ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Mapping, Characterization and Acceleration of Apache Spark Applications

Ιωάννης Γ. Σταμέλος

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Αθήνα, Ιούλιος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Mapping, Characterization and Acceleration of Apache Spark Applications

Ιωάννης Γ. Σταμέλος

Επιβλέπων: Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 4η Ιουλίου 2017.

.....	.....	.....
Δ. Σούντρης	Κ. Πεκμεστζή	Γ. Γκούμας
Αναπληρωτής Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής

Αθήνα, Ιούλιος 2017

.....

Ιωάννης Γ. Σταμέλος

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

Ευχαριστίες

Θα ήθελα να ευχαριστήσω για την εκπόνηση της παρούσης εργασίας τον εισηγητή και επιβλέποντα καθηγητή κ. Σούντρη Δημήτριο, για την βοήθεια και την υποστήριξη που μου παρείχε. Θα ήθελα επίσης να ευχαριστήσω τον κ. Κάχρη Χριστόφορο, για την υποστήριξη και την καθοδήγηση που μου παρείχε, ιδιαίτερα στο τεχνικό μέρος και στην υλοποίηση της εργασίας. Για τη συμμετοχή τους στην τριμελή επιτροπή, θα ήθελα να ευχαριστήσω τους κ.κ Πεκμεστζή Κιαμάλ και Γεώργιο Γκούμα. Ακόμη, θα ήθελα να ευχαριστήσω τους συμφοιτητές και τους φίλους μου. Χωρίς αυτούς ανφμίβολα πολλά θα ήταν διαφορετικά. Ένα μεγάλο ευχαριστώ στην οικογένειά μου, για την αγάπη τους καθώς και για την αμέριστη συμπαράσταση και κατανόηση που μου έδειξαν καθ' όλη τη διάρκεια των σπουδών. Τέλος, αφιερώνω την παρούσα στη μνήμη των παππούδων μου, που πλέον δεν βρίσκονται εν ζωή.

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος πινάκων

Περίληψη

Τα τελευταία χρόνια, αναδυόμενες εφαρμογές ιστού όπως η ανάλυση μεγάλων δεδομένων, έχουν αυξήσει σε σημαντικό βαθμό το φόρτο εργασίας των κέντρων δεδομένων. Το 2015, η συνολική κίνηση στο δίκτυο των κέντρων δεδομένων ήταν περίπου 4,7 Exabytes και εκτιμάται ότι μέχρι τα τέλη του 2018 θα ξεπεράσει τα 8,5 Exabytes. Οι αυξανόμενες απαιτήσεις όσον αφορά την απόδοση των συστημάτων, οδήγησαν τις εταιρείες στο να χαράξουν νέα μονοπάτια. Ως εκ τούτου, άρχισαν να εμφανίζονται στα κέντρα δεδομένων συστήματα ετερογενών αρχιτεκτονικών που διαθέτουν ενσωματωμένους επεξεργαστές και επιταχυντές επαναπρογραμματιζόμενης λογικής (FPGAs). Με αυτόν τον τρόπο, ο φόρτος εργασίας μπορεί να διαμοιραστεί και μέρος αυτού να αποφορτωθεί σε FPGAs ή σε ενσωματωμένους επεξεργαστές.

Για το σκοπό αυτό, αρχικά θα εγκαταστήσουμε το Apache Spark, ένα πρόγραμμα γενικού σκοπού, ανθεκτικό σε σφάλματα (δικτύου κλπ.), και ευρέως χρησιμοποιούμενο στον χώρο της ανάλυσης μεγάλων δεδομένων, σε τρία ενσωματωμένα συστήματα: το Raspberry Pi 3, το DragonBoard 410c και το PYNQ-Z1. Θα παρουσιάσουμε όλα τα βήματα αυτής της διαδικασίας καθώς και όλες τις απαραίτητες ρυθμίσεις στις οποίες προβήκαμε.

Στη συνέχεια, θα δημιουργήσουμε ένα σύμπλεγμα συστημάτων ετερογενούς αρχιτεκτονικής, το οποίο θα αποτελείται από τέσσερεις PYNQ-Z1 κόμβους και έναν βασισμένο στην Intel αρχιτεκτονική. Θα αναδείξουμε όλα τα απαραίτητα βήματα και τις ρυθμίσεις για την εγκατάσταση του Spark στο συγκεκριμένο σύμπλεγμα, ενώ στη συνέχεια θα παρουσιαστεί ένα προτεινόμενο σχήμα, ώστε οι εφαρμογές του Spark να κάνουν χρήση επιταχυντών υλικού, καθώς και ένα σύνολο βιβλιοθηκών που θα αποκρύπτουν τη χαμηλού επιπέδου επικοινωνία με τον επιταχυντή.

Στο τελευταίο κομμάτι της διπλωματικής εργασίας, αρχικά θα διερευνήσουμε τις δυνατότητες των ενσωματωμένων συστημάτων που χρησιμοποιήσαμε, λαμβάνοντας μετρήσεις για το χρόνο εκτέλεσης ενός συνόλου αλγορίθμων για επεξεργασία γράφων (graph processing) και μηχανικής εκμάθησης (machine learning), ενώ στη συνέ-

χεια θα γίνει περαιτέρω σύγκριση της απόδοσης και της ενεργειακής αποδοτικότητας κάθε συστήματος με έναν «ισχυρό», από άποψη απόδοσης, εξυπηρετητή. Τέλος, το προτεινόμενο σχήμα για την χρήση επιταχυντών υλικού, θα αξιολογηθεί χρησιμοποιώντας μια εφαρμογή που ανήκει στους machine learning αλγόριθμους και πιο συγκεκριμένα σε ένα σενάριο χρήσης του αλγόριθμου λογιστικής παλινδρόμισης (logistic regression).

Η συνολική αξιολόγηση δείχνει ότι γενικά ο χρόνος εκτέλεσης στα ενσωματωμένα συστήμάτα είναι 6,2 έως 13 φορές υψηλότερος σε σύγκριση με έναν τυπικό εξυπηρετητή κέντρου δεδομένων. Ωστόσο, φαίνεται να έχουν περίπου 2 - 3,5 φορές καλύτερη ενεργειακή απόδοση. Επιπλέον, το προτεινόμενο σχήμα για τη χρήση των επιταχυντών υλικού στο Spark, δείχνει ότι το PYNQ-Z1, που βασίζεται στο ετερογενές ZYNQ MPSoC, μπορεί να επιτύχει μέχρι και 2 φορές καλύτερο χρόνο εκτέλεσης σε σύγκριση με ένα σύστημα Xeon και 18 φορές καλύτερη ενεργειακή απόδοση. Ειδικά για ενσωματωμένες εφαρμογές, το προτεινόμενο σχήμα μπορεί να επιτύχει έως και 36 φορές καλύτερο χρόνο εκτέλεσης σε σύγκριση με τα αποτελέσματα από την εκτέλεση σε ενσωματωμένους επεξεργαστές χαμηλής ισχύος (όπως οι επεξεργαστές ARM) και 29 φορές χαμηλότερη κατανάλωση ενέργειας.

Λέξεις κλειδιά: Apache Spark, μηχανική εκμάθηση, επιταχυντές υλικού, ανάλυση μεγάλων δεδομένων, ενσωματωμένα συστήματα, Raspberry Pi 3, DragonBoard 410c, PYNQ-Z1

Abstract

Emerging web applications like big data analytics have significantly increased the workload on the data centers during the last years. In 2015, the total network traffic of the data centers was around 4.7 Exabytes and it is estimated that by the end of 2018 it will cross the 8.5 Exabytes mark. The growing demands both in performance and energy efficiency, have led companies into charting new paths for developing energy-efficient platforms for heterogeneous datacenters, therefore they recently started deploying FPGA accelerators and further offloading part of the workload to embedded processors (i.e. ARM processors) at a datacenter scale.

For this reason we are going to first map Apache Spark, a widely used, fault-tolerant and general-purpose cluster computing framework on several embedded systems including Raspberry Pi 3, DragonBoard 410c and PYNQ-Z1. We present the whole procedure of mapping and deploying Spark on the embedded devices along with any necessary configurations.

Subsequently, we are going to create a heterogeneous cluster consisting of four PYNQ-Z1 nodes and a typical Intel based one. Next on, we will go through all the necessary steps and configurations for deploying Spark on the implemented cluster. Then, a proposed framework for the seamless utilization of hardware accelerators for Spark applications will be presented, as well as a set of libraries to hide the accelerator's low-level details, simplifying in this way the incorporation of hardware accelerators in Spark.

In the last part of the thesis, we are going to first explore the capabilities of the embedded platforms we used, by taking execution metrics using a set of typical machine learning and graph processing algorithms and further comparing the performance and energy efficiency of each system with a mainstream powerful server. Finally, the proposed framework is evaluated in a machine learning application for a use case scenario on logistic regression. The overall evaluation shows that in general the execution time on embedded systems is 6.2x to 13x

higher compared to a typical datacenter server but the embedded platforms are 2x - 3.5x better in terms of energy efficiency. On the other hand, the proposed framework for the utilization of hardware accelerators in Spark shows that PYNQ's heterogeneous accelerator-based ZYNQ MPSoC, can achieve up to 2x system speedup compared to a Xeon system and 18x better energy-efficiency. Especially for embedded applications, the proposed framework can achieve up to 36x speedup compared to the software only implementation on low-power embedded processors (ARM processors) and 29x lower energy consumption.

