in the PL region of the target platform. PL kernels can be compiled into a Xilinx object form (XO) file using the Vitis compiler (v++). The Vitis compiler also links the kernel XO files with the hardware platform to create a device executable (.xclbin) for the application and similar to data center cards Xilinx object (XO) files are linked with the target hardware platform by the v++ --link command to create a device binary file (.xclbin) that is loaded into the Xilinx device on the target platform. Finally the tool gathers the required files to configure and boot the system, to load and run the application, including the host application and PL kernel binaries. This step builds the necessary package to create an SD card to run the application on hardware [23]. For an embedded processor platform, the contents of the produced ./sd_card folder are copied to an SD card which is used as the boot device for our

system. All that is left is to boot the device from the SD card and run the application from the command line, for example:

```
./host.exe kernel.xclbin
```

3.4 Kernel Optimizations

After creating a working kernel, the next step is to make suitable optimizations in order to achieve small execution time. The optimizations we made can be separated into three different categories, code transformations, optimization pragmas and migration from floating point to fixed point decimal numbers.

3.4.1 Code transformations

In the models we implemented we use two LSTM cells which in the original C++ code are represented by different functions. This means that each of these functions requires separate resources when implemented on the FPGA despite doing the same calculations (their code is basically the same, the only differences are the use of the `return_sequences` parameter in one of the two and the different arrays of weights) and not being able to run concurrently (the second LSTM waits for the output of the first). We can eliminate the need for separate functions by inserting some simple if-else statements in the places where weights are used and by correctly handling the final result depending if we need all the values (in the case `return_sequences` is true) or not. One other thing to notice is that the four array-vector multiplications in each cell have no data dependencies and can be executed concurrently. We write a separate function for each one and the tool automatically implements them in parallel. Finally in order for the pragmas to apply correctly in the nested array-vector multiplication loop we used a separate function for the inner loop.

3.4.2 Optimization pragmas

The HLS tool provides pragmas that can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. These pragmas can be added directly to the source code for the kernel [38]. Some of the pragmas we directly utilize in our code or the tool applies automatically are the following:

pipeline pragma
Reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations. A pipelined function or loop can process new inputs every N clock cycles, where N is the II of the loop or function. An II of 1 processes a new input every clock cycle. The initiation interval can be specified through the use of the II option for the pragma. As a default behavior, with the pipeline pragma Vitis HLS will generate the minimum II for the design according to the specified clock period constraint. If the

Vitis HLS tool cannot create a design with the specified II, it issues a warning and creates a design with the lowest possible II [38].

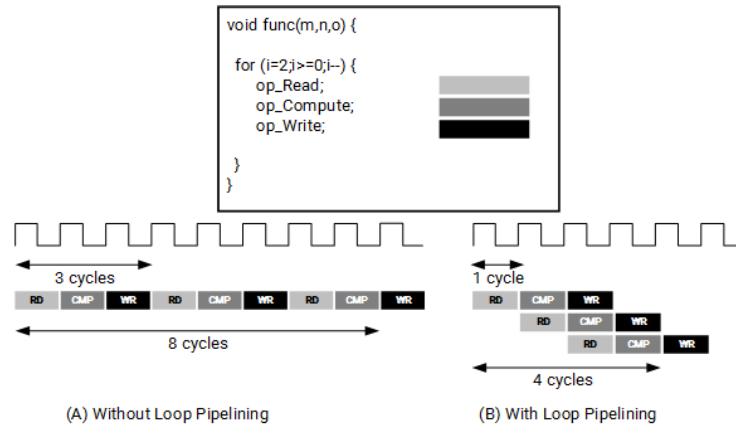

Figure 33. Loop Pipeline [38].

Example:

```
for(int i=0; i<N; ++i)
{
        #pragma HLS pipeline II=1
        //Calculations
}
```

This pragma is extremely useful when applied to loops as it reduces the total execution time without using as many resources as the unroll pragma. In our application we use it in the loops where there are no data dependencies between loop iterations.

unroll pragma

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. When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations might also be impacted by logic inside the loop body (for example, break conditions or modifications to a loop exit variable). Using the unroll pragma loops can be unrolled to increase data access and throughput.

Example:

```
for(int i=0; i<N; ++i)
{
        #pragma HLS unroll factor=4
        //Calculations
}
```

The unroll pragma allows the loop to be fully or partially unrolled. Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently. Partially unrolling a loop lets you specify a factor N, to create N copies of the loop body and reduce the loop iterations accordingly. Partial loop unrolling does not require N to be an integer factor of the maximum loop iteration count. The Vitis HLS tool adds an exit check to ensure that partially unrolled loops are functionally identical to the original loop [38].

In nested loops the unroll pragma is applied automatically in the inner loop when we use the pipeline pragma for the outer. We use it implicitly in the nested loops that perform array vector multiplications.

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. All operations are performed sequentially in a C description. The Vitis HLS tool seeks to minimize latency and improve concurrency. However, data dependencies can limit this. For example, functions or loops that access arrays must finish all read/write accesses to the arrays before they complete. This prevents the next function or loop that consumes the data from starting operation. The dataflow optimization enables the operations in a function or loop to start operation before the previous function or loop completes all its operations. When the dataflow pragma is specified, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL [38].

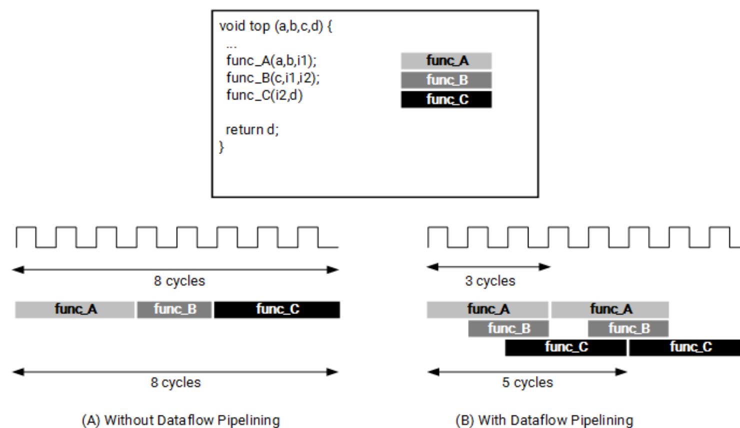

Figure 34. Dataflow Pragma [38].

Like the unroll pragma it is also applied implicitly to our design. The tool detects that there are no dependencies between the four array vector multiplications in the LSTM cell and adapts the design so that they are performed simultaneously.

array_partition pragma
Partitions an array into smaller arrays or individual elements and provides the following [38]:

- Results in RTL with multiple small memories or multiple registers instead of one large memory.
- Effectively increases the amount of read and write ports for the storage.
- Potentially improves the throughput of the design.
- Requires more memory instances or registers.

Example:

```
float A[9][4];
#pragma HLS array_partition variable=AB type=block factor=2 dim=2
```

This example partitions dimension two of the two-dimensional array A[9][4] into two new arrays of dimension [9][2].

We use this pragma to appropriately partition arrays when there is a need for simultaneous access to their elements. For example the inner loop that performs array vector multiplication is completely unrolled so we completely partition the second dimension of the 2D array as well as the vector so their elements can be available at the same time.

interface pragma
In C/C++ code, all input and output operations are performed, in zero time, through formal function arguments. In a RTL design, these same input and output operations must be performed through a port in the design interface and typically operate using a specific input/output (I/O) protocol. The interface pragma specifies how RTL ports are created from the function definition during interface synthesis [38]. We use it in the top-level hardware function to define how it communicates with the outside world.

3.4.3 Fixed point numbers

Several studies have demonstrated that inference of neural networks can be achieved with a reduced precision of operands. Weights are quantized using a fixed point representation scheme. In the simplest version of this format, numbers are encoded with the same bit-width that is set according to the numerical range and the desired precision. All the operands share the same exponent (i.e scale factor) that can be seen as the position of the radix point. When compared to floating point, fixed point computing with compact bit-width is known to be more efficient in terms of hardware utilization and power consumption [39].

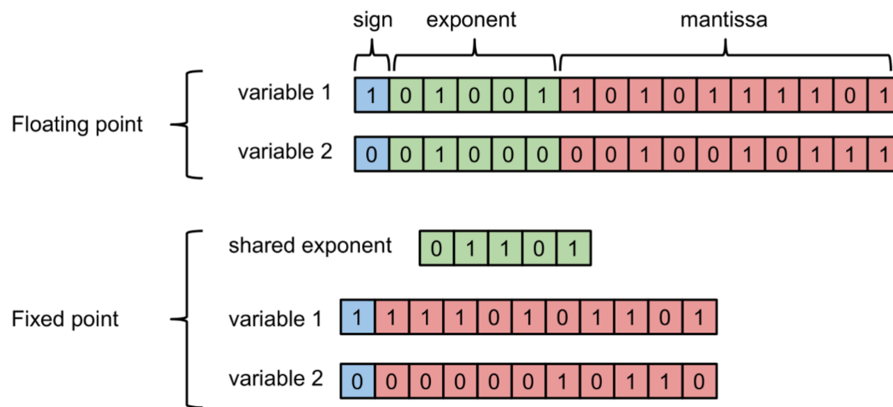

Figure 35. Comparison of the floating point and fixed point formats [40].

First of all we will describe floating point representation to make fixed point more clear. They consist of a sign, an exponent, and a mantissa. The exponent gives the floating point formats a wide range, and the mantissa gives them a good precision. Single precision floating point numbers use a total of 32 bits, 8 bits for the exponent, 23 bits for the mantissa and one for the sign. One can compute the value of a single floating point number using the following formula:

$$value = (-1)^{sign}\left(1 + \frac{mantissa}{2^{23}}\right)2^{(exponent-127)}$$

Fixed point formats consist in a signed mantissa and a global scaling factor shared between all fixed point variables. The scaling factor can be seen as the position of the radix point. It is usually fixed, hence the name ”fixed point”. Reducing the scaling factor reduces the range and augments the precision of the format. The scaling factor is typically a power of two for computational efficiency (the scaling multiplications are replaced with shifts). As a result, fixed point format can also be seen as a floating point format with a unique shared fixed exponent. Fixed point format relies on integer operations. It is hardware-wise cheaper than its floating point counterpart, as the exponent is shared and fixed [40]. This is especially true in FPGAs, where a single DSP block can either implement one 32 bit floating point multiplication, two 18×19 bit multiplications, or three 18×19 multiplications [39].

Vitis HLS provides arbitrary precision integer data types that manage the value of the integer numbers within the boundaries of a specified width. To use arbitrary precision integer data types in a C++ function we simply need to include the header file `ap_int.h` to the source code and change the data types to `ap_int<N>`, where N is a bit-size from 1 to 1024. Using fixed point decimal numbers is just as easy. First we must include the header file `ap_fixed.h` and then use the data type `ap_fixed<W, I, Q>`, where W is the word length in bits, I is the number of bits used to represent the integer value and Q is the quantization mode, which dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result [38].

For example:

```
#include <ap_fixed.h>

ap_fixed<16, 5, AP_RND> num = 5.34;
```

In our kernel we use two kinds of fixed point numbers, one for the weights and one for the calculations. They both have the same decimal bits but the weights have less bits for the integer value.

One other optimization we can make using fixed point numbers is to avoid making calculations for the activation functions, which are often quite complex. For example, tanh for values below -2 is almost equal to -1 and for values greater than 2 almost equal to 1. The values in between are finite since fixed point numbers can represent only certain values so it is easy to store them in an array and access the one we need every time. Thus we come up with the following implementation.

```
#define BITS 8
#define TANH_BASE 512 //2^(BITS+1)

typedef ap_fixed<BITS+4, 4, AP_RND> fixed_point;

fixed_point my_tanh(fixed_point x)
{
	if(x >= 2)
		return 1;
	else if(x < -2)
		return -1;
	else
	{
		ap_int<BITS+2> i = x.range();
		return tanh_vals[TANH_BASE + i.to_int()];
	}
}
```

It is easy to see that a complex calculation of a tanh value was completely eliminated and the only operation needed is an addition to calculate the index of the table with the precalculated tanh values.

Chapter 4: Results - Evaluation

In this chapter we are going to cover the experimental process followed in order to explore the potential for acceleration and low latency LSTM applications using FPGA based accelerators. We begin with a brief presentation of the FPGA devices used in this diploma thesis, namely the Xilinx Alveo U280 Data Center Accelerator Card and the Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit. Following there is a description of the models and datasets used and then our results after applying various optimizations.

4.1 Device setup

4.1.1 Alveo U280

Figure 36. Alveo U280 [29].

Xilinx Alveo Data Center accelerator cards are PCI Express compliant cards designed to accelerate compute-intensive applications such as machine learning, data analytics, and video processing in a server or workstation. On the Xilinx device, a platform consists of a static region and a dynamic region. The static region of the platform provides the basic infrastructure for the card to communicate with the host and hardware support for the kernel. It includes the following features [30]:

- Host Interface (HIF). PCIe endpoint to enable communication with external PCIe host.
- Direct Memory Access (DMA). XDMA IP and AXI Protocol Firewall IP.

- Clock, Reset, and Isolation (CRI). Basic clocking and reset for card bring-up and operation. Reset and Dynamic Function eXchange isolation structure are required for isolation during partial bitstream download.
- Card Management Peripheral (CMP). Peripherals responsible for board health and diagnostics, debug, and programming.
- Card Management Controller (CMC). UART/I2C communication to satellite controller (MSP432), QSFP, sensors and manages firmware updates from the host (over PCIe).
- Embedded RunTime Scheduler (ERT). Schedule and monitor compute units during kernel execution.

The dynamic region is the part of the device where accelerated kernels are stored and executed.

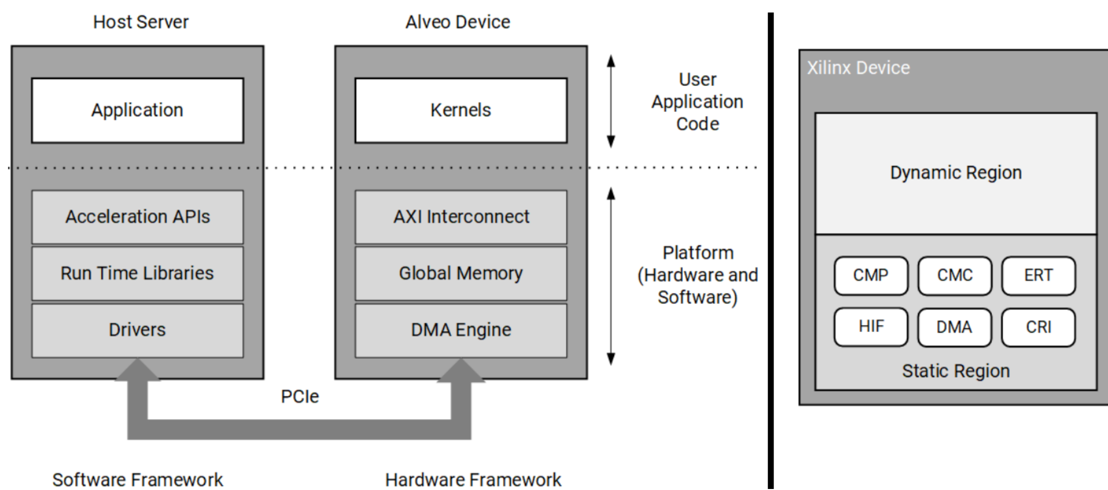

Figure 37. Platform overview (left) and dynamic and static regions in a platform (right) [30].