Keywords: Apache Spark, machine learning, FPGA accelerators, big data analytics, embedded systems, Raspberry Pi 3, DragonBoard 410c, PYNQ-Z1

Δημοσιεύσεις

Μέρη αυτής της εργασίας έχουν δημοσιευθεί στα παρακάτω συνέδρια:

- **Performance and Energy evaluation of Spark applications on low-power SoCs**, Christoforos Kachris, Ioannis Stamelos, Dimitrios Soudris
  International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), 2016

- **Spark acceleration on FPGAs: A use case on machine learning in Pynq**, Elias Koromilas, Ioannis Stamelos, Christoforos Kachris, Dimitrios Soudris
  International Conference on Modern Circuits and Systems Technologies (MOCAST), 2017

- **FPGA acceleration of Spark applications in a Pynq cluster**, Christoforos Kachris, Elias Koromilas, Ioannis Stamelos, Dimitrios Soudris
  International Conference in Field-Programmable Logic and Applications (FPL), 2017

- **SPynq: Acceleration of Machine Learning Applications over Spark on Pynq**, Christoforos Kachris, Elias Koromilas, Ioannis Stamelos, Dimitrios Soudris
  International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), 2017

1 Εισαγωγή

1.1 Μοντέρνα Συστήματα και Εφαρμογές

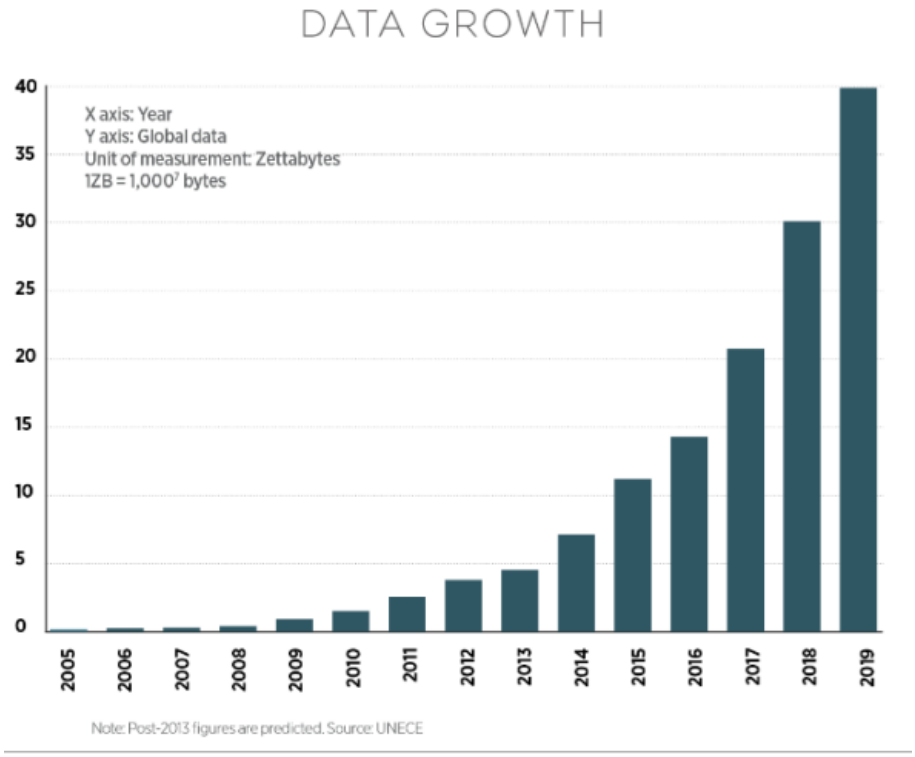
Με την πάροδο του χρόνου, κάθε επιστήμη και ειδικότητα χρησιμοποιεί την τεχνολογία για να προχωρήσει στην επίτευξη νέων στόχων. Είτε πρόκειται για αγρότες, για οικονομικούς αναλυτές ή προγραμματιστές υπολογιστών κτλ., όλοι χρησιμοποιούν συστήματα που τους βοηθούν να διεκπεραιώνουν τη κομμάτι της εργασίας τους σε μικρότερο χρονικό διάστημα και με λιγότερο κόπο.

Έτσι, υπάρχει μία αυξανόμενη ζήτηση στην ενσωμάτωση αισθητήρων και άλλων ηλεκτρονικών στοιχείων σε σχεδόν κάθε σύστημα, με σκοπό τα συστήματα αυτά να μπορούν να επικοινωνούν μεταξύ τους και να μπορούν να συλλέγουν καθώς και να ανταλλάσσουν δεδομένα. Το διαδίκτυο από την άλλη πλευρά, έχει αλλάξει ριζικά τον τρόπο με τον οποίο επικοινωνούμε και εργαζόμαστε. Τα σύγχρονα συστήματα χρησιμοποιούν το διαδίκτυο για να επικοινωνούν μεταξύ τους καθώς και για να μοιράζονται την επεξεργασία διαφόρων εργασιών που τους ανατίθενται. Ακόμη, το διαδίκτυο χρησιμοποιείται και ως μία μεγάλη βάση δεδομένων, που παρέχει πληροφορίες για σχεδόν οτιδήποτε μπορεί κανείς να φανταστεί.

Λαμβάνοντας υπόψιν τα παραπάνω, αναδυόμενες εφαρμογές όπως η ανάλυση μεγάλων δεδομένων και το Internet of Things (IoTs) απαιτούν ισχυρά συστήματα που μπορούν να επεξεργάζονται μεγάλο όγκο δεδομένων, χωρίς να καταναλώνουν πολλή ενέργεια. Αυτές οι εφαρμογές χρειάζονται γρήγορη εμπορική αξιοποίηση και μικρό χρόνο ανάπτυξης. Έτσι, για την αντιμετώπιση των μεγάλων απαιτήσεων επεξεργασίας των αναδυόμενων εφαρμογών, απαιτούνται πρωτότυπες αρχιτεκτονικές στον τομέα των υψηλής απόδοσης και ενεργειακά αποδοτικών επεξεργαστών [1].

1.2 Μεγάλα δεδομένα (Big Data)

Όπως φαίνεται στην προηγούμενη ενότητα, τα σύγχρονα συστήματα και οι εφαρμογές είτε παράγουν είτε χρειάζονται πολλά δεδομένα εισόδου, γεγονός που μας οδήγησε να ορίσουμε ένα νέο όρο, τα μεγάλα δεδομένα (big data). Ως μεγάλα δεδομένα, ορίζονται τα σύνολα δεδομένων τα οποία είναι τόσο μεγάλα ή πολύπλοκα, που οι παραδοσιακές εφαρμογές επεξεργασίας δεδομένων δεν μπορούν να τα χειριστούν και να τα επεξεργαστούν [2].



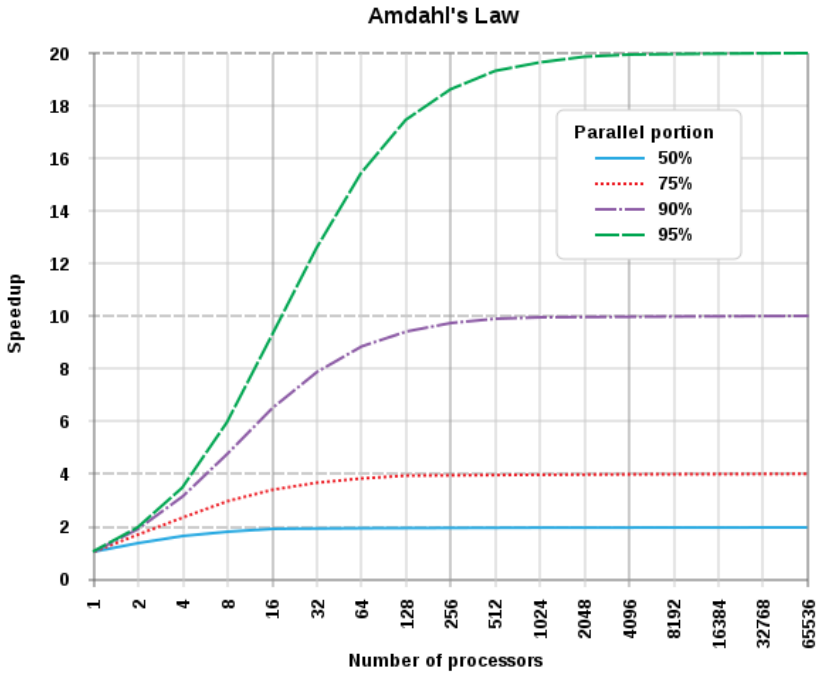
Σχήμα 1.1: Μεγάλα Δεδομένα [3]

Έτσι, υπάρχουν πολλές προκλήσεις όπως η καταγραφή, η αποθήκευση, η ανάλυση, η επεξεργασία, η αναζήτηση, η κοινή χρήση, η μεταφορά, η οπτικοποίηση και η ενημέρωση των δεδομένων, καθώς και προκλήσεις σχετικά με τη διασφάλιση της προστασίας των πληροφοριών που αυτά περιέχουν. Ακόμη, χρησιμοποιούμε τον όρο «μεγάλα δεδομένα» για να αναφερθούμε στα δεδομένα ή τα σύνολα δεδομένων που εξάγονται από μεθόδους πρόβλεψης, συμπεριφοράς καθώς και άλλες προηγμένες μεθόδους ανάλυσης δεδομένων. Τα σύνολα δεδομένων αναπτύσσονται ταχύτατα, καθώς υπάρχει μία πληθώρα συσκευών στον χώρο του Internet of Things (π.χ. κινητές και φορητές συσκευές, αρχεία καταγραφής λογισμικού, φωτογραφικές μηχανές κ.λπ.). Μέσω

της ανάλυσης αυτών των δεδομένων μπορούμε να καταλήξουμε σε πολύ σημαντικά συμπεράσματα, να βρούμε συσχετισμούς στα δεδομένα, να κάνουμε προβλέψεις και να εντοπίσουμε τάσεις, καθώς ακόμη και να αποτρέψουμε την εξάπλωση ασθενειών ή να βρούμε λύσεις για πολλά - μέχρι σήμερα - άλυτα προβλήματα.
Όμως, ο τεράστιος όγκος αυτών των δεδομένων καθιστά δύσκολη τη διαχείριση και την επεξεργασία τους. Οδηγούμαστε έτσι στην λύση του παράλληλου υπολογισμού, χρησιμοποιώντας λογισμικά τα οποία εκτελούνται ακόμη και σε χιλιάδες διακομιστές.

1.3 Παράλληλα και Κατανεμημένα Συστήματα

Το cluster computing, εμφανίστηκε ως μία λύση στο πρόβλημα του χειρισμού και της επεξεργασίας αυτού του τεράστιου όγκου δεδομένων. Ο όρος cluster αναφέρεται σε μια ομάδα συστημάτων που μπορούν να εκτελούν παράλληλα και να κατανέμουν τις εργασίες και λειτουργούν ως ένα ενοποιημένο σύστημα μέσω λογισμικού και δικτύωσης [4].



Σχήμα 1.2: Το μοντέλο του Amdahl για την αποδοτικότητα της κλιμάκωσης [6]

Τα cluster των υπολογιστών μπορούν να κυμαίνονται από δύο έως εκατοντάδες συνδεδεμένα συστήματα, ενώ υπάρχουν δύο βασικί λόγοι για τους οποίους γίνεται η

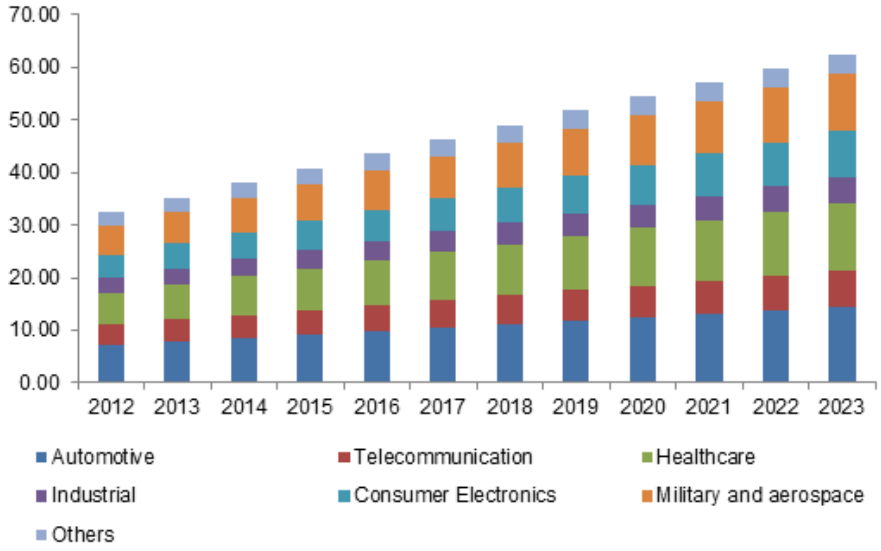
χρήση των clusters. Ο πρώτος αφορά στην υψηλή διαθεσιμότητα (High Availability - HA) για μεγαλύτερη αξιοπιστία και ο δεύτερος στο High Performance Computing (HPC), για περιοχές όπου απαιτείται μεγαλύτερη υπολογιστική ισχύς. Για παράδειγμα, τα μικρά συμπλέγματα υπολογιστών μπορούν να χρησιμοποιηθούν για τη βελτίωση της απόδοσης του ιστού, αφού μπορούν να χειριστούν παράλληλα πολλαπλές εισερχόμενες αιτήσεις. Από την άλλη πλευρά, μεγάλα συμπλέγματα μπορούν να χρησιμοποιηθούν για την εκτέλεση επιστημονικών υπολογισμών ή για έναν μεγάλο αριθμό σύνθετων αλγορίθμων [5].

Καθώς οι συστοιχίες υπολογιστών στον χώρο του HPC μεγαλώνουν (όσον αφορά των αριθμό των κόμβων των υπολογιστών), εμφανίζονται νέες προκλήσεις, μεταξύ των οποίων βρίσκονται η εξεύρεση λύσεων για τη μείωση της αυξανόμενης πολυπλοκότητας, καθώς και η ελαχιστοποίηση της συνολικής κατανάλωσης ενέργειας (συμπεριλαμβανομένων της κατανάλωσης των διακομιστών και της κατανάλωσης για ψύξη των συστημάτων κ.λπ.).

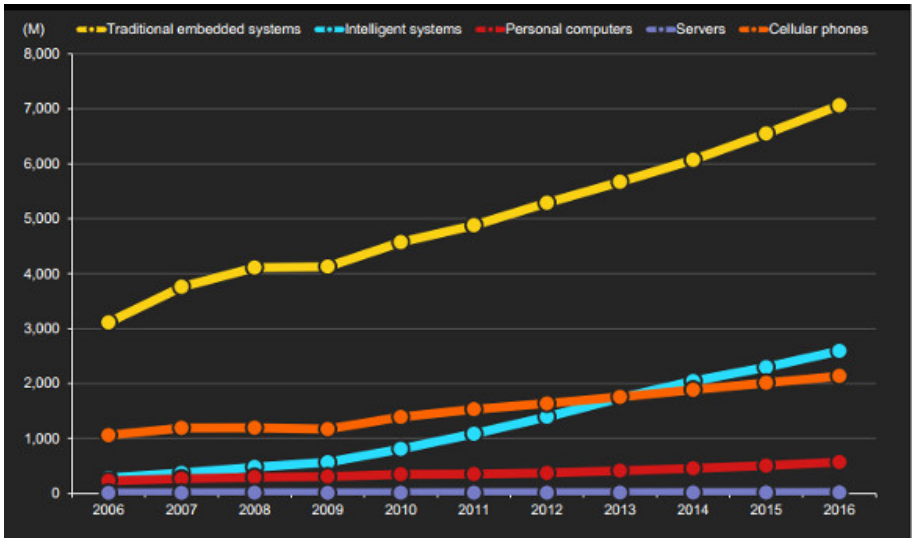
1.4 Ενσωματωμένα Συστήματα

Ο όρος *ενσωματωμένο σύστημα* αναφέρεται σε ένα σύστημα υπολογιστή που έχει σχεδιαστεί για να επιτελεί μια συγκεκριμένη λειτουργία και συνήθως βρίσκεται τοποθετημένο μέσα σε ένα μεγαλύτερο μηχανικό ή ηλεκτρικό σύστημα [7].

Στις μέρες μας, σε έναν κόσμο «έξυπνων» - αποκαλούμενων - συσκευών, τα ενσωματωμένα συστήματα μπορούν να βρεθούν παντού. Οι βιομηχανικές μηχανές, οι γεωργικές και μεταποιητικές συσκευές, τα αυτοκίνητα, ο ιατρικός εξοπλισμός, οι κάμερες, οι οικιακές συσκευές, τα αεροπλάνα, οι μηχανές αυτόματης πώλησης, τα παιχνίδια, καθώς και οι κινητές συσκευές είναι όλες υποψήφιες τοποθεσίες για ένα ενσωματωμένο σύστημα [7]. Χαρακτηρίζονται από το μικρό τους μέγεθος, τη χαμηλή κατανάλωση ενέργειας και το χαμηλό τους κόστος ανά μονάδα. Ωστόσο, χαρακτηρίζονται και από τους περιορισμένους διαθέσιμους πόρους επεξεργασίας, γεγονός που καθιστά τον προγραμματισμό τους και την αλληλεπίδραση μαζί τους μια επίπονη διαδικασία. Οι προγραμματιστές υπολογιστών και οι μηχανικοί σχεδιασμού είναι αναγκασμένοι να προβαίνουν σε όλες τις δυνατές βελτιστοποιήσεις τόσο στο κομμάτι του λογισμικού όσο και σε αυτό του υλικού, προκειμένου να πετυχαίνουν ταυτόχρονα χαμηλή κατανάλωση ενέργειας και υψηλή απόδοση.



Σχήμα 1.3: Μέγεθος της αγοράς ενσωματωμένων συστημάτων στην Ευρώπη, ανά εφαρμογή, 2012-2023 (εκατομμύρια δολάρια) [8]



Σχήμα 1.4: Η AMD επισημαίνει την μελλοντική ανάπτυξη της αγοράς ενσωματωμένων συστημάτων [9]

Τέλος, ένα ακόμη θετικό των ενσωματωμένων συστημάτων είναι ότι προσφέρονται για λύσεις που απαιτούν μεγάλη κλιμάκωση, καθώς το μικρό τους μέγεθος ενδείκνυται στη δημιουργία μεγάλων clusters που αποτελούνται από χιλιάδες κόμβους. Έτσι, τα θετικά στοιχεία του cluster computing μπορούν να συνδυαστούν με τα οφέλη της χαμηλής κατανάλωσης ενέργειας των ενσωματωμένων συστημάτων.

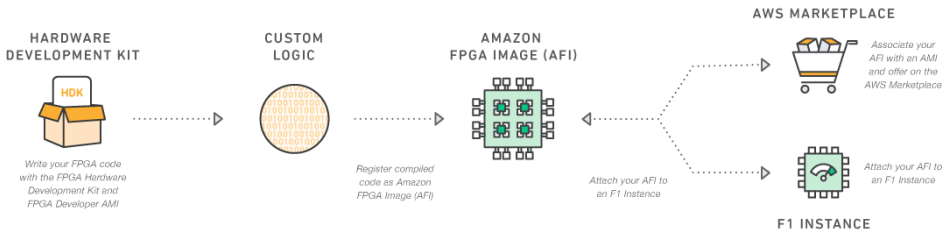
1.5 FPGAs στα Κέντρα Δεδομένων

Με τον όρο Field-Programmable Gate Array (FPGA) αναφερόμαστε σε μία επαναπρογραμματιζόμενη λογική που αποτελείται από ένα ολοκληρωμένο κύκλωμα, σχεδιασμένο να διαμορφώνεται μετά την κατασκευή του από κάποιον πελάτη ή σχεδιαστή [10]. Η διαμόρφωση ενός FPGA καθορίζεται συνήθως χρησιμοποιώντας μια γλώσσα περιγραφής υλικού (HDL). Από το όνομά του, διαφαίνεται ότι ένα FPGA περιέχει μια σειρά προγραμματιζόμενων λογικών μπλοκ και μια ιεραρχία αναδιαμορφώσιμων διασυνδέσεων που επιτρέπουν στα μπλοκ να συνδέονται μεταξύ τους, όπως και πολλές λογικές πύλες που μπορούν να διασυνδεθούν σε διαφορετικές διαμορφώσεις. Τα λογικά μπλοκ μπορούν να διαμορφωθούν έτσι ώστε να εκτελούν σύνθετες συνδυαστικές λειτουργίες ή να σχηματίζουν απλές λογικές πύλες όπως οι AND και οι XOR.