The Xilinx Alveo U280 accelerator card is a custom-built UltraScale+ FPGA that runs optimally (and exclusively) on the Alveo architecture. The card features the XCU280 FPGA, which uses Xilinx 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 XCU280 comprises three SLRs with the bottom SLR (SLR0) integrating an HBM controller to interface with the adjacent 8 GB HBM2 memory. The bottom SLR also connects to 16 lanes of PCI Express that can operate up to 16 GT/s (Gen4). SLR0 and SLR1 both connect to a DDR4 16 GB, 2400 MT/s, 64-bit with error correcting code (ECC) DIMM for a total of 32 GB of DDR4. SLR2 connects two QSFP28 connectors with associated clocks generated on the U280 board [31].

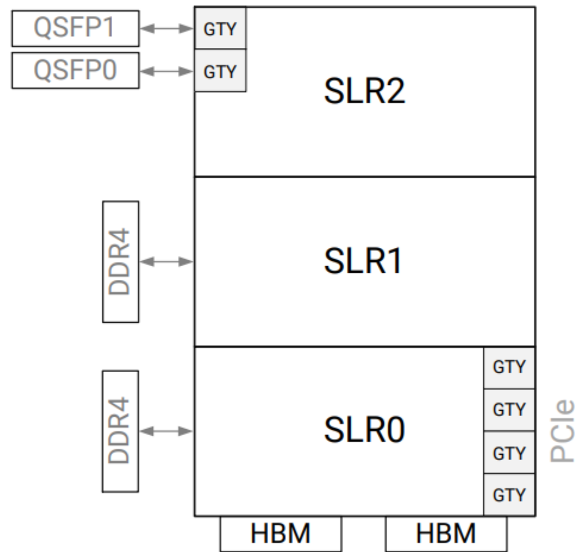

Figure 38. Floorplan of the XCU280 Device[31]

Resources

Resource	Total
BRAM	2016
DSP	9024
LUT	1303680
FF	2607360
REG	2476694

Figure 39. Alveo U280 available resources as shown in the Vitis IDE.

4.1.2 MPSoC ZCU104

Figure 40. MPSoC ZCU104 [32].

Zynq UltraScale+ MPSoC is the Xilinx second-generation Zynq platform, combining a powerful processing system (PS) and user-programmable logic (PL) into the same device. The processing system features the Arm flagship Cortex-A53 64-bit quad-core or dual-core processor and Cortex-R5F dual-core real-time processor. In addition to the cost and integration benefits previously provided by the Zynq-7000 devices, the Zynq UltraScale+ MPSoC and RFSoC devices also provide these new features and benefits [33] :

- Scalable PS with scaling for power and performance.
- Low-power running mode and sleep mode.
- Flexible user-programmable power and performance scaling.
- Advanced configuration system with device and user-security support.
- Extended connectivity support including PCIe, SATA, and USB 3.0 in the PS.
- Advanced user interface(s) with GPU and DisplayPort in the PS.
- RF circuitry with up to 16 channels of RF-ADCs and RF-DACs (RFSoC devices).
- Increased DRAM and PS-PL bandwidth.
- Improved memory traffic using Arm's advanced QoS regulators.
- Improved safety and reliability.

These new devices offer the flexibility and scalability of an FPGA, while providing the performance, power, and ease-of-use typically associated with ASICs and ASSPs. The range of the Zynq UltraScale+ family enables designers to target cost-sensitive and high-performance applications from a single platform using industry-standard tools. There are two versions of the PS; dual Cortex-A53 and quad Cortex-A53. The features of the PL vary from one device type to another. As a result, the Zynq UltraScale+ MPSoCs are able to serve a wide range of applications [33].

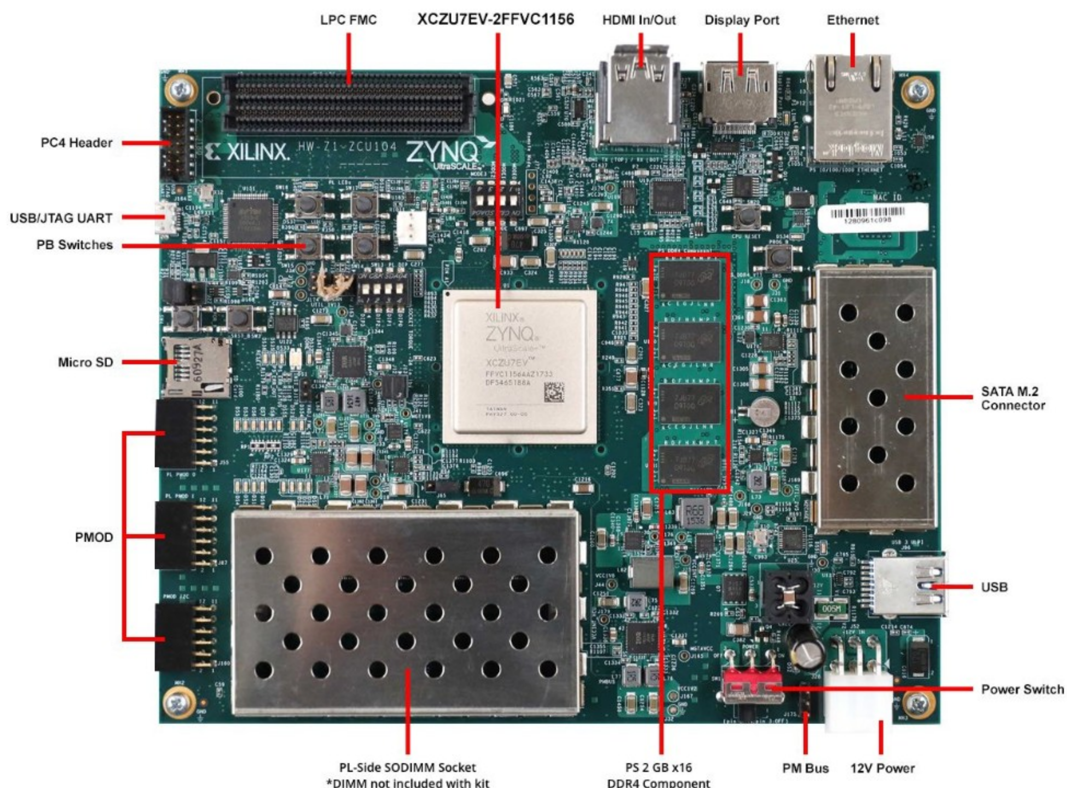

Figure 41. ZCU 104 board features [32].

The ZCU104 board is populated with the Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC, which combines a powerful processing system (PS) and programmable logic (PL) in the same device. The PS in a Zynq UltraScale+ MPSoC features the Arm flagship Cortex-A53 64-bit quad-core processor and Cortex-R5 dual-core real-time processor. The Zynq UltraScale+ MPSoC PS block has three major processing units [34]:

- Cortex-A53 application processing unit (APU)-Arm v8 architecture-based 64-bit quad-core multiprocessing CPU.
- Cortex-R5 real-time processing unit (RPU)-Arm v7 architecture-based 32-bit dual real-time processing unit with dedicated tightly coupled memory (TCM).
- Mali-400 graphics processing unit (GPU)-graphics processing unit with pixel and geometry processor and 64 KB L2 cache.

The Zynq UltraScale+ MPSoC PS has four high-speed serial I/O (HSSIO) interfaces supporting these protocols [34]:

- Integrated block for PCI Express® interface-PCIe base specification version 2.1 compliant.
- SATA 3.1 specification compliant interface.
- DisplayPort interface-implements a DisplayPort source-only interface with video resolution up to 4K x 2K-30 (300 MHz pixel rate).
- USB 3.0 interface-compliant to USB 3.0 specification implementing a 5 Gb/s line rate.
- Serial GMII interface-supports a 1 Gb/s SGMII interface.

The PS and PL can be coupled with multiple interfaces and other signals to effectively integrate user-created hardware accelerators and other functions in the PL logic that are accessible to the processors. They can also access memory resources in the PS. The PS I/O peripherals, including the static/flash memory interfaces, share a multiplexed I/O (MIO) of up to 78 MIO pins. Zynq UltraScale+ MPSoCs can also use the I/O in the PL domain for many of the PS I/O peripherals. This is done through an extended multiplexed I/O interface (EMIO) and boots at power-up or reset [34].

Resources

Resource	Total
BRAM	312
DSP	1728
LUT	230400
FF	460800

Figure 42. ZCU104 available resources as shown in the Vitis IDE.

4.2 Datasets and models used

4.2.1 Johnson & Johnson stock price

In our first application we detect anomalies in Johnson & Johnson’s (J&J) historical stock price time series data. The time period selected was from 1985-09-04 to 2020-09-03 and

this data is available through Yahoo Finance [41]. The network used for this task is an autoencoder.

Autoencoders are a specific type of feedforward neural networks where the input is the same as the output. They compress the input into a lower dimensional code and then reconstruct the output from this representation. The code is a compact "summary" or "compression" of the input, also called the latent-space representation. An autoencoder consists of 3 components: encoder, code and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code [42].

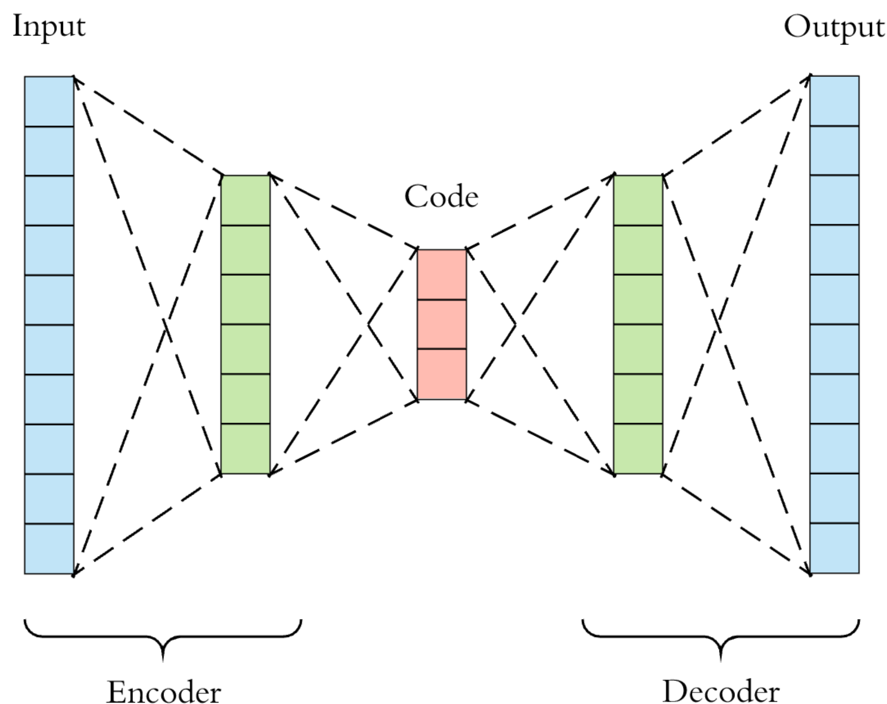

Figure 43. Autoencoder visualization [42].

Our aim when training the autoencoder network is to minimize the reconstruction error based on a loss function. First of all we divide our data into train and test sets. Since anomaly points are rare we can safely assume that our train data contain no anomalies. After training we use the autoencoder to calculate the reconstruction error on the test data. If the reconstruction error is above some threshold, we label the data point as an anomaly. The model used is the following [43]:

```
model = Sequential()
model.add(LSTM(128, input_shape=(30, 1)))
model.add(Dropout(rate=0.2))
model.add(RepeatVector(30))
model.add(LSTM(128, return_sequences=True))
model.add(Dropout(rate=0.2))
model.add(TimeDistributed(Dense(1)))
model.compile(optimizer='adam', loss='mae')
```

The first LSTM layer is the encoder, its output the code and the second LSTM layer together with the TimeDistributed-Dense layer the decoder.

4.2.2 Numenta Anomaly Benchmark

For our second application we used the Numenta Anomaly Benchmark (NAB) data set that is publicly available on Kaggle. It is a novel benchmark for evaluating algorithms for anomaly detection in streaming, online applications. It consists of over 50 labeled real world and artificial time series data files plus a novel scoring mechanism designed for real-time applications [44]. This dataset includes a time series data named 'ambient_temperature_system_failure' in the CSV format. It comprises temperature sensor data of the ambient temperature in an office setting. This is the data we used.

During preprocessing we made some additions to the dataset features. Each timestep only included the temperature (which we converted from Fahrenheit to Celsius) at this time. We added four new features, the hour of the timestep, if there was daylight at that time (we assumed that daylight is present after 7am and before 10pm), the day of the week and if it was a weekday or weekend. The model used is the following [45]:

```
model = Sequential()
model.add(LSTM(50, return_sequences=True))
model.add(Dropout(rate=0.2))
model.add(LSTM(100, return_sequences=False))
model.add(Dropout(rate=0.2))
model.add(Dense(1))
model.compile(loss='mse', optimizer='rmsprop')
```

For this model our aim in the training process is to minimize the prediction error for the next timestep based on a loss function. Same as before we divide our data into train and test sets assuming no anomalies in the test data. Then we use our model to calculate the prediction error for the train data. We make the assumption that only 1% of the timesteps are anomaly points so we chose those with the highest prediction error as the anomaly points.

The full code for both models is available at the Appendix.

4.3 Performance

After training the python models and obtaining satisfactory low error the next step is to transfer them both to the FPGA to start the acceleration process. First of all we used standard floating point decimal numbers to verify the correctness of our design, that is, that it produces the same results as the python code. Transferring from C++ that runs on CPU to the FPGA requires some code transformations so it is possible that some errors might occur. Our next step is to migrate our application to using fixed point decimal numbers. Besides their operations being faster than those of floating point numbers,

similar to integers, fixed point numbers require fewer resources thus allowing further optimizations. Their main drawback is a loss of accuracy so before deciding on a specific number of decimal bits it is necessary to make some tests to find the one that best meets our requirements. Using one or two bits less does not have a serious performance impact but it reduces the resources needed making it possible for more aggressive optimizations using the optimization pragmas.

In our models we used two different fixed point numbers, one for the weights of the trained models and one for the calculations of the different layers, for example an array vector multiplication. They use the same number of decimal bits but the weights do not require many bits for their integer part. The weights' values are less than two and greater than minus one so they only need two bits for their integer part, one for the sign and one for the value. For the integer part of the larger fixed point we looked at the range of values produced in the calculations for the training set and assumed that using enough bits to cover these numbers would be sufficient, which turned out to be correct. It is obvious that in our exploration for the best fitting fixed point data type we kept the integer bits stable and only changed the number of decimal bits.

The process we followed is simple. We built many different applications with each one having a different number of decimal bits. Then we run each one looking at differences between the python software implementation and the hardware implementation and also keeping track of some accuracy metrics. The metrics we used are:

$$Precision = \frac{TP}{TP+FP},\ Recall = \frac{TP}{TP+FN},\ F_measure = 2\frac{Precision*Recall}{Precision+Recall} = \frac{TP}{TP+\frac{1}{2}(FP+FN)}$$

where TP=True Positives, FP=False Positives, TN=True Negatives, FN=False Negatives

Since our aim is to create results as similar to the python model as possible the comparison is made between the python predictions and the ones made after running the app on the FPGA. So true positives are the anomaly points found by both the FPGA and the python model, false positives those found only by the FPGA etc. These experiments were made using the Alveo u280 acceleration card due to the fact that it was more easily available all the time and it did not require rewriting an SD card and rebooting to run the application. The results of these experiments are summarized in the following figures.