Το μεγαλύτερο πλεονέκτημα των FPGAs είναι η ευελιξία τους. Μέσω της HDL, μπορεί κανείς γρήγορα να υλοποιήσει ή να επαναπρογραμματίσει τη λογική ενός FPGA [11]. Ο HDL κώδικας, μεταφορτώνεται ως κώδικας bit-stream στο FPGA και στη συνέχεια δημιουργείται η λογική του. Δεδομένου ότι ο κώδικας των προγραμματιστών μεταφράζεται σε λογική υλικού, ένα άλλο πλεονέκτημα των FPGAs είναι η πολύ υψηλή τους απόδοση και η διατήρηση της πολύ χαμηλής κατανάλωσης ενέργειας.

Σήμερα, τα FPGAs χρησιμοποιούνται για την επεξεργασία δεδομένων αισθητήρων που βρίσκονται σε αυτοκίνητα, για ενσωματωμένες βιομηχανικές εφαρμογές καθώς και για την επιτάχυνση εφαρμογών δικτύου και άλλων εργασιών όπου απαιτούνται υψηλές επιδόσεις και σημαντική ενεργειακή αποδοτικότητα [12]. Παράλληλα, είναι ιδανικά για τα συστήματα του μέλλοντος, τα οποία απαιτούν μεγαλύτερες υπολογιστικές δυνατότητες, οι οποίες θα τους επιτρέπουν να υποστηρίζουν ένα εκτεταμένο σύνολο φόρτων εργασίας καθώς και τους υποκείμενους εξελισσόμενους αλγόριθμούς τους. Για παράδειγμα, η ανάλυση μεγάλων δεδομένων, η μηχανική εκμάθηση, η επεξεργασία όρασης, η γονιδιωματική κ.α., είναι εφαρμογές που δεν μπορούν να εκτελεστούν από τα υπάρχοντα συστήματα με αποδοτικό και οικονομικό (από άποψη κατανάλωσης) τρόπο [13].

Αυτός είναι και ο λόγος για τον οποίο μεγάλες εταιρείες όπως η Amazon, έχουν ήδη αρχίσει να χρησιμοποιούν FPGAs στα κέντρα δεδομένων και να πωλούν υπηρεσίες που βασίζονται σε αυτά [14].

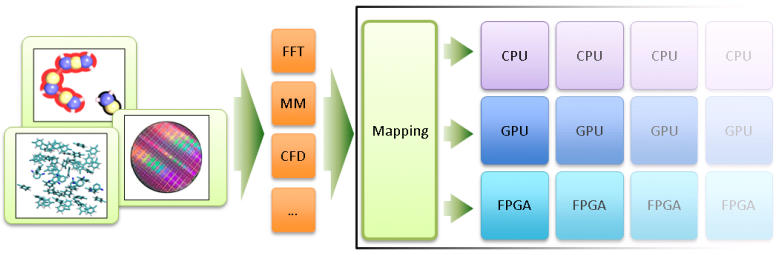


Σχήμα 1.5: Σχήμα λειτουργίας των μηχανών F1 της Amazon [14]

1.6 Σκοπός της Εργασίας

Σκοπός αυτής της εργασίας είναι να διερευνηθούν οι δυνατότητες των ενσωματωμένων συστημάτων και επεξεργαστών, λαμβάνοντας μετρήσεις εκτέλεσης χρησιμοποιώντας ένα σύνολο τυπικών αλγορίθμων μηχανικής εκμάθησης και επεξεργασίας γράφων και η περαιτέρω σύγκριση κάθε συστήματος για την αξιολόγηση της συνολικής απόδοσης και της ενέργειας που καταναλώνει. Έπειτα, θα γίνει μια προσπάθεια προκειμένου να συνδυαστεί η χρήση των CPU και των FPGAs και τέλος θα πραγματοποιηθεί μια αξιολόγηση των πλεονεκτημάτων και των μειονεκτημάτων που ενέχει αυτή η προσέγγιση ετερογενούς συστήματος.

Πιο αναλυτικά, βασικός στόχος είναι να καταλήξουμε σε συμπεράσματα και να λάβουμε feedback σχετικά με το εάν οι ενσωματωμένες συσκευές και οι επεξεργαστές θα μπορούσαν να κερδίσουν μια «θέση κλειδί» στον χώρο του HPC ή ακόμη και σε τομείς όπου η κατανάλωση ενέργειας είναι ο κύριος περιοριστικός παράγοντας. Τα αποτελέσματα αυτής της αξιολόγησης θα παρέχουν πληροφορίες σχετικά με τις επιδόσεις, την κατανάλωση ενέργειας καθώς και τις δυνατότητες κλιμάκωσης.



Σχήμα 1.6: Παράδειγμα πλατφόρμας ετερογενούς αρχιτεκτονικής [15].

Όσον αφορά τον ετερογενή υπολογισμό (heterogeneous computing), είναι αυτονόητο ότι πρόκειται για έναν καινούργιο τομέα ο οποίος ενέχει πολλές προκλήσεις για την υπολογιστική κοινότητα. Η παρουσία πολλαπλών στοιχείων επεξεργασίας εγείρει

όλα τα ζητήματα που σχετίζονται με τα ομοιογενή συστήματα παράλληλης επεξεργασίας, ενώ το επίπεδο της ετερογένειας σε ένα σύστημα, μπορεί να εισάγει προβλήματα στην ομοιόμορφη ανάπτυξη των συστημάτων και στις χρησιμοποιούμενες πρακτικές προγραμματισμού [16]. Για αυτό, θα υπάρξει μια προσπάθεια δημιουργίας ενός API για την απρόσκοπτη χρήση επιταχυντών υλικού, που θα μπορούν να χρησιμοποιηθούν τόσο σε ενσωματωμένα συστήματα όσο και σε εφαρμογές με απαιτήσεις υψηλής απόδοσης όπως το cloud, edge και fog computing. Έπειτα, θα ληφθούν μετρήσεις απόδοσης συγκρίνοντας τον χρόνο εκτέλεσης της περίπτωσης όπου χρησιμοποιούνται μόνο οι CPU πυρήνες με αυτήν όπου γίνεται συνδυαστική χρήση των CPU πυρήνων και των επιταχυντών υλικού.

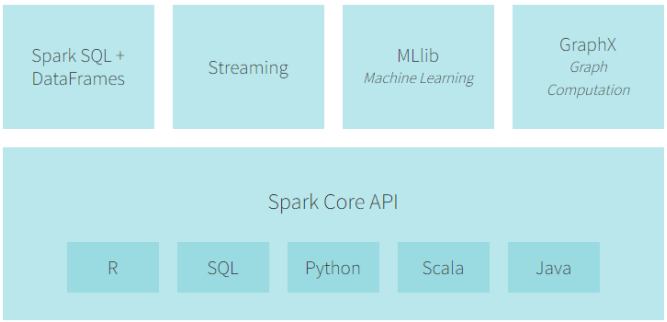
Τέλος, είναι σημαντικό να σημειωθεί ότι το Apache Spark θα χρησιμοποιηθεί τόσο για την αξιολόγηση των επεξεργαστών χαμηλής ισχύος που βασίζονται σε SoC (και χρησιμοποιούνται κυρίως σε ενσωματωμένα συστήματα) όσο και για την προσέγγιση των ετερογενών συστημάτων που αναφέραμε. Το Spark είναι ένα από τα πιο ευρέως χρησιμοποιούμενα frameworks στο χώρο του cloud computing, και περιλαμβάνει μία πληθώρα ενσωματωμένων βιβλιοθηκών και εφαρμογών πάνω στις οποίες θα ληφθούν και οι μετρήσεις. Στο επόμενο κεφάλαιο θα εξετάσουμε αναλυτικότερα τη λειτουργικότητα του Apache Spark καθώς και τα βασικά του χαρακτηριστικά.

2 Apache Spark

2.1 Επισκόπηση

Το Apache Spark αποτελεί μία ισχυρή, 100% ανοιχτού κώδικα μηχανή επεξεργασίας που βασίζεται στην ταχύτητα, την ευκολία χρήσης και τις εξελιγμένες δυνατότητες στην ανάλυση δεδομένων και αναπτύχθηκε στο UC Berkeley το 2009 [17]. Πιο αναλυτικά, το Spark είναι ένα γρήγορο, γενικού σκοπού και ανεκτικό σε σφάλματα cluster computing σύστημα. Παρέχει APIs υψηλού επιπέδου σε Java, Scala, Python και R, καθώς και έναν βελτιστοποιημένο μηχανισμό που υποστηρίζει γενικά την εκτέλεση γράφων. Υποστηρίζει επίσης ένα πλούσιο σύνολο εργαλείων υψηλότερου επιπέδου, όπως το Spark SQL για SQL και επεξεργασία δομημένων δεδομένων, την βιβλιοθήκη MLlib που προσφέρεται για μηχανική εκμάθηση, το πακέτο GraphX για επεξεργασία γράφων και το Spark Streaming [18]. Επίσης χρησιμοποιείται συχνά με άλλα εργαλεία για την ανάλυση μεγάλων δεδομένων. Συγκεκριμένα, το Spark μπορεί να τρέξει σε Hadoop clusters ([19]) και να έχει πρόσβαση σε οποιαδήποτε Hadoop πηγή δεδομένων, συμπεριλαμβανομένου του πακέτου Cassandra [20].

(α′) Apache Spark logo



(β′) Το οικοσύστημα του Apache Spark [17]

Σχήμα 2.1: Το Apache Spark

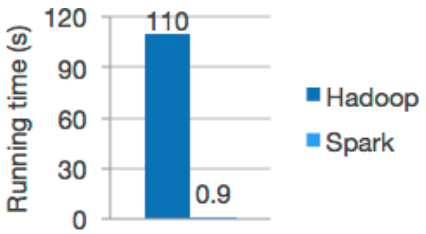
Από την κυκλοφορία του, το Apache Spark έχει γίνει αποδεκτό και χρησιμοποιείται από επιχειρήσεις σε ένα ευρύ φάσμα βιομηχανιών. Η Netflix, η Yahoo και το eBay χρησιμοποιούν το Spark σε πολύ μεγάλη κλίμακα, επεξεργάζοντας συλλογικά πολλαπλά petabytes δεδομένων, σε clusters άνω των 8.000 κόμβων. Τέλος, θα ήταν χρήσιμο να σημειώσουμε ότι αποτελεί το μεγαλύτερο έργο ανοιχτού κώδικα στην επεξεργασία δεδομένων [17].

2.2 Πλεονεκτήματα

Όσον αφορά τα πλεονεκτήματα του Apache Spark, μπορούν να χωριστούν σε τρεις βασικές κατηγορίες: την *ταχύτητα*, την *ευκολία χρήσης* και το γεγονός ότι όλη του η λειτουργικότητα βασίζεται σε ένα *ενοποιημένο πλαίσιο*.

2.2.1 Ταχύτητα

Η ταχύτητα είναι ένας πολύ σημαντικός παράγοντας στο πεδίο της ανάλυσης μεγάλων δεδομένων, όπου η επεξεργασία μεγάλων συνόλων δεδομένων σημαίνει τη διαφορά μεταξύ της διαδραστικής διερεύνησης δεδομένων και της αναμονής λεπτών ή ωρών.



Σχήμα 2.2: Εκτέλεση του αλγορίθμου λογιστικής παλινδρόμησης στο Hadoop και στο Spark

Το Spark είναι κατασκευασμένο με βάση το bottom-up σχήμα για απόδοση και χαρακτηρίζεται ως ένα από τα ταχύτερα cluster computing συστήματα για την ανάλυση μεγάλων δεδομένων. Είναι έως και 100 φορές ταχύτερο από το Hadoop, επεκτείνοντας το δημοφιλές MapReduce μοντέλο για την αποτελεσματική υποστήριξη περισσότερων τύπων υπολογισμών, συμπεριλαμβανομένων των διαδραστικών ερωτημάτων και της επεξεργασίας ροής δεδομένων [21]. Ο κύριος λόγος για τον οποίο είναι τόσο γρήγορο, εντοπίζεται στην ικανότητά του να εκτελεί υπολογισμούς στη μνήμη, ενώ αξίζει να σημειωθεί ότι το Apache Spark κατέχει μέχρι σήμερα το παγκόσμιο ρεκόρ ταξινόμησης δεδομένων μεγάλης κλίμακας σε δίσκο.

2.2.2 Ένα Ενοποιημένο Πλαίσιο

Το Spark έχει σχεδιαστεί για να καλύπτει ένα ευρύ φάσμα φόρτων εργασίας, που προηγουμένως απαιτούσαν χωριστά κατανεμημένα συστήματα,. Υποστηρίζοντας μία πληθώρα φόρτων εργασίας, το Spark καθιστά την επεξεργασία διαφορετικών τύπων δεδομένων που δημιουργούν σύνθετους φόρτους εργασίας μία εύκολη και φθηνή διαδικασία, κάτι που είναι συχνά απαραίτητο στον χώρο της ανάλυσης μεγάλων δεδομένων. Με αυτόν τον τρόπο, δεν απαιτείται η διαχείριση, ανάπτυξη, συντήρηση, δοκιμή και υποστήριξη ξεχωριστών εργαλείων, ενώ παράλληλα μπορεί να αυξηθεί και η παραγωγικότητα των προγραμματιστών.

2.2.3 Ευκολία Χρήσης

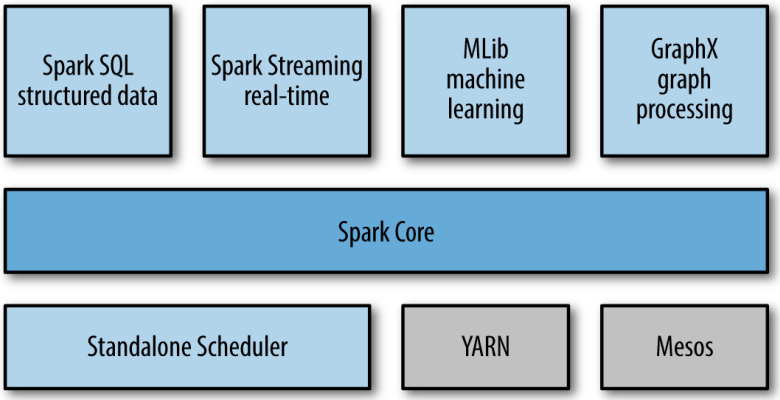
Το Spark διαθέτει εύχρηστα APIs για την διαχείριση μεγάλων συνόλων δεδομένων. Ακόμη, η φιλοσοφία του ενοποιημένου πλαισίου που περιεγράφηκε στην προηγούμενη υπό-ενότητα έχει αρκετά οφέλη. Αρχικά, όλες οι βιβλιοθήκες και τα στοιχεία υψηλότερου επιπέδου, επωφελούνται από βελτιώσεις στα χαμηλότερα επίπεδα. Για παράδειγμα, αν προστεθεί κάποια βελτιστοποίηση στον πυρήνα (core engine) του Spark, αντίστοιχη βελτίωση ή επιτάχυνση παρατηρείται αυτομάτως και στις βιβλιοθήκες του Spark (πχ. στην SQL στην MLlib κλπ.). Έπειτα, κάθε φορά που ένα νέο στοιχείο προστίθεται στη στοίβα του Spark, κάθε χρήστης του Spark μπορεί αμέσως να το δοκιμάσει. Με αυτόν τον τρόπο καθίσταται εύκολη, σε χρήστες και εταιρείες, η δοκιμή ενός νέου τύπου ανάλυσης δεδομένων, σε ένα framework που ήδη χρησιμοποιούν και έχουν στήσει, εξοικονομώντας έτσι πολύτιμο χρόνο.

2.3 Συστατικά Στοιχεία

Σε αυτή την ενότητα, θα προβούμε σε μία λεπτομερή ανάλυση των βασικών συστατικών στοιχείων του Apache Spark, τα οποία αποτελούνται από τα στοιχεία που παρουσιάζονται στο σχήμα 2.1 , προσθέτοντας σε αυτά τη στοίβα του διαχειριστή συμπλέγματος (cluster manager), όπως φαίνεται στο ακόλουθο σχήμα (2.3).

2.3.1 Spark Core

Το Spark Core, ο πυρήνας του Spark, είναι η υποκείμενη μηχανή που στεγάζει όλη τη λειτουργικότητα του Spark, συμπεριλαμβανομένων των στοιχείων για τον χρονο-



Σχήμα 2.3: Η Στοίβα του Spark [21].

προγραμματισμό εργασιών, τη διαχείριση της μνήμης, την ανάνηψη από σφάλματα, την αλληλεπίδραση με τα συστήματα αποθήκευσης κ.α. Το Spark Core φιλοξενεί επίσης το API που ορίζει το Resilient Distributed Dataset (RDD), το οποίο αποτελεί την κύρια οντότητα δεδομένων του Spark και θα παρουσιαστεί λεπτομερέστερα ακολούθως.

Resilient Distributed Dataset - RDD

Όπως αναφέρθηκε προηγουμένως, το Resilient Distributed Dataset (γνωστό και ως RDD) είναι η πρωταρχική οντότητα δεδομένων του Apache Spark. Ένα RDD είναι μια συλλογή στοιχείων που έχουν ανοχή σε σφάλματα και τα οποία μπορούν να επεξεργάζονται παράλληλα. Η έννοια του RDD καθώς και το όνομά τους εμφανίστηκαν για πρώτη φορά στο δημπσίευση με τίτλο *Resilient Distributed Datasets: A Tolerant Fault Abstraction for Computing Cluster In Memory* [22].
Τα χαρακτηριστικά των RDDs (αποσυνθέτοντας το όνομά):

- **Resilient"** δηλ. ανεκτικά σε σφάλματα, με τη βοήθεια ενός γράφου «ζωής» κάθε RDD. Είναι καθ' αυτόν τον τρόπο ικανά να υπολογίσουν εκ νέου κομμάτια των δεδομένων που λείπουν ή έχουν υποστεί ζημιά εξαιτίας αποτυχιών ενός κόμβου.
- **Distributed:** τα δεδομένα βρίσκονται διαμοιρασμένα σε πολλούς κόμβους ενός cluster.
- **Dataset:** μια συλλογή δεδομένων που είναι χωρισμένη σε κομμάτια (partitioned collection).