J&J dataset

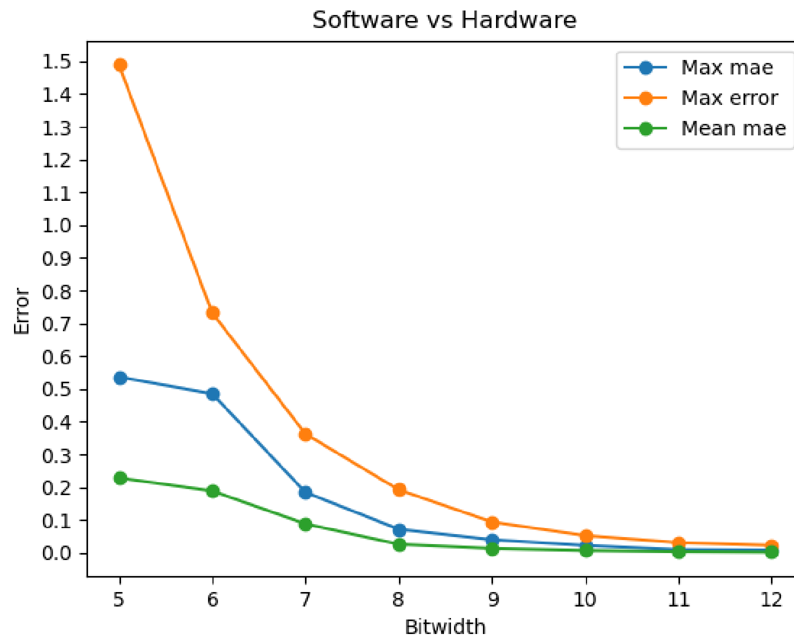

Figure 44. Reconstruction error of the FPGA implementations compared with the Python one.

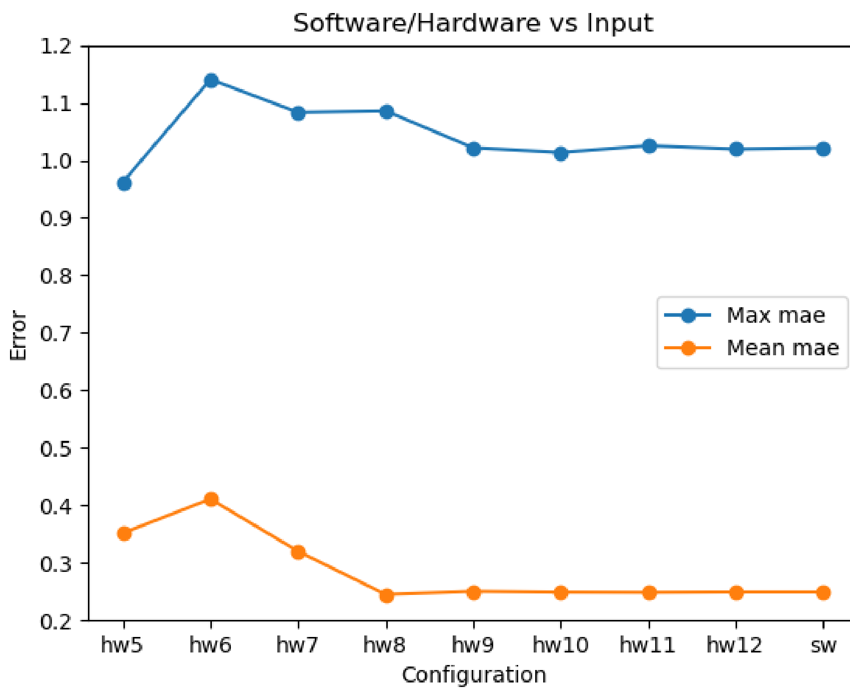

Figure 45. Reconstruction error of the FPGA-Python implementations when compared with the original data.

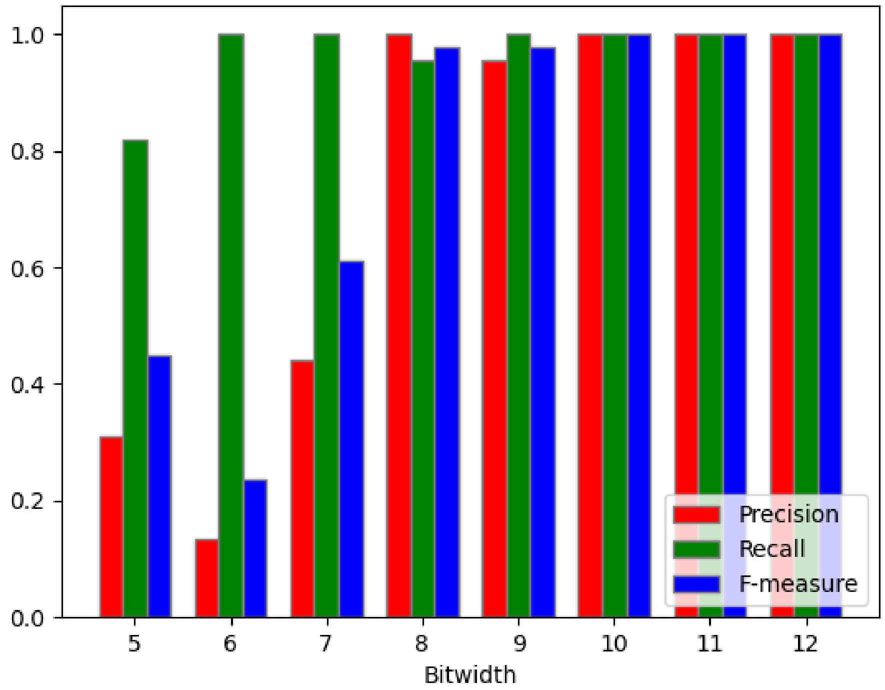

Figure 46. Accuracy metrics for the FPGA implementations.

NAB dataset

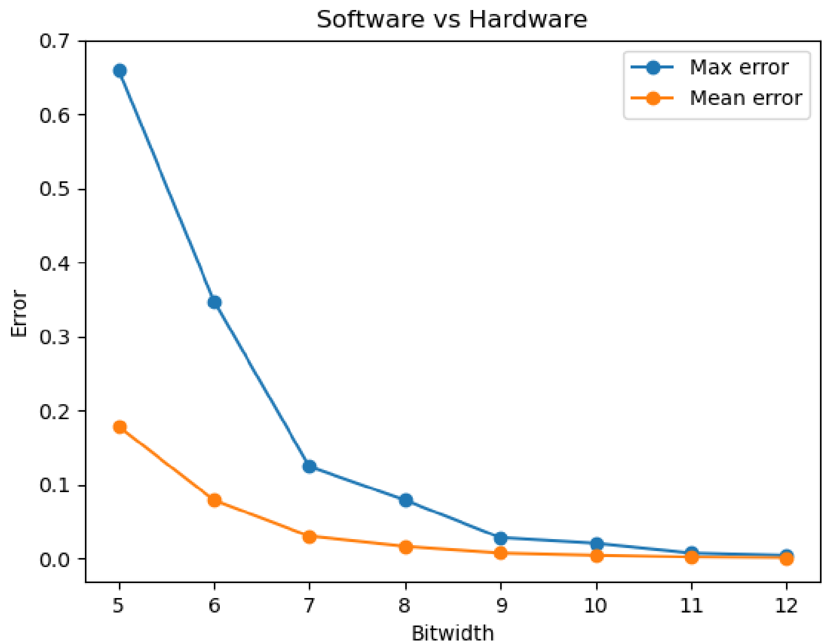

Figure 47. Prediction error of the FPGA implementations compared with the Python one.

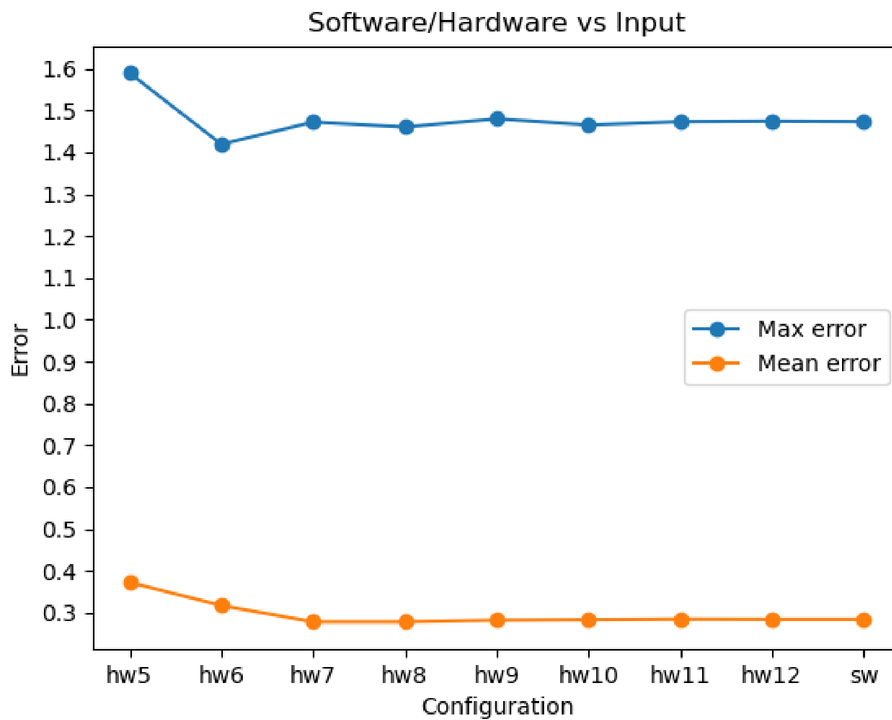

Figure 48. Prediction error of the FPGA-Python implementations when compared with the original data.

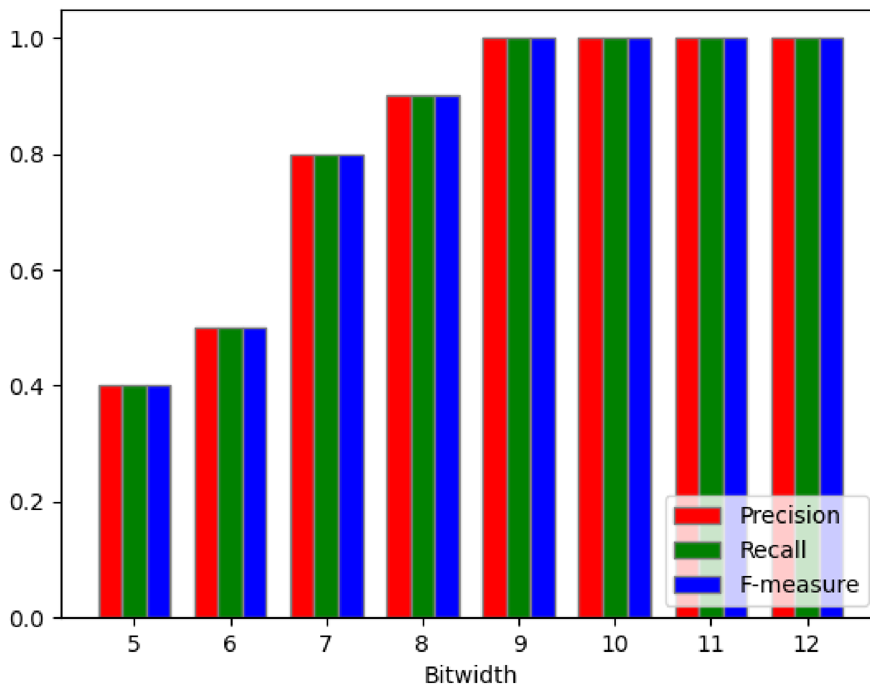

Figure 49. Accuracy metrics for the FPGA implementations.

After looking at the results for all the different configurations we notice that going from nine decimal bits and upwards all the error metrics that we keep track of become very small, the results are almost equal to those made by our original python models. The

mean reconstruction error (for the J&J dataset) and the mean prediction error (for the NAB dataset) do not show much improvement when increasing the number of decimal bits and they are almost equal with the software implementation. Moreover the value of the accuracy metrics (precision, recall, F-measure) approaches one. So we reach the conclusion that nine bits are enough for our needs and we proceed using this implementation for further acceleration.

We continue by making some code transformations on our 9-bit fixed point kernels. The two separate functions for the different LSTM cells are merged and a discrete function is used for each gate (allowing concurrent execution). The array_partition pragma is applied to arrays where simultaneous access to many elements is needed, loops where there are no data dependencies between iterations are pipelined (using the corresponding pragma) and the inner loop of the array-vector multiplications is written as a separate function so that the outer loop can be pipelined and the inner loop can be unrolled by the Vitis tool.

We present two separate versions for our accelerated applications, one being for an embedded system and the other more suitable for a server or a data center. For the embedded system we use the ZCU104 board. Its ARM processor is a typical representative of an embedded system processor. The python model is converted to TensorFlow Lite (which is a set of tools that enables on-device machine learning by helping developers run their models on mobile, embedded, and edge devices) in order to run on the board. For the data center we use a system equipped with two Intel(R) Xeon(R) Silver 4210 CPUs and the Alveo u280 acceleration card. The final acceleration for our applications is presented below. We measure the mean execution time for the inference of one data point and we display execution times for the python model and C++ without parallelization, both of which run on the CPU, and for the FPGA kernel. For the embedded system we do not provide a floating point version for the FPGA kernel. In embedded systems the available resources are limited and if we do not take advantage of fixed point numbers the potential for acceleration is severely restricted.

J&J dataset

embedded application (MPSoC ZCU 104)

python	C++	FPGA kernel
22ms	63.879ms	0.477ms

The execution time for the FPGA kernel includes 0.133ms for the transfer of data from the CPU to the FPGA and vice versa, the execution-only time is 0.344ms.

data center application (Alveo U280)

python	C++	FPGA kernel (floating point)	FPGA kernel (fixed point)
1.209ms	18.514ms	1.167ms	0.477ms

The execution time for the FPGA floating point kernel includes 0.193ms for the transfer of data from the CPU to the FPGA and vice versa, the execution-only time is 0.974ms. Accordingly, the time for the fixed point kernel includes 0.183ms for the transfer of data from the CPU to the FPGA and vice versa, the execution-only time is 0.294ms.

NAB dataset
embedded application (MPSoC ZCU 104)

python	C++	FPGA kernel
12ms	50.298ms	0.428ms

The execution time for the FPGA kernel includes 0.078ms for the transfer of data from the CPU to the FPGA and vice versa, the execution-only time is 0.350ms.

data center application (Alveo U280)

python	C++	FPGA kernel (floating point)	FPGA kernel (fixed point)
0.658ms	13.354ms	1.018ms	0.423ms

The execution time for the FPGA floating point kernel includes 0.146ms for the transfer of data from the CPU to the FPGA and vice versa, the execution-only time is 0.872ms. Accordingly, the time for the fixed point kernel includes 0.139ms for the transfer of data from the CPU to the FPGA and vice versa, the execution-only time is 0.284ms.