Τα RDDs μπορούν να δημιουργηθούν με δύο τρόπους: είτε με παραλληλισμό μιας υπάρχουσας συλλογής δεδομένων στο πρόγραμμα οδήγησης (driver program) ή με αναφορά ενός συνόλου δεδομένων από οποιαδήποτε πηγή αποθήκευσης υποστηριζόμενη από τον Hadoop (συμπεριλαμβανομένων του τοπικού συστήματος αρχείων, το HDFS, κ.λπ.). Το Spark, υποστηρίζει αρχεία κειμένου, SequenceFiles και οποιαδήποτε άλλο τύπο αρχείου εισόδου του Hadoop [18]. Μια σημαντική παράμετρος για τις παράλληλες συλλογές, είναι ο αριθμός των κατατμήσεων για την αποκοπή του συνόλου δεδομένων. Το Spark εκτελεί μία εργασία για κάθε κομμάτι της συλλογής δεδομένων.

Επιπλέον, τα RDDs επιδέχονται δύο τύπους ενεργειών: τους *μετασχηματισμούς* (transformations), οι οποίοι δημιουργούν ένα νέο σύνολο δεδομένων από ένα υπάρχον και τις *δράσεις* (actions), οι οποίες επιστρέφουν μια τιμή στο πρόγραμμα οδήγησης μετά την εκτέλεση ενός υπολογισμού στο σύνολο των δεδομένων. Για παράδειγμα, η συνάρτηση map() είναι ένας μετασχηματισμός που μεταβιβάζει κάθε στοιχείο ενός συνόλου δεδομένων μέσω μιας συνάρτησης και επιστρέφει ένα νέο RDD που περιέχει τα αποτελέσματα. Από την άλλη πλευρά, η reduce() αποτελεί μια ενέργεια που συγκεντρώνει όλα τα στοιχεία ενός RDD, χρησιμοποιώντας κάποια συνάρτηση, και επιστρέφει το τελικό αποτέλεσμα στο πρόγραμμα οδήγησης. Προκειμένου το Spark να είναι πιο αποδοτικό, όλοι οι μετασχηματισμοί σε ένα RDD είναι lazy, με την έννοια ότι δεν υπολογίζουν τα αποτελέσματά τους μέχρι μία «δράση» να απαιτήσει την επιστροφή ενός αποτελέσματος στο πρόγραμμα οδήγησης.

Μία από τις πιο σημαντικές δυνατότητες του Spark είναι ότι τα δεδομένα των RDD μπορούν να αποθηκευτούν προσωρινά στη μνήμη ή το δίσκο. Με αυτόν τον τρόπο, τα στοιχεία ενός RDD που είχε προηγουμένως υποστεί έναν μετασχηματισμό μπορούν να προσπελαστούν πολύ ταχύτερα, δεδομένου ότι οι νέες ενέργειες πάνω σε αυτό δεν απαιτούν εκ νέου υπολογισμούς. Η προσωρινή αποθήκευση είναι ένα σημαντικότατο εργαλείο όσον αφορά τους επαναληπτικούς αλγόριθμους, όπου η επαναχρησιμοποίηση των δεδομένων είναι μεγάλη και συχνή.

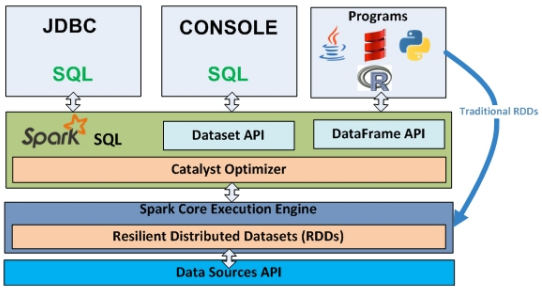
2.3.2 Ενσωματωμένες Βιβλιοθήκες

Μηχανική Εκμάθηση (MLlib)

Το Spark περιλαμβάνει μια βιβλιοθήκη η οποία ονομάζεται MLlib και παρέχει αλγορίθμους μηχανικής εκμάθησης «υψηλής ποιότητας» (π.χ. μέσω πολλαπλών επαναλήψεων αυξάνεται η ακρίβεια του εκπαιδευόμενου μοντέλου), συμπεριλαμβανομένων της ταξινόμησης, της ομαδοποίησης, της παλινδρόμησης και του συνδυαστικού φιλτραρίσματος. Ακόμη, η βιβλιοθήκη μπορεί να χρησιμοποιηθεί ως μέρος εφαρμογών του Spark, οι οποίες είναι γραμμένες σε Java, Python ή Scala.

Spark SQL

Το Spark SQL αποτελεί ένα κομμάτι του Spark για την επεξεργασία δομημένων δεδομένων. Παρέχει ένα αφαιρετικό επίπεδο προγραμματισμού, τα DataFrames, και μπορεί επίσης να λειτουργήσει ως ένα κατανεμημένο σύστημα υποβολής ερωτημάτων SQL. Επιπλέον, το Spark SQL προσφέρει τη δυνατότητα συνδυασμού των προγραμματιστικών δυνατοτήτων του RDD με τη δυνατότητα υποβολής ερωτημάτων SQL.



Σχήμα 2.4: Η αρχιτεκτονική του Spark SQL [24].

Spark Streaming

Το Spark Streaming αποτελεί μία βιβλιοθήκη του Spark που επιτρέπει την επεξεργασία ροών δεδομένων. Είναι ένα πολύ σημαντικό πακέτο, δεδομένου ότι πολλές εφαρμογές χρειάζονται την ικανότητα επεξεργασίας και ανάλυσης ροών δεδομένων σε πραγματικό χρόνο. Οι ροές δεδομένων θα μπορούσαν να περιλαμβάνουν αρχεία καταγραφής ενός διακομιστή, ουρές μηνυμάτων με ενημερώσεις κατάστασης, που δημοσιεύονται από χρήστες μιας υπηρεσίας ιστού κ.α. Πιο συγκεκριμένα, το Spark Streaming παρέχει ένα API για το χειρισμό ροών δεδομένων που ταιριάζει απόλυτα

με το RDD API του Spark Core και συνδυάζεται εύκολα με μια μεγάλη ποικιλία δημοφιλών πηγών δεδομένων, συμπεριλαμβανομένων των HDFS, Flume, Kafka και Twitter.

Σχήμα 2.5: Η αρχιτεκτονική του Spark Streaming

Επεξεργασία Γράφων (GraphX)

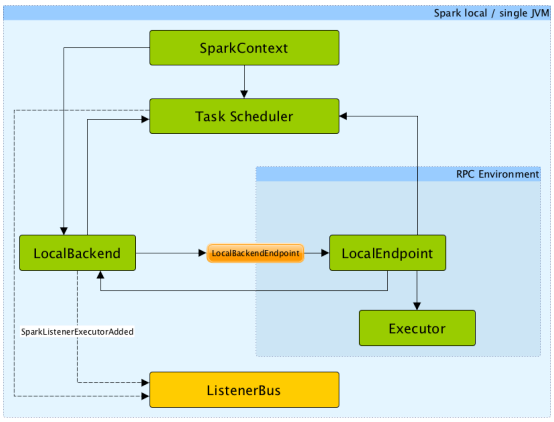
Το GraphX αποτελεί μία βιβλιοθήκη του Spark για την επεξεργασία και τον χειρισμό γράφων (π.χ. γράφων ενός κοινωνικού δικτύου κ.α.) καθώς και την εκτέλεση παράλληλων υπολογισμών σε αυτούς. Όπως το Spark Streaming και το Spark SQL, το GraphX επεκτείνει το Spark RDD API, παρέχοντας έτσι τη δυνατότητα δημιουργίας ενός κατευθυνόμενου γράφου με αυθαίρετες ιδιότητες προσαρτημένες σε κάθε κορυφή και ακμή αυτού.

2.3.3 Διαχειριστές Συμπλέγματος (Cluster Managers)

Μια εφαρμογή του Spark μπορεί να εκτελεστεί είτε τοπικά ή σε κατανεμημένη λειτουργία σε ένα cluster υπολογιστών.

Στην περίπτωση εκτέλεσης μίας εφαρμογής σε ***local mode***, δημιουργείται ένα και μοναδικό JVM. Σε αυτό το JVM, το Spark δημιουργεί όλα τα απαραίτητα στοιχεία, συμπεριλαμβανομένων του driver program, του executor, του master και του LocalSchedulerBackend (όπως φαίνονται στο σχήμα 2.6). Το local mode είναι πολύ βολικό για δοκιμές και εντοπισμό σφαλμάτων σε εφαρμογές, καθώς δεν απαιτεί προηγούμενη διαμόρφωση του Spark. Είναι επίσης πολύ βολικό για δοκιμές σε ενσωματωμένα συστήματα, όπου οι διαθέσιμοι πόροι, κυρίως όσον αφορά τη μνήμη RAM, είναι αρκετά περιορισμένοι. Το πρόβλημα αυτό θα αναδειχθεί κυριώς στο κεφάλαιο 3.

Ωστόσο, το Spark έχει σχεδιαστεί ώστε να προσφέρει αποτελεσματική κλιμάκωση για clusters λίγων έως και πολλών χιλιάδων υπολογιστικών κόμβων. Με αυτόν τον



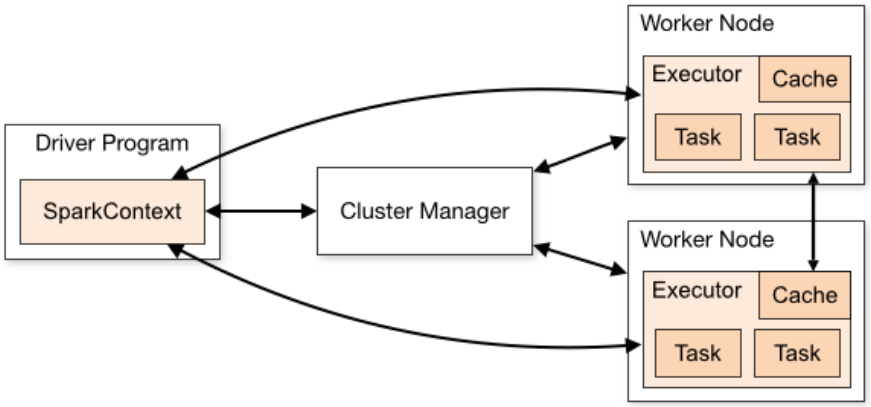
Σχήμα 2.6: Η αρχιτεκτονική του Spark Local Mode [23].