By looking at the execution times above there are some conclusions to be drawn. First of all, the floating point kernel achieves very little (J&J) to no acceleration (NAB) at all. Also while looking at the execution times for the python and C++ models the NAB model seems to be way faster but the FPGA kernels have almost the same latency. This can be explained by our approach in the acceleration process. Both models consist of two LSTM cells and a Dense layer. The most time consuming operations are the four array-vector multiplications of each cell. These are done in parallel so we just need to look at one of them as a unit and not all four as a whole. In the case of the NAB model the array has dimensions of 100x150 and the vector size of 150 while in the case of the J&J model the sizes are 128x256 and 256 respectively. The multiplications are done using a nested loop, where the inner loop with a number of iterations equal to the second dimension of the array is fully unrolled, so it takes almost equal amount of time for both models. All that remains is the outer loop with a number of iterations equal to the first dimension of the array, 100 for the NAB and 128 for the J&J. Every iteration for both models does the same operations so it takes about the same time. If we consider the fact that the outer loops are pipelined with an iteration interval of one clock cycle, it is obvious that the execution times for the array-vector multiplication are not very different between both models which explains the similar total execution times. Moreover, we notice that data transfer between the CPU and the FPGA is much faster in the ZCU104 board, which makes sense since the CPU and FPGA are part of the same chip. Finally,

when we focus only on the embedded application, the acceleration is almost twenty times bigger than that of the data center application. This is explained by the lower performance of the embedded ARM processor which makes the TensorFlow inference slower.

One other useful metric for measuring the performance of our applications is the number of operations per second (FLOPS). The total operations for one inference of the J&J model are 7,861,200 multiplications, 7,875,840 additions and 38,400 activations while for the NAB model we have 6,030,100 multiplications, 6,010,100 additions and 50,000 activations. If we exclude the activations (as said before we use lookup tables with precalculated values) the total is 15,737,040 and 12,040,200 operations respectively. The following table summarizes the FLOPS for each kernel.

	J&J	NAB
ZCU104	32.99 GFLOPS	28.13 GFLOPS
Alveo U280	32.99 GFLOPS	28.46 GFLOPS

Finally, it is useful to cite the total resources used by each kernel. We used the same kernel for both the ZCU104 and the Alveo U280 so it is expected that the much bigger Alveo acceleration card will have a smaller percentage of resources used.

Autoencoder (used for the J&J dataset)

Resource	Utilization	Available	Utilization %
LUT	161996	1302720	12.44
LUTRAM	12476	600480	2.08
FF	191607	2605440	7.35
BRAM	208	2016	10.32
DSP	1032	9024	11.44
IO	19	297	6.40
GT	16	24	66.67
BUFG	37	1008	3.67
MMCM	3	12	25.00
PLL	4	24	16.67
PCIe	1	6	16.67

Figure 50. Post synthesis utilization (Alveo U280).

Resource	Utilization	Available	Utilization %
LUT	50343	230400	21.85
LUTRAM	1483	101760	1.46
FF	39181	460800	8.50
BRAM	18	312	5.77
DSP	1028	1728	59.49
BUFG	8	544	1.47
MMCM	1	8	12.50

Figure 51. Post synthesis utilization (ZCU104).

LSTM network (used for the NAB dataset)

Resource	Utilization	Available	Utilization %
LUT	144090	1302720	11.06
LUTRAM	12551	600480	2.09
FF	181174	2605440	6.95
BRAM	208	2016	10.32
DSP	608	9024	6.74
IO	19	297	6.40
GT	16	24	66.67
BUFG	37	1008	3.67
MMCM	3	12	25.00
PLL	4	24	16.67
PCIe	1	6	16.67

Figure 52. Post synthesis utilization (Alveo U280).

Resource	Utilization	Available	Utilization %
LUT	33415	230400	14.50
LUTRAM	1556	101760	1.53
FF	31150	460800	6.76
BRAM	18.50	312	5.93
DSP	604	1728	34.95
BUFG	8	544	1.47
MMCM	1	8	12.50

Figure 53. Post synthesis utilization (ZCU104).

Chapter 5: Conclusion

5.1 Summary

In this diploma thesis we present the results of our work in an attempt to accelerate LSTM based anomaly detection machine learning models on FPGAs. Our effort was focused on achieving as low latency as possible while targeting the MPSoC ZCU104 and the Alveo U280, representatives of embedded and data center applications respectively.

These models were designed using the TensorFlow library in python. After proper training they were transferred in C++ which was then used for the FPGA acceleration with the help of the High Level Synthesis tools provided by Xilinx. In order to effectively deal with the problem of acceleration we then used three techniques, code transformations, application of optimization pragmas and migration from floating point to fixed point numbers. For the evaluation of our application we used two datasets, the Johnson & Johnson stock price and the Numenta Anomaly Benchmark. The final results showed that both models were accelerated with our approach, having minimal to no loss in accuracy, with both different systems (ZCU105 and U280) having almost the same latency. The acceleration is about 2.5 times for the J&J model and 1.5 times for the NAB model. If we focus only on the embedded application and we use this latency as reference, the acceleration becomes 46 and 28 times respectively. Embedded systems in general are resource constrained in terms of processing power and memory and the fact that we achieve the same acceleration as in the server application means that we can benefit from the lower energy consumption of an embedded system without concessions in terms of latency. On the other hand due to the low percentage of resources utilization of the acceleration card we could run more kernel instances in parallel, thus increasing throughput, or run a completely different kernel, thus accelerating two different applications at the same time.

5.2 Future directions

In future work, the examination of different accelerations strategies might lead to more efficient results in two ways. First of all both models achieve about the same latency while the NAB is much smaller in terms of total operations and it is much faster in python and C++ when executed in the CPU. So a different approach could be explored when dealing with such models, with smaller LSTM cells. Finally we notice that resource utilization is low in the Alveo U280 acceleration card. The exploration of a different, more aggressive acceleration strategy could lead to even better results in terms of speedup.

Appendix

Transferring weights from python to C++

```
def print_weights(f, w, u, name, data_dim, units):
  f.write('float W'+name+'['+str(units)+']['+str(data_dim+units)+'] =
{\n')
  for i in range(len(w)):
    f.write('\t{ ')
    for w1 in w[i]:
      f.write(str(w1))
      f.write(', ')
    for j in range(len(u[i])):
      if j==len(u[i])-1:
        f.write(str(u[i][j]))
        f.write(' ')
      else:
        f.write(str(u[i][j]))
        f.write(', ')
    if i==len(u)-1:
      f.write('}\n')
    else:
      f.write('},\n')
  f.write('};\n')

def print_bias(f, b, name, units):
  f.write('float b'+name+'['+str(units)+'] = { ')
  for i in range(len(b)):
    if i==len(b)-1:
      f.write(str(b[i]))
      f.write(' ')
    else:
      f.write(str(b[i]))
      f.write(', ')
  f.write('};\n')

def print_lstm_params(f, lstm, units, data_dim, name):
  '''
  Prints the weights of an LSTM layer
  f: file object corresponding to our C++ header file
  lstm: the lsmt layer
  units: dimensionality of the layer's output data
  data_dim: dimensionality of the layer's input data (for each
timestep)
  name: identifier
```

```
  '''
  W = lstm.get_weights()[0]
  U = lstm.get_weights()[1]
  b = lstm.get_weights()[2]

  # kernel
  W_i = W[:, :units]
  W_f = W[:, units: units * 2]
  W_c = W[:, units * 2: units * 3]
  W_o = W[:, units * 3:]

  # recurrent kernel
  U_i = U[:, :units]
  U_f = U[:, units: units * 2]
  U_c = U[:, units * 2: units * 3]
  U_o = U[:, units * 3:]

  # bias
  b_i = b[:units]
  b_f = b[units: units * 2]
  b_c = b[units * 2: units * 3]
  b_o = b[units * 3:]

  print_weights(f, np.transpose(W_i), np.transpose(U_i), 'i'+name,
data_dim, units)
  f.write('\n')
  print_weights(f, np.transpose(W_f), np.transpose(U_f), 'f'+name,
data_dim, units)
  f.write('\n')
  print_weights(f, np.transpose(W_c), np.transpose(U_c), 'c'+name,
data_dim, units)
  f.write('\n')
  print_weights(f, np.transpose(W_o), np.transpose(U_o), 'o'+name,
data_dim, units)
  f.write('\n')
  print_bias(f, b_i, 'i'+name, units)
  f.write('\n')
  print_bias(f, b_f, 'f'+name, units)
  f.write('\n')
  print_bias(f, b_c, 'c'+name, units)
  f.write('\n')
  print_bias(f, b_o, 'o'+name, units)
  f.write('\n')

def print_dense_params(f, dense, inp, out, name):
  '''
```

```
  Prints the weights of a Dense layer
  f: file object corresponding to our C++ header file
  dense: the dense layer
  inp: dimensionality of the layer's input data
  out: dimensionality of the layer's output data
  name: identifier
  '''
  W = np.transpose(dense.get_weights()[0])
  b = dense.get_weights()[1]

  f.write('float W'+name+'['+str(out)+']['+str(inp)+'] = {\n')
  for i in range(len(W)):
    f.write('\t{ ')
    for j in range(len(W[i])):
      if j==len(W[i])-1:
        f.write(str(W[i][j]))
        f.write(' ')
      else:
        f.write(str(W[i][j]))
        f.write(', ')
    if i==len(W)-1:
      f.write('}\n')
    else:
      f.write('},\n')
  f.write('};\n')
  f.write('\n')
  print_bias(f, b, '', out)
  f.write('\n')

'''
Example of printing weights for the following model
model = Sequential()
model.add(LSTM(128, input_shape=(30, 1)))
model.add(Dropout(rate=0.2))
model.add(RepeatVector(30))
model.add(LSTM(128, return_sequences=True))
model.add(Dropout(rate=0.2))
model.add(TimeDistributed(Dense(1)))
model.compile(optimizer='adam', loss='mae')
'''
def print_all(model):
  f = open('weights.h', 'w')
  f.write('#ifndef _WEIGHTS_H_\n')
  f.write('#define _WEIGHTS_H_\n\n')
  print_lstm_params(f, model.layers[0], 128, 1, '1')
  print_lstm_params(f, model.layers[3], 128, 128, '2')
```

```
print_dense_params(f, model.layers[5], 128, 1, '')
f.write('#endif')
f.close()
```

Example of an LSTM layer in C++

```
void LSTM_1(float data[TIMESTEPS][DATA_DIM1], float res[UNITS1])
{
      float c[UNITS1], h[UNITS1];

      for(int i=0; i<UNITS1; ++i)
      {
            c[i] = 0.0;
            h[i] = 0.0;
      }

      for(int k=0; k<TIMESTEPS; ++k)
      {
            float f_t[UNITS1], i_t[UNITS1], c_t[UNITS1], o_t[UNITS1],
x_h[UNITS1+DATA_DIM1];

            for(int i=0; i<DATA_DIM1; ++i)
                  x_h[i]=data[k][i];
            for(int i=DATA_DIM1; i<DATA_DIM1+UNITS1; ++i)
                  x_h[i]=h[i-DATA_DIM1];

            //i_t = Wi1*x_h + bi1
            array_vector_mul(Wi1, bi1, x_h, i_t);
            array_vector_mul(Wf1, bf1, x_h, f_t);
            array_vector_mul(Wc1, bc1, x_h, c_t);
            array_vector_mul(Wo1, bo1, x_h, o_t);

            vector_sigmoid(i_t);
            vector_sigmoid(f_t);
            vector_tanh(c_t);
            vector_sigmoid(o_t);

            for(int i=0; i<UNITS1; ++i)
                  c[i] = f_t[i]*c[i] + c_t[i]*i_t[i];

            for(int i=0; i<UNITS1; ++i)
                  h[i] = o_t[i]*tanh(c[i]);
      }
```

```
        for(int i=0; i<UNITS1; ++i)
                res[i]=h[i];
}
```

Example of Dense layer in C++

```
void Dense(float data[DATA_DIM2], float res[UNITS2])
{
        for(int i=0; i<UNITS2; ++i)
        {
                for(int j=0; j<DATA_DIM2; ++j)
                {
                        float sum = (j==0) ? b[i] : res[i];
                        res[i] = sum + W[i][j]*data[j];
                }
        }
}
```

Quantizing tanh and sigmoid

```
def sigmoid(x):
    f = 1 / (1 + np.exp(-x))
    return f

def print_activations(bitwidth):
  '''
  Prints the values for tanh and sigmoid functions for a specified
number of decimal bits in a header file.
  bitwidth: the number of decimal bits used in our fixed point
numbers
  '''
  f = open('activations'+str(bitwidth)+'.h', 'w')
  step = 2**(-bitwidth)
  index = -2.0
  iter = 2**(bitwidth+2)
  f.write('#ifndef _ACTIVATIONS'+str(bitwidth)+'_H_\n')
  f.write('#define _ACTIVATIONS'+str(bitwidth)+'_H_\n\n')
  f.write('//tanh values from -2 to '+str(2-step)+' with a step of
'+str(step)+'\n')
  f.write('fixed_point tanh_vals['+str(2**(bitwidth+2))+'] = {')
  for i in range(iter):
    if i==iter-1:
```

```
        f.write(str(np.tanh(index)))
        f.write(' };\n')
      else:
        f.write(str(np.tanh(index)))
        f.write(', ')
        index += step
    f.write('\n')
    index = -4.0
    iter = 2**(bitwidth+3)
    f.write('//sigmoid values from -4 to '+str(4-step)+' with a step of
  '+str(step)+'\n')
    f.write('fixed_point sigmoid_vals['+str(2**(bitwidth+3))+'] = {')
    for i in range(iter):
      if i==iter-1:
        f.write(str(sigmoid(index)))
        f.write(' };\n')
      else:
        f.write(str(sigmoid(index)))
        f.write(', ')
        index += step
    f.write('\n')
    f.write('#endif\n')
    f.close()
```

Python code for J&J dataset

Code taken from [43]

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import plotly.graph_objects as go

np.random.seed(1)
tf.random.set_seed(1)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout,
RepeatVector, TimeDistributed

#Data
df = pd.read_csv('JNJ.csv')
```

```
df = df[['Date', 'Close']]
df['Date'] = pd.to_datetime(df['Date'])

#Visualize the timeseries
fig = go.Figure()
fig.add_trace(go.Scatter(x=df['Date'], y=df['Close'], name='Close
price'))
fig.update_layout(showlegend=True, title='Johnson and Johnson Stock
Price 1985-2020')
fig.show()

#Preprocessing
train, test = df.loc[df['Date'] <= '2013-09-03'], df.loc[df['Date'] >
'2013-09-03']

##Standardize the data
scaler = StandardScaler()
scaler = scaler.fit(train[['Close']])

train['Close'] = scaler.transform(train[['Close']])
test['Close'] = scaler.transform(test[['Close']])

##Create sequences
TIME_STEPS=30

def create_sequences(X, y, time_steps=TIME_STEPS):
    Xs, ys = [], []
    for i in range(len(X)-time_steps):
        Xs.append(X.iloc[i:(i+time_steps)].values)
        ys.append(y.iloc[i+time_steps])

    return np.array(Xs), np.array(ys)

X_train, y_train = create_sequences(train[['Close']], train['Close'])
X_test, y_test = create_sequences(test[['Close']], test['Close'])

print(f'Training shape: {X_train.shape}')
print(f'Testing shape: {X_test.shape}')

#Build the Model
model = Sequential()
model.add(LSTM(128, input_shape=(X_train.shape[1],
X_train.shape[2])))
model.add(Dropout(rate=0.2))
model.add(RepeatVector(X_train.shape[1]))
model.add(LSTM(128, return_sequences=True))
```

```
model.add(Dropout(rate=0.2))
model.add(TimeDistributed(Dense(X_train.shape[2])))
model.compile(optimizer='adam', loss='mae')
model.summary()