τρόπο, μία εφαρμογή του Spark μπορεί να εκτελεστεί σε ***cluster mode***, χρησιμοποιώντας έναν από τους τρεις επόμενους διαχειριστές συμπλεγμάτων: τον Hadoop YARN, τον Apache Mesos και τον Standalone Scheduler. Όλοι διαθέτουν επιλογές και ρυθμίσεις για τον έλεγχο της χρήσης των πόρων και άλλων δυνατοτήτων, και έχουν ενσωματωμένα εργαλεία επίβλεψης (monitoring tools). Ο διαχειριστής συμπλέγματος είναι υπεύθυνος για τον προγραμματισμό και την κατανομή πόρων σε όλους τους κόμβους ενός cluster.

Το Spark χρησιμοποιεί την αρχιτεκτονική master/slave ενώ το επονομαζόμενο πρόγραμμα οδήγησης (γνωστό και ως SparkContext) είναι ο κεντρικός συντονιστής και η κύρια διεργασία μιας υποβληθείσας εφαρμογής. Το Spark driver ζητά executors από έναν διαχειριστή συμπλέγματος. Κάθε executor συνοδεύεται από έναν αριθμό επεξεργαστών και μια ποσότητα μνήμης. Με άλλα λόγια, οι πόροι που ζητούνται από τον cluster manager μεταφράζονται σε πυρήνες CPU και σε μέγεθος μνήμης. Είναι ευθύνη του cluster manager να εκκινήσει τους Spark executors στο cluster. Η διεργασία του προγράμματος οδήγησης είναι υπεύθυνη για τη μετατροπή μιας εφαρμογής σε μικρότερες μονάδες εκτέλεσης που ονομάζονται εργασίες (tasks) και που εκτελούνται στη συνέχεια από τους executors [25].

Ένα άλλο αξιοσημείωτο χαρακτηριστικό του cluster mode του Spark είναι ότι μια εφαρμογή μπορεί να υποβληθεί σε δύο modes: στο ***client*** και στο ***cluster***. Στο ***cluster deploy mode*** το πρόγραμμα οδήγησης εκκινείται μέσα στο cluster σε έναν από τους workers αυτού και η διεργασία πελάτη (client process) ολοκληρώνεται με την υποβολή της εφαρμογής, χωρίς να αναμένει τον τερματισμό αυτής. Στο ***client***

deploy mode, η εκκίνηση του προγράμματος οδήγησης λαμβάνει χώρα εκτός του cluster, ως ανεξάρτητη διεργασία η οποία είναι συνήθως η ίδια με τη διεργασία πελάτη που χρησιμοποιήθηκε για την εκκίνηση της εργασίας.
Μια επισκόπηση του cluster mode του Spark απεικονίζεται στο ακόλουθο σχήμα. Αντίστοιχα, ο πίνακας 2.1 συνοψίζει τους χρησιμοποιούμενους όρους για την αναφορά στις έννοιες ενός cluster.



Σχήμα 2.7: Επισκόπηση του Cluster Mode [18].

Standalone Scheduler

Για την υποβολή όλων των εφαρμογών στο Spark, χρησιμοποιήθηκε ο Standalone Scheduler.
Ο Standalone Scheduler είναι ένας απλός διαχειριστής συμπλέγματος που περιλαμβάνεται στη διανομή του Spark. Δεδομένου ότι είναι ενσωματωμένος στο περιβάλλον του Spark, προσφέρει τον ευκολότερο τρόπο εκτέλεσης εφαρμογών σε αυτό. Υποστηρίζει υψηλή διαθεσιμότητα για τον master κόμβο, ανοχή σε σφάλματα των worker κόμβων και έχει τη δυνατότητα να διαχειρίζεται τους πόρους που παρέχονται ανά εφαρμογή. Ένα από τα πιο σημαντικά χαρακτηριστικά του είναι ότι μπορεί να τρέξει παράλληλα με ένα υπάρχον Hadoop cluster και να έχει πρόσβαση σε δεδομένα από ένα κατανεμημένο σύστημα αρχείων Hadoop (γνωστό και ως HDFS).

2.3.4 Εργαλεία Επίβλεψης

Υπάρχουν διάφοροι τρόποι επίβλεψης των εφαρμογών του Spark: web UIs, μετρήσεις και εξωτερικά όργανα.

Term	Meaning
Application	User program built on Spark. Consists of a driver program and executors on the cluster.
Application jar	A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.
Driver program	The process running the main() function of the application and creating the SparkContext
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Worker node	Any node that can run application code in the cluster
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Task	A unit of work that will be sent to one executor
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect); you'll see this term used in the driver's logs.
Stage	Each job gets divided into smaller sets of tasks called stages that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

Πίνακας 2.1: Ορολογία σχετικά με τις έννοιες ενός Spark cluster [18].

Κάθε εφαρμογή του Spark διαθέτει ένα web UI για λόγους επίβλεψης, το οποίο εκκινείται από το SparkContext της εφαρμογής. Το περιβάλλον αυτό, εμφανίζει πληροφορίες σχετικά με τα στάδια (stages) και τις εργασίες (tasks) του scheduler, μια περίληψη των μεγεθών των RDDs και τη χρήση μνήμης, καθώς και πληροφορίες σχετικά με τους τρέχοντες executors και τον αποθηκευτικό χώρο που χρησιμοποιείται. Ένα web UI παρέχεται επίσης και μέσω του Standalone cluster manager, το οποίο περιέχει πληροφορίες για τα στατιστικά στοιχεία ενός cluster, καθώς και λεπτομερή καταγραφή για κάθε υποβληθείσα εργασία. Οι προηγούμενες υποβληθείσες εφαρμογές μπορούν επίσης να αναδομηθούν σε μία γραφική προβολή χρησιμοποιώντας τον Spark History Server. Παραδείγματα του UI του Spark και του History Server

πρόκειται να παρουσιαστούν αργότερα, κατά την αξιολόγηση του προτεινόμενού μας framework.

2.4 Ρυθμίσεις

Το Spark παρέχει ένα συμπαγές σχήμα διαμορφώσεων. Περιλαμβάνει ρυθμίσεις που ελέγχουν τις περισσότερες παραμέτρους των εφαρμογών, μεταβλητές περιβάλλοντος που μπορούν να χρησιμοποιηθούν για τις ρυθμίσεις κάθε κόμβου (πχ. Διεύθυνση IP, χρησιμοποιούμενη έκδοση της Python για το PySpark κ.λπ.) και ρυθμίσεις που αφορούν αρχεία καταγραφής.

2.4.1 Ιδιότητες του Spark

Οι ιδιότητες του Spark ελέγχουν τις περισσότερες ρυθμίσεις μίας εφαρμογής και ορίζονται ξεχωριστά ανά εφαρμογή. Αυτές οι ιδιότητες μπορούν να οριστούν απευθείας σε ένα αντικείμενο SparkConf, το οποίο στη συνέχεια μεταβιβάζεται σε ένα αντικείμενο SparkContext ή προσθέτοντας τις αντίστοιχες επιλογές στο αρχείο ρυθμίσεων ***conf/spark-defaults.conf***.

2.4.2 Μεταβλητές Περιβάλλοντος

Ορισμένες ρυθμίσεις του Spark μπορούν να οριστούν μέσω μεταβλητών περιβάλλοντος, οι οποίες με τη σειρά τους ορίζονται στο bash script ***conf/spark-env.sh*** που βρίσκεται στον κατάλογο όπου έχει εγκατασταθεί το Spark. Το ***conf/spark-env.sh*** bash script γίνεται source κατά την εκτέλεση εφαρμογών στο Spark.
Είναι το πιο σημαντικό αρχείο διαμόρφωσης (για αυτήν την εργασία), μαζί με το *conf/spark-defaults.conf*, καθώς θα χρησιμοποιηθούν για τη διαμόρφωση του Spark στα ενσωματωμένα συστήματα.

2.4.3 Αρχεία Καταγραφής

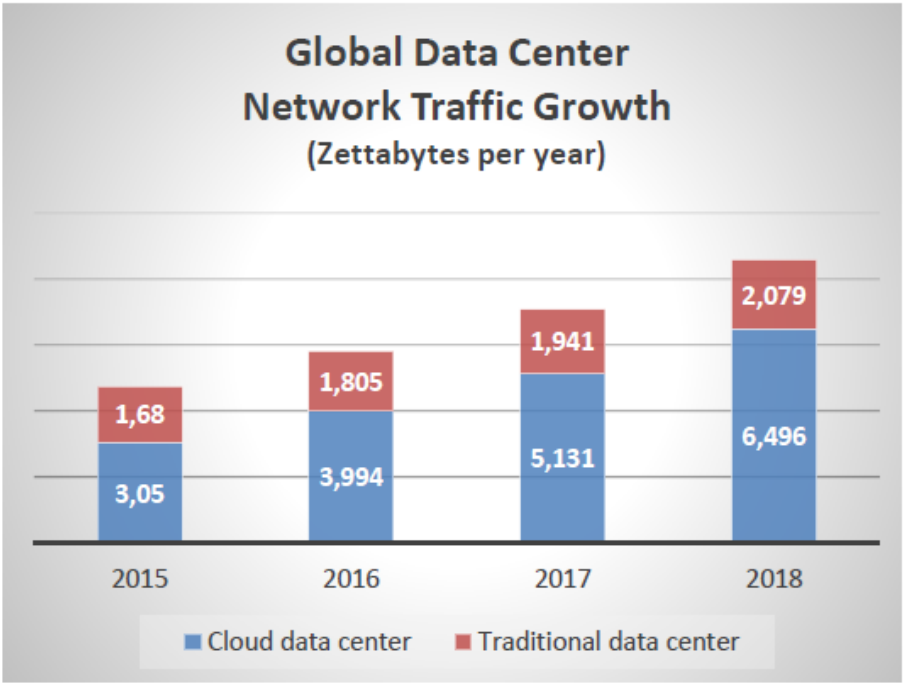
Το Spark χρησιμοποιεί το log4j για την δημιουργία αρχείων καταγραφής. Οι ρυθμίσεις για το log4j ορίζονται στο αρχείο log4j.properties του φακέλου conf του Spark.