#Train the Model
history = model.fit(X_train, y_train, epochs=100, batch_size=32,
validation_split=0.1,
callbacks=[keras.callbacks.EarlyStopping(monitor='val_loss',
patience=3, mode='min')], shuffle=False)

plt.plot(history.history['loss'], label='Training loss')
plt.plot(history.history['val_loss'], label='Validation loss')
plt.legend()
plt.show()

#Determine Anomalies
X_train_pred = model.predict(X_train, verbose=0)
train_mae_loss = np.mean(np.abs(X_train_pred - X_train), axis=1)

plt.hist(train_mae_loss, bins=50)
plt.xlabel('Train MAE loss')
plt.ylabel('Number of Samples')
plt.show()

threshold = np.max(train_mae_loss)
print(f'Reconstruction error threshold: {threshold}')

X_test_pred = model.predict(X_test, verbose=0)
test_mae_loss = np.mean(np.abs(X_test_pred-X_test), axis=1)

plt.hist(test_mae_loss, bins=50)
plt.xlabel('Test MAE loss')
plt.ylabel('Number of samples')
plt.show()

test_score_df = pd.DataFrame(test[TIME_STEPS:])
test_score_df['loss'] = test_mae_loss
test_score_df['threshold'] = threshold
test_score_df['anomaly'] = test_score_df['loss'] >
test_score_df['threshold']
test_score_df['Close'] = test[TIME_STEPS:]['Close']

fig = go.Figure()
fig.add_trace(go.Scatter(x=test_score_df['Date'],
y=test_score_df['loss'], name='Test loss'))
```

```
fig.add_trace(go.Scatter(x=test_score_df['Date'],
y=test_score_df['threshold'], name='Threshold'))
fig.update_layout(showlegend=True, title='Test loss vs. Threshold')
fig.show()

anomalies = test_score_df.loc[test_score_df['anomaly'] == True]

#Visualize Anomalies
fig = go.Figure()
fig.add_trace(go.Scatter(x=test_score_df['Date'],
y=scaler.inverse_transform(test_score_df[['Close']]).reshape(-1),
name='Close price'))
fig.add_trace(go.Scatter(x=anomalies['Date'],
y=scaler.inverse_transform(anomalies[['Close']]).reshape(-1),
mode='markers', name='Anomaly'))
fig.update_layout(showlegend=True, title='Detected anomalies')
fig.show()
```

Python code for NAB dataset

Code taken from [45]

```
import pandas as pd
import numpy as np
import matplotlib
import seaborn
import matplotlib.dates as md
from matplotlib import pyplot as plt
from sklearn import preprocessing
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, Activation,
TimeDistributed

#Data
df = pd.read_csv('ambient_temperature_system_failure.csv')

#Visualize Data
df['timestamp'] = pd.to_datetime(df['timestamp'])
df['value'] = (df['value'] - 32) * 5/9
df.plot(x='timestamp', y='value', figsize=(10,5))
plt.title('Temperature (Degree Celcius)', fontsize=16)
plt.grid()
plt.show()
```

```
#Formating the data into required format
df['hours'] = df['timestamp'].dt.hour
df['daylight'] = ((df['hours'] >= 7) & (df['hours'] <=
22)).astype(int)
df['DayOfTheWeek'] = df['timestamp'].dt.dayofweek
df['WeekDay'] = (df['DayOfTheWeek'] < 5).astype(int)

# Anomaly estimated population
outliers_fraction = 0.01

df['time_epoch'] =
(df['timestamp'].astype(np.int64)/100000000000).astype(np.int64)

df['categories'] = df['WeekDay']*2 + df['daylight']
a = df.loc[df['categories'] == 0, 'value']
b = df.loc[df['categories'] == 1, 'value']
c = df.loc[df['categories'] == 2, 'value']
d = df.loc[df['categories'] == 3, 'value']

#Visualizing the formatted data
fig, ax = plt.subplots(figsize=(10,5))
a_heights, a_bins = np.histogram(a)
b_heights, b_bins = np.histogram(b, bins=a_bins)
c_heights, c_bins = np.histogram(c, bins=a_bins)
d_heights, d_bins = np.histogram(d, bins=a_bins)
width = (a_bins[1] - a_bins[0])/6
ax.bar(a_bins[:-1], a_heights*100/a.count(), width=width,
facecolor='blue', label='Weekend Night')
ax.bar(b_bins[:-1]+width, (b_heights*100/b.count()), width=width,
facecolor='green', label ='Weekend Light')
ax.bar(c_bins[:-1]+width*2, (c_heights*100/c.count()), width=width,
facecolor='red', label ='Weekday Night')
ax.bar(d_bins[:-1]+width*3, (d_heights*100/d.count()), width=width,
facecolor='black', label ='Weekday Light')
plt.legend()
plt.show()

#Preparing the data for LSTM model
data_n = df[['value', 'hours', 'daylight', 'DayOfTheWeek',
'WeekDay']]
min_max_scaler = preprocessing.StandardScaler()
np_scaled = min_max_scaler.fit_transform(data_n)
data_n = pd.DataFrame(np_scaled)

#Important parameters and training/Test size
prediction_time = 1
```

```
testdatasize = 1000
unroll_length = 50
testdatacut = testdatasize + unroll_length  + 1

#Training data
x_train = data_n[0:-prediction_time-testdatacut].values
y_train = data_n[prediction_time:-testdatacut  ][0].values

#Test data
x_test = data_n[0-testdatacut:-prediction_time].values
y_test = data_n[prediction_time-testdatacut:  ][0].values

def unroll(data,sequence_length=24):
    result = []
    for index in range(len(data) - sequence_length):
        result.append(data[index: index + sequence_length])
    return np.asarray(result)

#Adapt the datasets for the sequence data shape
x_train = unroll(x_train,unroll_length)
x_test  = unroll(x_test,unroll_length)
y_train = y_train[-x_train.shape[0]:]
y_test  = y_test[-x_test.shape[0]:]

#Shape of the data
print("x_train", x_train.shape)
print("y_train", y_train.shape)
print("x_test", x_test.shape)
print("y_test", y_test.shape)

#Building the model
model = Sequential()
model.add(LSTM(50, input_dim=x_train.shape[-1],
return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(100, return_sequences=False))
model.add(Dropout(0.2))
model.add(Dense(units=1))
model.add(Activation('linear'))
model.compile(loss='mse', optimizer='rmsprop')
model.summary()

#Train the model

model.fit(x_train, y_train, batch_size=3028, epochs=50,
validation_split=0.1)
```

```
#Creating the list of difference between prediction and test data
loaded_model = model
diff=[]
ratio=[]
p = loaded_model.predict(x_test)
for u in range(len(y_test)):
    pr = p[u][0]
    ratio.append((y_test[u]/pr)-1)
    diff.append(abs(y_test[u]- pr))

#Plotting the prediction and the reality (for the test data)
plt.figure(figsize = (10, 5))
plt.plot(p,color='red', label='Prediction')
plt.plot(y_test,color='blue', label='Test Data')
plt.legend(loc='upper left')
plt.grid()
plt.legend()
plt.show()

#Pick the most distant prediction/reality data points as anomalies
diff = pd.Series(diff)
number_of_outliers = int(outliers_fraction*len(diff))
threshold = diff.nlargest(number_of_outliers).min()
#Data with anomaly label
test = (diff >= threshold).astype(int)
complement = pd.Series(0, index=np.arange(len(data_n)-testdatasize))
df['anomaly27'] = complement.append(test, ignore_index='True')
print(df['anomaly27'].value_counts())

#Visualizing anomalies (Red Dots)
plt.figure(figsize=(15,10))
a = df.loc[df['anomaly27'] == 1, ['time_epoch', 'value']] #anomaly
plt.plot(df['time_epoch'], df['value'], color='blue', zorder=1)
plt.scatter(a['time_epoch'],a['value'], color='red', label =
'Anomaly', zorder=2)
plt.axis([1.370*1e7, 1.405*1e7, 15,30])
plt.grid()
plt.legend()
plt.show()
```

C++ FPGA code for J&J dataset

host code

main.cpp

```
#include <stdlib.h>
#include <fstream>
#include <iostream>
#define CL_HPP_CL_1_2_DEFAULT_BUILD
#define CL_HPP_TARGET_OPENCL_VERSION 120
#define CL_HPP_MINIMUM_OPENCL_VERSION 120
#define CL_HPP_ENABLE_PROGRAM_CONSTRUCTION_FROM_ARRAY_COMPATIBILITY 1
#include <CL/cl2.hpp>
#include "read_csv.h"
#include "standard_scaler.h"
#include "layers.h"

using namespace std;

float mae(float a[][1], float b[])
{
	float sum = 0.0;
	for(int i=0; i<TIMESTEPS; ++i)
		sum += abs(a[i][0]-b[i]);
	return sum/TIMESTEPS;
}

int main(int argc, char* argv[])
{
	vector<vector<string>> data_train = read_csv("JNJ.csv", 1,
7059, {0, 4});
	vector<vector<string>> data_test = read_csv("JNJ.csv", 7060,
8823, {0, 4});
	vector<float> close;
	float test[1764][1];

	for(auto it=data_train[1].begin(); it!=data_train[1].end();
it++)
		close.push_back(stof(*it));

	StandardScaler sc;
	sc.fit(close);

	close.clear();

	for(auto it=data_test[1].begin(); it!=data_test[1].end(); it++)
```

```
            close.push_back(stof(*it));

      sc.transform(close, test);

    //TARGET_DEVICE macro needs to be passed from gcc command line
    if(argc != 2) {
            std::cout << "Usage: " << argv[0] <<" <xclbin>" <<
std::endl;
            return EXIT_FAILURE;
      }

    char* xclbinFilename = argv[1];

    // Compute the size of array in bytes
    size_t size_in_bytes = TIMESTEPS * DATA_DIM1 * sizeof(float);
    size_t size_out_bytes = TIMESTEPS * DATA_DIM2 * sizeof(float);

    std::vector<cl::Device> devices;
    cl::Device device;
    std::vector<cl::Platform> platforms;
    bool found_device = false;

    //traversing all Platforms To find Xilinx Platform and targeted
    //Device in Xilinx Platform
    cl::Platform::get(&platforms);
    for(size_t i = 0; (i < platforms.size() ) & (found_device ==
false) ;i++){
        cl::Platform platform = platforms[i];
        std::string platformName =
platform.getInfo<CL_PLATFORM_NAME>();
        if ( platformName == "Xilinx"){
            devices.clear();
            platform.getDevices(CL_DEVICE_TYPE_ACCELERATOR,
&devices);
          if (devices.size()){
                device = devices[0];
                found_device = true;
                break;
          }
        }
    }
    if (found_device == false){
       std::cout << "Error: Unable to find Target Device "
           << device.getInfo<CL_DEVICE_NAME>() << std::endl;
       return EXIT_FAILURE;
    }
```

```
    // Creating Context and Command Queue for selected device
    cl::Context context(device);
    cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE);

    // Load xclbin
    std::cout << "Loading: '" << xclbinFilename << "'\n";
    std::ifstream bin_file(xclbinFilename, std::ifstream::binary);
    bin_file.seekg (0, bin_file.end);
    unsigned nb = bin_file.tellg();
    bin_file.seekg (0, bin_file.beg);
    char *buf = new char [nb];
    bin_file.read(buf, nb);

    // Creating Program from Binary File
    cl::Program::Binaries bins;
    bins.push_back({buf,nb});
    devices.resize(1);
    cl::Program program(context, devices, bins);

    // This call will get the kernel object from program. A kernel is
an
    // OpenCL function that is executed on the FPGA.
    cl::Kernel
krnl_lstm_autoencoder(program,"krnl_lstm_autoencoder");

    // These commands will allocate memory on the Device. The
cl::Buffer objects can
    // be used to reference the memory locations on the device.
    cl::Buffer buffer_data(context, CL_MEM_READ_ONLY, size_in_bytes);
    cl::Buffer buffer_result(context, CL_MEM_WRITE_ONLY,
size_out_bytes);

    //set the kernel Arguments
    int narg=0;
    krnl_lstm_autoencoder.setArg(narg++,buffer_data);
    krnl_lstm_autoencoder.setArg(narg++,buffer_result);

    //We then need to map our OpenCL buffers to get the pointers
    float *ptr_data = (float *) q.enqueueMapBuffer (buffer_data ,
CL_TRUE , CL_MAP_WRITE , 0, size_in_bytes);
    float *ptr_result = (float *) q.enqueueMapBuffer (buffer_result ,
CL_TRUE , CL_MAP_READ , 0, size_out_bytes);

    ofstream fldump_hw, fldump_sw;
    fldump_hw.open("hw_results8.dat");
```

```
    fldump_sw.open("sw_results.dat");
    float threshold=0.58;
    cout<<"ANOMALY POINTS:\n";
    for(int j=0; j<1733; ++j)
    {
            //setting input data
            for(int i = 0 ; i< TIMESTEPS; i++){
                  ptr_data[i] = test[i+j][0];
            }

            // Data will be migrated to kernel space
            q.enqueueMigrateMemObjects({buffer_data},0/* 0 means from
host*/);

            //Launch the Kernel
            q.enqueueTask(krnl_lstm_autoencoder);

            // The result of the previous kernel execution will need
to be retrieved in
            // order to view the results. This call will transfer the
data from FPGA to
            // source_results vector

q.enqueueMigrateMemObjects({buffer_result},CL_MIGRATE_MEM_OBJECT_HOST
);

            q.finish();

            // Print result
            for(int i=0; i<TIMESTEPS; ++i)
                  fldump_hw<<ptr_result[i]<<" ";
            fldump_hw<<endl;
            float err = mae(test+j, ptr_result);
            if(err > threshold)
                  cout<<data_test[0][j+TIMESTEPS]<<" "<<err<<endl;

            float res1[UNITS1], res2[TIMESTEPS][UNITS2],
res[TIMESTEPS][DATA_DIM2];
            LSTM_1(test+j, res1);
            LSTM_2(res1, res2);
            for(int i=0; i<TIMESTEPS; ++i)
            {
                  Dense(res2[i], res[i]);
            }
            for(int i=0; i<TIMESTEPS; ++i)
                  fldump_sw<<res[i][0]<<" ";
```

```
            fldump_sw<<endl;
        }
        fldump_hw.close();
        fldump_sw.close();

        q.enqueueUnmapMemObject(buffer_data , ptr_data);
        q.enqueueUnmapMemObject(buffer_result , ptr_result);
        q.finish();

        return 0;
}
```

read_csv.h

```
#ifndef _READ_CSV_H_
#define _READ_CSV_H_

#include <fstream>
#include <vector>
#include <sstream>

std::vector<std::vector<std::string>> read_csv(std::string file, int
low, int high, const std::vector<int> &columns);

#endif
```

read_csv.cpp

```
#include "read_csv.h"

using namespace std;

vector<vector<string>> read_csv(string file, int low, int high, const
vector<int> &columns)
{
    fstream fin;
    fin.open(file, ios::in);

    int rownum=0;
    vector<vector<string>> data(columns.size());
    vector<string> row;
    string line, word;

    while (getline(fin, line)) {

        row.clear();
```

```
        stringstream s(line);

        while (getline(s, word, ','))
            row.push_back(word);

        if(rownum++ < low)
            continue;
        else if(rownum-1 > high)
            break;

        for(unsigned i=0; i<columns.size(); ++i)
            data[i].push_back(row[columns[i]]);
    }

    fin.close();
    return data;
}
```

standard_scaler.h

```
#ifndef _STANDARD_SCALER_H_
#define _STANDARD_SCALER_H_

#include <vector>
#include <cmath>

class StandardScaler {
public:
      StandardScaler();

      void fit(const std::vector<float> &data);

      void transform(const std::vector<float> &data, float res[][1]);

private:
      float u, s;
};

#endif
```

standard_scaler.cpp

```
#include "standard_scaler.h"

using namespace std;

StandardScaler::StandardScaler() {}
```

```
void StandardScaler::fit(const vector<float> &data)
{
	u = 0.0;
	for(auto it=data.begin(); it!=data.end(); it++)
		u += *it;
	u /= data.size();

	s = 0.0;
	for(auto it=data.begin(); it!=data.end(); it++)
		s += (*it - u)*(*it - u);
	s = sqrt(s/data.size());
}

void StandardScaler::transform(const vector<float> &data, float
res[][1])
{
	for(unsigned i=0; i<data.size(); ++i)
		res[i][0] = (data[i]-u)/s;
}
```

layers.h

```
#ifndef _LSTM_H_
#define _LSTM_H_

#include <cmath>

#define TIMESTEPS 30

#define UNITS1 128
#define DATA_DIM1 1

#define UNITS2 128
#define DATA_DIM2 1

void LSTM_1(float data[TIMESTEPS][DATA_DIM1], float res[UNITS1]);

void LSTM_2(float data[UNITS1], float res[TIMESTEPS][UNITS2]);

void Dense(float data[UNITS2], float res[DATA_DIM2]);

#endif
```

layers.cpp

```
#include "layers.h"
#include "weights.h"

using namespace std;

float sigmoid(float x)
{
	return 1/(1+exp(-x));
}

void array_vector_mul(float w[], float b[], float h_x[], float
output[], int a, int c)
{
	for(int i=0; i<a; ++i)
	{
		for(int j=0; j<c; ++j)
		{
			float sum = (j==0) ? b[i] : output[i];
			output[i] = sum + w[i*c + j]*h_x[j];
		}
	}
}

void array_tanh(float x[], int n)
{
	for(int i=0; i<n; ++i)
		x[i] = tanh(x[i]);
}

void array_sigmoid(float x[], int n)
{
	for(int i=0; i<n; ++i)
		x[i] = sigmoid(x[i]);
}

void LSTM_1(float data[TIMESTEPS][DATA_DIM1], float res[UNITS1])
{
	float c[UNITS1], h[UNITS1];


	for(int i=0; i<UNITS1; ++i)
	{
		c[i] = 0.0;
		h[i] = 0.0;
	}
```

```
	for(int k=0; k<TIMESTEPS; ++k)
	{
		float f_t[UNITS1], i_t[UNITS1], c_t[UNITS1], o_t[UNITS1],
x_h[UNITS1+DATA_DIM1];

		for(int i=0; i<DATA_DIM1; ++i)
			x_h[i]=data[k][i];
		for(int i=DATA_DIM1; i<DATA_DIM1+UNITS1; ++i)
			x_h[i]=h[i-DATA_DIM1];

		array_vector_mul(Wi1[0], bi1, x_h, i_t, UNITS1,
UNITS1+DATA_DIM1);
		array_vector_mul(Wf1[0], bf1, x_h, f_t, UNITS1,
UNITS1+DATA_DIM1);
		array_vector_mul(Wc1[0], bc1, x_h, c_t, UNITS1,
UNITS1+DATA_DIM1);
		array_vector_mul(Wo1[0], bo1, x_h, o_t, UNITS1,
UNITS1+DATA_DIM1);

		array_sigmoid(i_t, UNITS1);
		array_sigmoid(f_t, UNITS1);
		array_tanh(c_t, UNITS1);
		array_sigmoid(o_t, UNITS1);

		for(int i=0; i<UNITS1; ++i)
			c[i] = f_t[i]*c[i] + c_t[i]*i_t[i];

		for(int i=0; i<UNITS1; ++i)
			h[i] = o_t[i]*tanh(c[i]);
	}
	for(int i=0; i<UNITS1; ++i)
		res[i]=h[i];
}

void LSTM_2(float data[UNITS1], float res[TIMESTEPS][UNITS2])
{
	float c[UNITS2], h[UNITS2], x_h[UNITS1+UNITS2];

	for(int i=0; i<UNITS2; ++i)
	{
		c[i] = 0.0;
		h[i] = 0.0;
	}

	for(int i=0; i<UNITS1; ++i)
		x_h[i]=data[i];
```

```
	for(int k=0; k<TIMESTEPS; ++k)
	{
		float f_t[UNITS2], i_t[UNITS2], c_t[UNITS2], o_t[UNITS2];

		for(int i=UNITS1; i<UNITS1+UNITS2; ++i)
			x_h[i]=h[i-UNITS1];

		array_vector_mul(Wi2[0], bi2, x_h, i_t, UNITS2,
UNITS1+UNITS2);
		array_vector_mul(Wf2[0], bf2, x_h, f_t, UNITS2,
UNITS1+UNITS2);
		array_vector_mul(Wc2[0], bc2, x_h, c_t, UNITS2,
UNITS1+UNITS2);
		array_vector_mul(Wo2[0], bo2, x_h, o_t, UNITS2,
UNITS1+UNITS2);

		array_sigmoid(i_t, UNITS2);
		array_sigmoid(f_t, UNITS2);
		array_tanh(c_t, UNITS2);
		array_sigmoid(o_t, UNITS2);

		for(int i=0; i<UNITS2; ++i)
			c[i] = f_t[i]*c[i] + c_t[i]*i_t[i];

		for(int i=0; i<UNITS2; ++i)
			h[i] = o_t[i]*tanh(c[i]);

		for(int i=0; i<UNITS2; ++i)
			res[k][i] = h[i];
	}
}


void Dense(float data[UNITS2], float res[DATA_DIM2])
{
	for(int i=0; i<DATA_DIM2; ++i)
	{
		for(int j=0; j<UNITS2; ++j)
		{
			float sum = (j==0) ? b[i] : res[i];
			res[i] = sum + W[i][j]*data[j];
		}
	}
}
```

kernel code

krnl_lstm_autoencoder.h

```
#ifndef _KRNL_LSTM_AUTOENCODER_H_
#define _KRNL_LSTM_AUTOENCODER_H_

#include <ap_fixed.h>
#include <ap_int.h>

#define TIMESTEPS 30

#define UNITS1 128
#define DATA_DIM1 1

#define UNITS2 128
#define DATA_DIM2 1

#define UNITS_MAX 128 //max(UNITS1, UNITS2)
#define COL_SIZE 256 //max(UNITS1+UNITS2, UNITS1+DATA_DIM1)

#define CELLS 2

#define BITS 8
#define TANH_BASE 512 //2^(BITS+1)
#define SIG_BASE 1024 //2^(BITS+2)

typedef ap_fixed<BITS+4, 4, AP_RND> fixed_point;
typedef ap_fixed<BITS+2, 2, AP_RND> fixed_point_small;

#endif
```

krnl_lstm_autoencoder.cpp

```
#include "krnl_lstm_autoencoder.h"
#include "weights.h"
#include "activations.h"

using namespace std;

fixed_point my_tanh(fixed_point x)
{
	if(x >= 2)
		return 1;
	else if(x < -2)
		return -1;
	else
	{
```

```
		ap_int<BITS+2> i = x.range();
		return tanh_vals[TANH_BASE + i.to_int()];
	}
}

fixed_point my_sigmoid(fixed_point x)
{
	if(x >= 4)
		return 1;
	else if(x < -4)
		return 0;
	else
	{
		ap_int<BITS+3> i = x.range();
		return sigmoid_vals[SIG_BASE + i.to_int()];
	}
}

void array_tanh(fixed_point x[UNITS_MAX], fixed_point y[UNITS_MAX])
{
	loop_array_tanh:for(int i=0; i<UNITS_MAX; ++i)
	{
		y[i] = my_tanh(x[i]);
	}
}

void array_sigmoid(fixed_point x[UNITS_MAX], fixed_point
y[UNITS_MAX])
{
	loop_array_sigmoid:for(int i=0; i<UNITS_MAX; ++i)
	{
		y[i] = my_sigmoid(x[i]);
	}
}

fixed_point vector_vector_mul(const fixed_point_small w[COL_SIZE],
fixed_point_small b, fixed_point h_x[COL_SIZE])
{
	fixed_point first, tmp, ret;
	loop_internal_mul:for(int j=0; j<COL_SIZE; ++j)
	{
		first = (j==0) ? (fixed_point) b : ret;
		tmp = w[j]*h_x[j];
		ret = first+tmp;
	}
	return ret;
```

```
}

void i_gate(const fixed_point_small Wi[UNITS_MAX][COL_SIZE], const
fixed_point_small bi[UNITS_MAX], fixed_point h_xi[COL_SIZE],
fixed_point out_i[UNITS_MAX])
{
#pragma HLS ARRAY_PARTITION variable=Wi dim=2 complete
#pragma HLS ARRAY_PARTITION variable=bi complete
#pragma HLS ARRAY_PARTITION variable=h_xi complete
	fixed_point i_t[UNITS_MAX];
	loop_igate:for(int i=0; i<UNITS_MAX; ++i)
	{
#pragma HLS PIPELINE
		i_t[i] = vector_vector_mul(Wi[i], bi[i], h_xi);
	}
	array_sigmoid(i_t, out_i);
}

void f_gate(const fixed_point_small Wf[UNITS_MAX][COL_SIZE], const
fixed_point_small bf[UNITS_MAX], fixed_point h_xf[COL_SIZE],
fixed_point out_f[UNITS_MAX])
{
#pragma HLS ARRAY_PARTITION variable=Wf dim=2 complete
#pragma HLS ARRAY_PARTITION variable=bf complete
#pragma HLS ARRAY_PARTITION variable=h_xf complete
	fixed_point f_t[UNITS_MAX];
	loop_fgate:for(int i=0; i<UNITS_MAX; ++i)
	{
#pragma HLS PIPELINE
		f_t[i] = vector_vector_mul(Wf[i], bf[i], h_xf);
	}
	array_sigmoid(f_t, out_f);
}

void c_gate(const fixed_point_small Wc[UNITS_MAX][COL_SIZE], const
fixed_point_small bc[UNITS_MAX], fixed_point h_xc[COL_SIZE],
fixed_point out_c[UNITS_MAX])
{
#pragma HLS ARRAY_PARTITION variable=Wc dim=2 complete
#pragma HLS ARRAY_PARTITION variable=bc complete
#pragma HLS ARRAY_PARTITION variable=h_xc complete
	fixed_point c_t[UNITS_MAX];
	loop_cgate:for(int i=0; i<UNITS_MAX; ++i)
	{
#pragma HLS PIPELINE
		c_t[i] = vector_vector_mul(Wc[i], bc[i], h_xc);
```

```
	}
	array_tanh(c_t, out_c);
}

void o_gate(const fixed_point_small Wo[UNITS_MAX][COL_SIZE], const
fixed_point_small bo[UNITS_MAX], fixed_point h_xo[COL_SIZE],
fixed_point out_o[UNITS_MAX])
{
#pragma HLS ARRAY_PARTITION variable=Wo dim=2 complete
#pragma HLS ARRAY_PARTITION variable=bo complete
#pragma HLS ARRAY_PARTITION variable=h_xo complete
	fixed_point o_t[UNITS_MAX];
	loop_ogate:for(int i=0; i<UNITS_MAX; ++i)
	{
#pragma HLS PIPELINE
		o_t[i] = vector_vector_mul(Wo[i], bo[i], h_xo);
	}
	array_sigmoid(o_t, out_o);
}

void Dense(const fixed_point_small W[DATA_DIM2][UNITS2], const
fixed_point_small b, fixed_point data[UNITS2], fixed_point
res[DATA_DIM2])
{
#pragma HLS ARRAY_PARTITION variable=W dim=2 complete
#pragma HLS ARRAY_PARTITION variable=data complete
	fixed_point first, tmp, ret;
	loop_dense:for(int i=0; i<UNITS2; ++i)
	{
#pragma HLS PIPELINE
		first = (i==0) ? (fixed_point) b : ret;
		tmp = W[0][i]*data[i];
		ret = first+tmp;
	}
	res[0] = ret;
}

extern "C" {
void krnl_lstm_autoencoder(const float *data, float *res)
{
#pragma HLS INTERFACE m_axi port=data offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=res offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=data bundle=control
#pragma HLS INTERFACE s_axilite port=res bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control
```

```
	fixed_point input[TIMESTEPS][DATA_DIM1],
output[TIMESTEPS][DATA_DIM2];

	loop_read_input:for(int i=0; i<TIMESTEPS; ++i)
		input[i][0] = data[i];

	fixed_point c[UNITS_MAX], h[UNITS_MAX];
	fixed_point c_f[UNITS_MAX], c_s[UNITS_MAX], c_tmp[UNITS_MAX],
c_tanh_tmp[UNITS_MAX], h_tmp[UNITS_MAX];
	fixed_point x_h[COL_SIZE];
	fixed_point i_t2[UNITS_MAX], f_t2[UNITS_MAX], c_t2[UNITS_MAX],
o_t2[UNITS_MAX];

	loop_entry1:for(int i=0; i<UNITS_MAX; ++i)
	{
		c[i] = 0.0;
		h[i] = 0.0;
	}

	loop_entry2:for(int i=0; i<COL_SIZE; ++i)
	{
		x_h[i] = 0.0;
	}

	loop_cells:for(int cell=0; cell<CELLS; ++cell)
	{
		loop_timesteps:for(int k=0; k<TIMESTEPS; ++k)
		{
			if(cell==0)
			{
				x_h[0] = input[k][0];
			}

			if(cell==0)
			{
				i_gate(Wi1, bi1, x_h, i_t2);
				f_gate(Wf1, bf1, x_h, f_t2);
				c_gate(Wc1, bc1, x_h, c_t2);
				o_gate(Wo1, bo1, x_h, o_t2);
			}
			else
			{
				i_gate(Wi2, bi2, x_h, i_t2);
				f_gate(Wf2, bf2, x_h, f_t2);
				c_gate(Wc2, bc2, x_h, c_t2);
				o_gate(Wo2, bo2, x_h, o_t2);
```

```
                }

                loop_cell_internal:for(int i=0; i<UNITS_MAX; ++i)
                {
#pragma HLS PIPELINE
                        c_f[i] = f_t2[i]*c[i];
                        c_s[i] = c_t2[i]*i_t2[i];
                        c_tmp[i] = c_f[i]+c_s[i];
                        c_tanh_tmp[i] = my_tanh(c_tmp[i]);
                        h_tmp[i] = o_t2[i]*c_tanh_tmp[i];
                        c[i] = c_tmp[i];
                        h[i] = h_tmp[i];
                        if(cell==0)
                        {
                                x_h[i+DATA_DIM1] = h_tmp[i];
                        }
                        else
                        {
                                x_h[i+UNITS1] = h_tmp[i];
                        }
                }
                if(cell==1)
                {
                        Dense(W, b[0], h_tmp, output[k]);
                }
        }
        loop_out:for(int i=0; i<COL_SIZE; ++i)
        {
                x_h[i] = (i<UNITS_MAX) ? h[i] : (fixed_point) 0.0;
                if(i<UNITS_MAX)
                {
                        h[i] = 0.0;
                        c[i] = 0.0;
                }
        }
    }
    loop_write_output:for(int i=0; i<TIMESTEPS; ++i)
        res[i] = output[i][0].to_float();
}
}
```

C++ FPGA code for NAB dataset

host code

main.cpp

```
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <algorithm>
#include <ctime>
#include <numeric>
#define CL_HPP_CL_1_2_DEFAULT_BUILD
#define CL_HPP_TARGET_OPENCL_VERSION 120
#define CL_HPP_MINIMUM_OPENCL_VERSION 120
#define CL_HPP_ENABLE_PROGRAM_CONSTRUCTION_FROM_ARRAY_COMPATIBILITY 1
#include <CL/cl2.hpp>
#include "read_csv.h"
#include "layers.h"

using namespace std;

float timestepToDay(string t)
{
	tm date = {0, 0, 0, stoi(t.substr(8, 2)), stoi(t.substr(5,
2))-1, stoi(t.substr(0,4))-1900};
	time_t time_temp = mktime(&date);
	const tm * time_out = localtime(&time_temp);
	return (float) ((time_out->tm_wday + 6) % 7);
}

void StandardScaler(std::vector<float> &data)
{
	float u = accumulate(data.begin(), data.end(), 0.0, [](float a,
float b){return a + b;}) / data.size();
	float s = sqrt(accumulate(data.begin(), data.end(), 0.0,
[u](float a, float b){return a + (b-u)*(b-u);}) / data.size());

	transform(data.begin(), data.end(), data.begin(), [u, s](float
d){return (d-u)/s;});
}

int main(int argc, char* argv[])
{
	vector<vector<string>> data =
read_csv("ambient_temperature_system_failure.csv", 1, 7268, {0, 1});
```

```
	vector<float> values(data[1].size());
	transform(data[1].begin(), data[1].end(), values.begin(),
[](string s){return (stof(s)-32)*5/9;});

	vector<float> hours(data[0].size());
	transform(data[0].begin(), data[0].end(), hours.begin(),
[](string s){return stof(s.substr(11, 2));});

	vector<float> daylight(hours.size());
	transform(hours.begin(), hours.end(), daylight.begin(),
[](float t){return (t>=6.9 && t<=22.1) ? 1.0 : 0.0;});

	vector<float> day(daylight.size());
	transform(data[0].begin(), data[0].end(), day.begin(),
timestepToDay);

	vector<float> weekday(day.size());
	transform(day.begin(), day.end(), weekday.begin(), [](float
d){return (d<4.9) ? 1.0: 0.0;});

	StandardScaler(values);
	StandardScaler(hours);
	StandardScaler(daylight);
	StandardScaler(day);
	StandardScaler(weekday);

	float test[1049][5];
	for(int i=0; i<1049; ++i)
	{
		test[i][0] = values[i+6216];
		test[i][1] = hours[i+6216];
		test[i][2] = daylight[i+6216];
		test[i][3] = day[i+6216];
		test[i][4] = weekday[i+6216];
	}

	//TARGET_DEVICE macro needs to be passed from gcc command line
	if(argc != 2) {
		std::cout << "Usage: " << argv[0] <<" <xclbin>" <<
std::endl;
		return EXIT_FAILURE;
	}

	char* xclbinFilename = argv[1];

	// Compute the size of array in bytes
```

```
	size_t size_in_bytes = TIMESTEPS * DATA_DIM1 * sizeof(float);
	size_t size_out_bytes = DATA_DIM2 * sizeof(float);

	std::vector<cl::Device> devices;
	cl::Device device;
	std::vector<cl::Platform> platforms;
	bool found_device = false;

	//traversing all Platforms To find Xilinx Platform and targeted
	//Device in Xilinx Platform
	cl::Platform::get(&platforms);
	for(size_t i = 0; (i < platforms.size() ) & (found_device ==
false) ;i++){
		cl::Platform platform = platforms[i];
		std::string platformName =
platform.getInfo<CL_PLATFORM_NAME>();
		if ( platformName == "Xilinx"){
			devices.clear();
			platform.getDevices(CL_DEVICE_TYPE_ACCELERATOR,
&devices);
			if (devices.size()){
				device = devices[0];
				found_device = true;
				break;
			}
		}
	}
	if (found_device == false){
		std::cout << "Error: Unable to find Target Device "
		<< device.getInfo<CL_DEVICE_NAME>() << std::endl;
		return EXIT_FAILURE;
	}

	// Creating Context and Command Queue for selected device
	cl::Context context(device);
	cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE);

	// Load xclbin
	std::cout << "Loading: '" << xclbinFilename << "'\n";
	std::ifstream bin_file(xclbinFilename, std::ifstream::binary);
	bin_file.seekg (0, bin_file.end);
	unsigned nb = bin_file.tellg();
	bin_file.seekg (0, bin_file.beg);
	char *buf = new char [nb];
	bin_file.read(buf, nb);
```

```
	// Creating Program from Binary File
	cl::Program::Binaries bins;
	bins.push_back({buf,nb});
	devices.resize(1);
	cl::Program program(context, devices, bins);

	// This call will get the kernel object from program. A kernel
is an
	// OpenCL function that is executed on the FPGA.
	cl::Kernel krnl_lstm(program,"krnl_lstm");

	// These commands will allocate memory on the Device. The
cl::Buffer objects can
	// be used to reference the memory locations on the device.
	cl::Buffer buffer_data(context, CL_MEM_READ_ONLY,
size_in_bytes);
	cl::Buffer buffer_result(context, CL_MEM_WRITE_ONLY,
size_out_bytes);

	//set the kernel Arguments
	int narg=0;
	krnl_lstm.setArg(narg++,buffer_data);
	krnl_lstm.setArg(narg++,buffer_result);

	//We then need to map our OpenCL buffers to get the pointers
	float *ptr_data = (float *) q.enqueueMapBuffer (buffer_data ,
CL_TRUE , CL_MAP_WRITE , 0, size_in_bytes);
	float *ptr_result = (float *) q.enqueueMapBuffer (buffer_result
, CL_TRUE , CL_MAP_READ , 0, size_out_bytes);

	float res[1000][1], res_hw[1000][1];
      ofstream fldump_hw, fldump_sw;
      fldump_hw.open("hw_results8.dat");
      fldump_sw.open("sw_results.dat");
	for(int i=0; i<1000; ++i)
	{
		//setting input data
		for(int j=0; j< TIMESTEPS; ++j)
		{
			for(int k=0; k<DATA_DIM1; ++k)
			{
				ptr_data[j*DATA_DIM1 + k] = test[i+j][k];
			}
		}

		// Data will be migrated to kernel space
```

```
            q.enqueueMigrateMemObjects({buffer_data},0/* 0 means from
host*/);

            //Launch the Kernel
            q.enqueueTask(krnl_lstm);

            // The result of the previous kernel execution will need
to be retrieved in
            // order to view the results. This call will transfer the
data from FPGA to
            // source_results vector

q.enqueueMigrateMemObjects({buffer_result},CL_MIGRATE_MEM_OBJECT_HOST
);

            q.finish();

            // Print result
            fldump_hw<<ptr_result[0]<<'\n';
            res_hw[i][0] = ptr_result[0];


            float res1[TIMESTEPS][UNITS1], res2[UNITS2];
            LSTM_1(test+i, res1);
            LSTM_2(res1, res2);
            Dense(res2, res[i]);
            fldump_sw<<res[i][0]<<'\n';
      }
    fldump_hw.close();
    fldump_sw.close();

      pair<float, int> diff[1000];
      for(int i=0; i<1000; ++i)
            diff[i] = make_pair(abs(values[i+6267]-res_hw[i][0]), i);

      int number_of_outliers = 10; //outlier_fraction=0.01,
values=1000 -> 0.01*1000=10
      nth_element(diff, diff+989, diff+1000);
      sort(diff+990, diff+1000, [](pair<float, int> a, pair<float,
int> b){return a.second < b.second;});

      cout<<"ANOMALY POINTS:\n";
      for(int i=0; i<number_of_outliers; ++i)
      {
            int idx = diff[i+990].second;
            cout<<data[0][idx+6267]<<' '<<diff[i+990].first<<endl;
```

```
        }

        q.enqueueUnmapMemObject(buffer_data , ptr_data);
        q.enqueueUnmapMemObject(buffer_result , ptr_result);
        q.finish();

        return 0;
}
```

read_csv.h

```
#ifndef _READ_CSV_H_
#define _READ_CSV_H_

#include <fstream>
#include <vector>
#include <sstream>

std::vector<std::vector<std::string>> read_csv(std::string file, int
low, int high, const std::vector<int> &columns);

#endif
```

read_csv.cpp

```
#include "read_csv.h"

using namespace std;

vector<vector<string>> read_csv(string file, int low, int high, const
vector<int> &columns)
{
    fstream fin;
    fin.open(file, ios::in);

    int rownum=0;
    vector<vector<string>> data(columns.size());
    vector<string> row;
    string line, word;

    while (getline(fin, line)) {

        row.clear();

        stringstream s(line);

        while (getline(s, word, ','))
```

```
            row.push_back(word);

        if(rownum++ < low)
            continue;
        else if(rownum-1 > high)
            break;

        for(unsigned i=0; i<columns.size(); ++i)
            data[i].push_back(row[columns[i]]);
    }

    fin.close();
    return data;
}
```

layers.h

```
#ifndef _LSTM_H_
#define _LSTM_H_

#include <cmath>

#define TIMESTEPS 50

#define UNITS1 50
#define DATA_DIM1 5

#define UNITS2 100
#define DATA_DIM2 1


void LSTM_1(float data[TIMESTEPS][DATA_DIM1], float
res[TIMESTEPS][UNITS1]);

void LSTM_2(float data[TIMESTEPS][UNITS1], float res[UNITS2]);

void Dense(float data[UNITS2], float res[DATA_DIM2]);

#endif
```

layers.cpp

```
#include "layers.h"
#include "weights.h"

using namespace std;
```

```
float sigmoid(float x)
{
	return 1/(1+exp(-x));
}

void array_vector_mul(float w[], float b[], float h_x[], float
output[], int a, int c)
{
	for(int i=0; i<a; ++i)
	{
		for(int j=0; j<c; ++j)
		{
			float sum = (j==0) ? b[i] : output[i];
			output[i] = sum + w[i*c + j]*h_x[j];
		}
	}
}

void array_tanh(float x[], int n)
{
	for(int i=0; i<n; ++i)
		x[i] = tanh(x[i]);
}

void array_sigmoid(float x[], int n)
{
	for(int i=0; i<n; ++i)
		x[i] = sigmoid(x[i]);
}

void LSTM_1(float data[TIMESTEPS][DATA_DIM1], float
res[TIMESTEPS][UNITS1])
{
	float c[UNITS1], h[UNITS1];

	for(int i=0; i<UNITS1; ++i)
	{
		c[i] = 0.0;
		h[i] = 0.0;
	}

	for(int k=0; k<TIMESTEPS; ++k)
	{
		float f_t[UNITS1], i_t[UNITS1], c_t[UNITS1], o_t[UNITS1],
x_h[UNITS1+DATA_DIM1];
```

```
            for(int i=0; i<DATA_DIM1; ++i)
                  x_h[i]=data[k][i];
            for(int i=DATA_DIM1; i<DATA_DIM1+UNITS1; ++i)
                  x_h[i]=h[i-DATA_DIM1];

            array_vector_mul(Wi1[0], bi1, x_h, i_t, UNITS1,
UNITS1+DATA_DIM1);
            array_vector_mul(Wf1[0], bf1, x_h, f_t, UNITS1,
UNITS1+DATA_DIM1);
            array_vector_mul(Wc1[0], bc1, x_h, c_t, UNITS1,
UNITS1+DATA_DIM1);
            array_vector_mul(Wo1[0], bo1, x_h, o_t, UNITS1,
UNITS1+DATA_DIM1);

            array_sigmoid(i_t, UNITS1);
            array_sigmoid(f_t, UNITS1);
            array_tanh(c_t, UNITS1);
            array_sigmoid(o_t, UNITS1);

            for(int i=0; i<UNITS1; ++i)
                  c[i] = f_t[i]*c[i] + c_t[i]*i_t[i];

            for(int i=0; i<UNITS1; ++i)
                  h[i] = o_t[i]*tanh(c[i]);

            for(int i=0; i<UNITS1; ++i)
                  res[k][i]=h[i];
      }
}

void LSTM_2(float data[TIMESTEPS][UNITS1], float res[UNITS2])
{
      float c[UNITS2], h[UNITS2];

      for(int i=0; i<UNITS2; ++i)
      {
            c[i] = 0.0;
            h[i] = 0.0;
      }

      for(int k=0; k<TIMESTEPS; ++k)
      {
            float f_t[UNITS2], i_t[UNITS2], c_t[UNITS2], o_t[UNITS2],
x_h[UNITS1+UNITS2];

            for(int i=0; i<UNITS1; ++i)
```

```
                x_h[i]=data[k][i];
            for(int i=UNITS1; i<UNITS1+UNITS2; ++i)
                x_h[i]=h[i-UNITS1];

            array_vector_mul(Wi2[0], bi2, x_h, i_t, UNITS2,
UNITS1+UNITS2);
            array_vector_mul(Wf2[0], bf2, x_h, f_t, UNITS2,
UNITS1+UNITS2);
            array_vector_mul(Wc2[0], bc2, x_h, c_t, UNITS2,
UNITS1+UNITS2);
            array_vector_mul(Wo2[0], bo2, x_h, o_t, UNITS2,
UNITS1+UNITS2);

            array_sigmoid(i_t, UNITS2);
            array_sigmoid(f_t, UNITS2);
            array_tanh(c_t, UNITS2);
            array_sigmoid(o_t, UNITS2);

            for(int i=0; i<UNITS2; ++i)
                c[i] = f_t[i]*c[i] + c_t[i]*i_t[i];

            for(int i=0; i<UNITS2; ++i)
                h[i] = o_t[i]*tanh(c[i]);
        }
        for(int i=0; i<UNITS2; ++i)
            res[i] = h[i];
}


void Dense(float data[UNITS2], float res[DATA_DIM2])
{
        for(int i=0; i<DATA_DIM2; ++i)
        {
            for(int j=0; j<UNITS2; ++j)
            {
                float sum = (j==0) ? b[i] : res[i];
                res[i] = sum + W[i][j]*data[j];
            }
        }
}
```

kernel code

krnl_lstm.h

```
#ifndef _KRNL_LSTM_H_
```

```
#define _KRNL_LSTM_H_

#include <ap_fixed.h>
#include <ap_int.h>

#define TIMESTEPS 50

#define UNITS1 50
#define DATA_DIM1 5

#define UNITS2 100
#define DATA_DIM2 1

#define UNITS_MAX 100 //max(UNITS1, UNITS2)
#define COL_SIZE 150 //max(UNITS1+UNITS2, UNITS1+DATA_DIM1)

#define CELLS 2

#define BITS 8
#define TANH_BASE 512 //2^(BITS+1)
#define SIG_BASE 1024 //2^(BITS+2)

typedef ap_fixed<BITS+3, 3, AP_RND> fixed_point;
typedef ap_fixed<BITS+2, 2, AP_RND> fixed_point_small;

#endif
```

krnl_lstm.cpp

```
#include "krnl_lstm.h"
#include "weights.h"
#include "activations.h"

using namespace std;

fixed_point my_tanh(fixed_point x)
{
	if(x >= 2)
		return 1;
	else if(x < -2)
		return -1;
	else
	{
		ap_int<BITS+2> i = x.range();
		return tanh_vals[TANH_BASE + i.to_int()];
	}
```

```
}

fixed_point my_sigmoid(fixed_point x)
{
	if(x >= 4)
		return 1;
	else if(x < -4)
		return 0;
	else
	{
		ap_int<BITS+3> i = x.range();
		return sigmoid_vals[SIG_BASE + i.to_int()];
	}
}

void array_tanh(fixed_point x[UNITS_MAX], fixed_point y[UNITS_MAX])
{
	loop_array_tanh:for(int i=0; i<UNITS_MAX; ++i)
		y[i] = my_tanh(x[i]);
}

void array_sigmoid(fixed_point x[UNITS_MAX], fixed_point
y[UNITS_MAX])
{
	loop_array_sigmoid:for(int i=0; i<UNITS_MAX; ++i)
		y[i] = my_sigmoid(x[i]);
}

fixed_point vector_vector_mul(const fixed_point_small w[COL_SIZE],
fixed_point_small b, fixed_point h_x[COL_SIZE])
{
	fixed_point first, tmp, ret;
	loop_internal_mul:for(int j=0; j<COL_SIZE; ++j)
	{
		first = (j==0) ? (fixed_point) b : ret;
		tmp = w[j]*h_x[j];
		ret = first+tmp;
	}
	return ret;
}

void i_gate(const fixed_point_small Wi[UNITS_MAX][COL_SIZE], const
fixed_point_small bi[UNITS_MAX], fixed_point h_xi[COL_SIZE],
fixed_point out_i[UNITS_MAX])
{
#pragma HLS ARRAY_PARTITION variable=Wi dim=2 complete
```

```
#pragma HLS ARRAY_PARTITION variable=bi complete
#pragma HLS ARRAY_PARTITION variable=h_xi complete
      fixed_point i_t[UNITS_MAX];
      loop_igate:for(int i=0; i<UNITS_MAX; ++i)
      {
#pragma HLS PIPELINE
            i_t[i] = vector_vector_mul(Wi[i], bi[i], h_xi);
      }
      array_sigmoid(i_t, out_i);
}

void f_gate(const fixed_point_small Wf[UNITS_MAX][COL_SIZE], const
fixed_point_small bf[UNITS_MAX], fixed_point h_xf[COL_SIZE],
fixed_point out_f[UNITS_MAX])
{
#pragma HLS ARRAY_PARTITION variable=Wf dim=2 complete
#pragma HLS ARRAY_PARTITION variable=bf complete
#pragma HLS ARRAY_PARTITION variable=h_xf complete
      fixed_point f_t[UNITS_MAX];
      loop_fgate:for(int i=0; i<UNITS_MAX; ++i)
      {
#pragma HLS PIPELINE
            f_t[i] = vector_vector_mul(Wf[i], bf[i], h_xf);
      }
      array_sigmoid(f_t, out_f);
}

void c_gate(const fixed_point_small Wc[UNITS_MAX][COL_SIZE], const
fixed_point_small bc[UNITS_MAX], fixed_point h_xc[COL_SIZE],
fixed_point out_c[UNITS_MAX])
{
#pragma HLS ARRAY_PARTITION variable=Wc dim=2 complete
#pragma HLS ARRAY_PARTITION variable=bc complete
#pragma HLS ARRAY_PARTITION variable=h_xc complete
      fixed_point c_t[UNITS_MAX];
      loop_cgate:for(int i=0; i<UNITS_MAX; ++i)
      {
#pragma HLS PIPELINE
            c_t[i] = vector_vector_mul(Wc[i], bc[i], h_xc);
      }
      array_tanh(c_t, out_c);
}

void o_gate(const fixed_point_small Wo[UNITS_MAX][COL_SIZE], const
fixed_point_small bo[UNITS_MAX], fixed_point h_xo[COL_SIZE],
fixed_point out_o[UNITS_MAX])
```

```
{
#pragma HLS ARRAY_PARTITION variable=Wo dim=2 complete
#pragma HLS ARRAY_PARTITION variable=bo complete
#pragma HLS ARRAY_PARTITION variable=h_xo complete
      fixed_point o_t[UNITS_MAX];
      loop_ogate:for(int i=0; i<UNITS_MAX; ++i)
      {
#pragma HLS PIPELINE
            o_t[i] = vector_vector_mul(Wo[i], bo[i], h_xo);
      }
      array_sigmoid(o_t, out_o);
}

void Dense(const fixed_point_small W[DATA_DIM2][UNITS2], const
fixed_point_small b, fixed_point data[UNITS2], fixed_point
res[DATA_DIM2])
{
#pragma HLS ARRAY_PARTITION variable=W dim=2 complete
#pragma HLS ARRAY_PARTITION variable=data complete
      fixed_point first, tmp, ret;
      loop_dense:for(int i=0; i<UNITS2; ++i)
      {
#pragma HLS PIPELINE
            first = (i==0) ? (fixed_point) b : ret;
            tmp = W[0][i]*data[i];
            ret = first+tmp;
      }
      res[0] = ret;
}

extern "C" {
void krnl_lstm(const float *input, float *res)
{
#pragma HLS INTERFACE m_axi port=input offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=res offset=slave bundle=gmem1
#pragma HLS INTERFACE s_axilite port=input bundle=control
#pragma HLS INTERFACE s_axilite port=res bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

      fixed_point data[TIMESTEPS][DATA_DIM1], output[1];
      loop_read_input_out:for(int i=0; i<TIMESTEPS; ++i)
            loop_read_input_in:for(int j=0; j<DATA_DIM1; ++j)
                  data[i][j] = input[i*DATA_DIM1 + j];

      fixed_point c[UNITS_MAX], h[UNITS_MAX];
      fixed_point c_f[UNITS_MAX], c_s[UNITS_MAX], c_tmp[UNITS_MAX],
```

```
c_tanh_tmp[UNITS_MAX], h_tmp[UNITS_MAX];
	fixed_point x_h[COL_SIZE];
	fixed_point f_t2[UNITS_MAX], i_t2[UNITS_MAX], c_t2[UNITS_MAX],
o_t2[UNITS_MAX];
	fixed_point data2[TIMESTEPS][UNITS1];

	loop_cells:for(int cell=0; cell<CELLS; ++cell)
	{
		loop_entry:for(int i=0; i<COL_SIZE; ++i)
		{
			x_h[i] = 0.0;
			if(i<UNITS_MAX)
			{
				h[i] = 0.0;
				c[i] = 0.0;
			}
		}
		loop_timesteps:for(int k=0; k<TIMESTEPS; ++k)
		{
			if(cell==0)
			{
				for(int i=0; i<DATA_DIM1; ++i)
					x_h[i]=data[k][i];
			}
			else
			{
				for(int i=0; i<UNITS1; ++i)
					x_h[i]=data2[k][i];
			}

			if(cell==0)
			{
				i_gate(Wi1, bi1, x_h, i_t2);
				f_gate(Wf1, bf1, x_h, f_t2);
				c_gate(Wc1, bc1, x_h, c_t2);
				o_gate(Wo1, bo1, x_h, o_t2);
			}
			else
			{
				i_gate(Wi2, bi2, x_h, i_t2);
				f_gate(Wf2, bf2, x_h, f_t2);
				c_gate(Wc2, bc2, x_h, c_t2);
				o_gate(Wo2, bo2, x_h, o_t2);
			}

			loop_cell_internal:for(int i=0; i<UNITS_MAX; ++i)
```

```
			{
#pragma HLS PIPELINE
				c_f[i] = f_t2[i]*c[i];
				c_s[i] = c_t2[i]*i_t2[i];
				c_tmp[i] = c_f[i]+c_s[i];
				c_tanh_tmp[i] = my_tanh(c_tmp[i]);
				h_tmp[i] = o_t2[i]*c_tanh_tmp[i];
				c[i] = c_tmp[i];
				h[i] = h_tmp[i];

				if(cell==0)
				{
					x_h[i+DATA_DIM1] = h_tmp[i];
				}
				else
				{
					x_h[i+UNITS1] = h_tmp[i];
				}
			}

			if(cell==0)
			{
				loop_out:for(int i=0; i<UNITS1; ++i)
					data2[k][i]=h[i];
			}
		}
	}
	Dense(W, b[0], h, output);
	res[0] = output[0].to_float();
}
}
```

References

[1] Braei, Mohammad & Wagner, Sebastian. (2020). Anomaly Detection in Univariate Time-series: A Survey on the State-of-the-Art.
[2] *What is Anomaly Detection? Definition & FAQs | Avi Networks*. Available at: https://avinetworks.com/glossary/anomaly-detection/
[3] Y. Guan, Z. Yuan, G. Sun and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), 2017, pp. 629-634, doi: 10.1109/ASPDAC.2017.7858394.
[4] *What Is the Definition of Machine Learning?*. Available at: https://www.expert.ai/blog/machine-learning-definition/
[5] *What is Machine Learning? | IBM*. Available at: https://www.ibm.com/cloud/learn/machine-learning
[6] *What Is Machine Learning and Why Is It Important?*. Available at: https://www.techtarget.com/searchenterpriseai/definition/machine-learning-ML
[7] Haykin, S. S. (2009), Neural networks and learning machines, Third Edition. Pearson Education, Upper Saddle River, NJ
[8] *CS231n Convolutional Neural Networks for Visual Recognition*. Available at: https://cs231n.github.io/neural-networks-1/
[9] *Understanding LSTM Networks -- colah's blog*. Available at: http://colah.github.io/posts/2015-08-Understanding-LSTMs/
[10] Hochreiter, Sepp & Schmidhuber, Jürgen. (1997). Long Short-term Memory. Neural computation. 9. 1735-80. 10.1162/neco.1997.9.8.1735.
[11] *Animated RNN, LSTM and GRU. Recurrent neural network cells in GIFs | by Raimi Karim | Towards Data Science*. Available at: https://towardsdatascience.com/animated-rnn-lstm-and-gru-ef124d06cf45
[12] Maxfield, C. (2004) The Design Warrior's Guide to FPGAs: Devices, Tools and Flows. Newnes, USA
[13] *ug998-vivado-intro-fpga-design-hls.pdf • Viewer • Documentation Portal*. Available at: https://docs.xilinx.com/v/u/en-US/ug998-vivado-intro-fpga-design-hls
[14] S. Gandhare and B. Karthikeyan, "Survey on FPGA Architecture and Recent Applications," 2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN), 2019, pp. 1-4, doi: 10.1109/ViTECoN.2019.8899550.
[15] *Logic block - Wikipedia*. Available at: https://en.wikipedia.org/wiki/Logic_block
[16] *Compare Benefits of CPUs, GPUs, and FPGAs for Different oneAPI*. Available at: https://www.intel.com/content/www/us/en/developer/articles/technical/comparing-cpus-gpus-and-fpgas-for-oneapi.html#gs.4xjkew
[17] *FPGA vs. GPU vs. CPU – hardware options for AI applications | Avnet Silica*. Available at: https://www.avnet.com/wps/portal/silica/resources/article/fpga-vs-gpu-vs-cpu-hardware-options-for-ai-applications/
[18] S. Asano, T. Maruyama and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," 2009 International Conference on Field

Programmable Logic and Applications, 2009, pp. 126-131, doi: 10.1109/FPL.2009.5272532.
[19] T. Riesgo, Y. Torroja and E. de la Torre, "Design methodologies based on hardware description languages," in IEEE Transactions on Industrial Electronics, vol. 46, no. 1, pp. 3-12, Feb. 1999, doi: 10.1109/41.744370.
[20] *ug902-vivado-high-level-synthesis.pdf • Viewer • Documentation Portal*. Available at: https://docs.xilinx.com/v/u/2018.3-English/ug902-vivado-high-level-synthesis
[21] *The Vitis Unified Software Platform: What We Learned at XDF – Digilent Blog*. Available at: https://digilent.com/blog/what-we-learned-at-xdf/
[22] *Vitis Software Platform*. Available at: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html
[23] *Getting Started with Vitis • Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393) • Reader • Documentation Portal*. Available at: https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Getting-Started-with-Vitis
[24] *TensorFlow*. Available at: https://www.tensorflow.org/
[25] *TensorFlow is in a relationship with Keras — Introducing TF 2.0 | by Muhammad Zaid | Towards Data Science*. Available at: https://towardsdatascience.com/tensorflow-is-in-a-relationship-with-keras-introducing-tf-2-0-dcf1228f73ae
[26] *Why TensorFlow*. Available at: https://www.tensorflow.org/about
[27] *About Keras*. Available at: https://keras.io/about/
[28] *TensorFlow 2 Tutorial: Get Started in Deep Learning With tf.keras*. Available at: https://machinelearningmastery.com/tensorflow-tutorial-deep-learning-with-tf-keras/
[29] *Alveo U280 Data Center Accelerator Card*. Available at: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html
[30] *Overview • Alveo Data Center Accelerator Card Platforms User Guide (UG1120) • Reader • Documentation Portal*. Available at: https://docs.xilinx.com/r/en-US/Alveo-Data-Center-Accelerator-Card-Platforms-User-Guide-UG1120/Overview
[31] *Alveo U280 Data Center Accelerator Card Data Sheet*. Available at: https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds963-u280.pdf
[32] *Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit*. Available at: https://www.xilinx.com/products/boards-and-kits/zcu104.html
[33] *ug1085-zynq-ultrascale-trm.pdf • Viewer • Documentation Portal*. Available at: https://docs.xilinx.com/v/u/en-US/ug1085-zynq-ultrascale-trm
[34] *ZCU104 Evaluation Board User Guide (UG1267)*. Available at: https://www.xilinx.com/support/documents/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf
[35] *tf.keras.Sequential | TensorFlow Core v2.9.0*. Available at: https://www.tensorflow.org/api_docs/python/tf/keras/Sequential
[36] *tf.keras.layers.Dropout | TensorFlow Core v2.9.0*. Available at: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout
[37] *tf.keras.layers.TimeDistributed | TensorFlow Core v2.9.0*. Available at: https://www.tensorflow.org/api_docs/python/tf/keras/layers/TimeDistributed

[38] *Getting Started with Vitis HLS • Vitis High-Level Synthesis User Guide (UG1399) • Reader • Documentation Portal*. Available at:
https://docs.xilinx.com/r/en-US/ug1399-vitis-hls
[39] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Sérot and François Berry, F. 2018. Accelerating CNN inference on FPGAs: A Survey. arXiv: arXiv:1806.01683
[40] Matthieu Courbariaux, Yoshua Bengio, Jean-Pierre David. 2015. Training deep neural networks with low precision multiplications. Accepted as a workshop contribution at ICLR 2015. arXiv: arXiv:1412.7024
[41] *Johnson & Johnson (JNJ) Stock Historical Prices & Data - Yahoo Finance*. Available at:
https://finance.yahoo.com/quote/JNJ/history?period1=494640000&period2=1599091200&interval=1d&filter=history&frequency=1d&includeAdjustedClose=true
[42] *Applied Deep Learning - Part 3: Autoencoders | by Arden Dertat | Towards Data Science*. Available at:
https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798
[43] *Time Series of Price Anomaly Detection with LSTM | by Susan Li | Towards Data Science*. Available at:
https://towardsdatascience.com/time-series-of-price-anomaly-detection-with-lstm-11a12ba4f6d9
[44] *Numenta Anomaly Benchmark (NAB) | Kaggle*. Available at:
https://www.kaggle.com/datasets/boltzmannbrain/nab
[45] *Anomaly Detection in Temperature Sensor Data using LSTM RNN Model*. Available at:
https://analyticsindiamag.com/anomaly-detection-in-temperature-sensor-data-using-lstm-rnn-model/