3 Το Spark σε Ενσωματωμένα Συστήματα

3.1 Σκοπός

Αναδυόμενες εφαρμογές ιστού, όπως η ανάλυση μεγάλων δεδομένων, έχουν αυξήσει σημαντικά το φόρτο εργασίας στα κέντρα δεδομένων τα τελευταία χρόνια. Το 2015 η συνολική κίνηση του δικτύου των κέντρων δεδομένων ήταν περίπου 4,7 Exabytes, ενώ εκτιμάται ότι μέχρι τα τέλη του 2018 θα έχει ξεπεράσει το όριο των 8,5 Exabytes, ακολουθούμενο από έναν συνολικό ετήσιο ρυθμό ανάπτυξης της τάξης των 33% (σχήμα 3.1, [26]). Προκειμένου να ανταπεξέλθουν σε αυτήν την κλιμάκωση της κίνησης του δικτύου, οι χειριστές των κέντρων δεδομένων έχουν καταφύγει στη χρήση ισχυρότερων διακομιστών.

Από την άλλη πλευρά, βασιζόμενοι στον νόμο του Moore, οι τεχνολογίες που αφορούν τους επεξεργαστές έχουν υποστεί μεγάλη κλιμάκωση τα τελευταία χρόνια, μέσω της τοποθέτησης ενός αυξανόμενου αριθμού τρανζίστορ μέσα στα τσιπ, για την επίτευξη υψηλότερων επιδόσεων. Ωστόσο, οι συχνότητες του ρολογιού των τσιπ δεν μπόρεσαν να ακολουθήσουν αυτή την ανοδική τάση. Έτσι, πριν από μερικά χρόνια υιοθετήθηκε μια εναλλακτική λύση για το πρόβλημα, στρεφόμενοι σε επεξεργαστές πολλαπλών πυρήνων. Με τους επεξεργαστές πολλαπλών πυρήνων, η απόδοση των διακομιστών θα μπορούσε να αυξηθεί χωρίς να χρειάζεται αντίστοιχη αύξηση στη συχνότητα του ρολογιού. Δυστυχώς, και αυτή η λύση βρέθηκε να έχει προβλήματα κλιμάκωσης, καθώς τα κέρδη που επιτυγχάνονται σε όρους απόδοσης με την προσθήκη περισσότερων πυρήνων μέσα σε έναν επεξεργαστή, έρχονται με το κόστος διάφορων περιπλοκών, όπως η επικοινωνία μεταξύ των πυρήνων, η συνοχή της μνή-



Σχήμα 3.1: Η αύξηση στην κίνηση του δικτύου των κεντρων δεδομένων, Πηγή: Cisco Global Cloud Index 2016.

μης και κυρίως η κατανάλωση ενέργειας [27].

Ένας τρόπος αντιμετώπισης αυτού του προβλήματος είναι μέσω της χρήσης μικρο-εξυπηρετητών (microservers). Οι μικρο-εξυπηρετητές κέρδισαν πρόσφατα την προσοχή της κοινότητας, ως χαμηλού κόστους και ισχύος διακομιστές που βασίζονται κυρίως σε επεξεργαστές χαμηλής κατανάλωσης ενέργειας, όπως αυτοί που χρησιμοποιούνται σε ενσωματωμένα συστήματα. Οι μικρο-εξυπηρετητές στοχεύουν κυρίως σε απλές (ελαφρές) ή σε παράλληλες εφαρμογές που επωφελούνται περισσότερο από μεμονωμένους διακομιστές, που προσφέρουν επαρκές I/O μεταξύ των κόμβων.

Όλα αυτά, μας οδήγησαν στο να διενεργήσουμε μία διεξοδική αξιολόγηση της απόδοσης, όσον αφορά τον χρόνο εκτέλεσης και την κατανάλωση ενέργειας, διαφόρων ενσωματωμένων συστημάτων. Η σύγκριση των αποτελεσμάτων με αυτά ενός ισχυρού τυπικού εξυπηρετητή, θα μπορούσε να μας δώσει απαντήσεις σχετικά με το εάν τα ενσωματωμένα συστήματα θα μπορούσαν να διαδραματίσουν κάποιο ρόλο στα κέντρα δεδομένων και αν υπάρχουν οφέλη από την εκτέλεση εφαρμογών μεγάλων δεδομένων σε αυτά. Για να διενεργήσουμε αυτήν την αξιολόγηση, έπρεπε αρχικά να προμηθευτούμε μία σειρά από ενσωματωμένα συστήματα και στη συνέχεια να βρούμε κάποιες αντιπροσωπευτικές εφαρμογές για τη λήψη των μετρήσεων. Το Apache Spark ήταν

ιδανικό για το σκοπό μας. Είναι 100% ανοιχτού κώδικα, είναι ένα framework που προσφέρεται για την ανάλυση μεγάλων δεδομένων και παρέχει built-in αλγόριθμους και εφαρμογές.

Στη συνέχεια, θα εξετάσουμε διεξοδικά τις ενσωματωμένες πλατφόρμες και τις συγκεκριμένες εφαρμογές που χρησιμοποιήσαμε για την αξιολόγηση. Έπειτα, θα αναδείξουμε τις προκλήσεις που αντιμετωπίσαμε προσπαθώντας να στήσουμε το Spark σε αυτές τις πλατφόρμες, περιγράφοντας λεπτομερώς όλα τα απαραίτητα βήματα που ακολουθήσουμε για το mapping του Spark στα ενσωματωμένα συστήματα, συμπεριλαμβανομένων των απαιτούμενων αλλαγών στη διαμόρφωση του Spark.

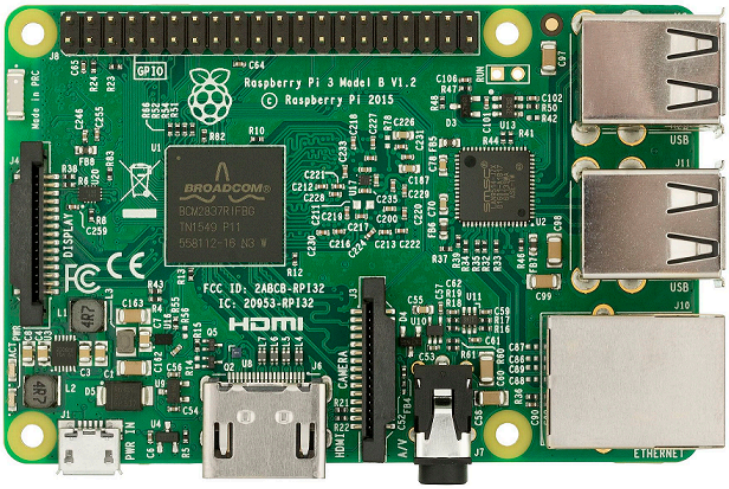
3.2 Ενσωματωμένα Συστήματα

Στις μέρες μας, υπάρχει μια πληθώρα ενσωματωμένων συστημάτων χαμηλής ισχύος που διατίθενται προς πώληση. Για τους σκοπούς αυτής της αξιολόγησης επιλέξαμε να προμηθεύουμε πλατφόρμες, οι επεξεργαστές χαμηλής ισχύος των οποίων βασίζονται στην ARM αρχιτεκτονική.

3.2.1 Raspberry Pi 3 - Model B

Το πρώτο σύστημα που χρησιμοποιήσαμε, είναι το διάσημο Raspberry Pi 3 (σχήμα 3.2). Είναι ένας σχετικά μικρός υπολογιστής ο οποίος αρχικά αναπτύχθηκε στο Ηνωμένο Βασίλειο από το Raspberry Pi Foundation, για να προωθήσει τη διδασκαλία της πληροφορικής στα σχολεία και στις αναπτυσσόμενες χώρες. Το αρχικό μοντέλο έγινε πολύ πιο δημοφιλές από το αναμενόμενο, πουλώντας εκτός της στοχευόμενης αγοράς του, για χρήσεις όπως η ρομποτική κ.α. [28]

Το Pi 3, διαθέτει το Broadcom BCM2837 SoC, το οποίο περιλαμβάνει μια κεντρική μονάδα επεξεργασίας (CPU) βασισμένη στην ARM αρχιτεκτονική και μια on-chip μονάδα επεξεργασίας γραφικών (VideoCore IV). Παρέχει πολλές θύρες για διασύνδεση με άλλες συσκευές, όπως 4 υποδοχές USB 2, HDMI, composite video output κ.α. Διαθέτει επίσης θύρα Ethernet, Wi-Fi 802.11n και Bluetooth. Όσον αφορά το CPU, στελεχώνεται από έναν 4-πύηρνο ARM Cortex-A53 64-bit επεξεργαστή, χρονισμένο στα 1,2 GHz, ενώ όσον αφορά την κύρια μνήμη, διαθέτει 1 GB LPDDR2 RAM που λειτουργεί στα 900 MHz [29]. Secure Digital (SD) κάρτες χρησιμοποιούνται για την αποθήκευση του λειτουργικού συστήματος και τη μνήμη των προγραμμάτων. Από το

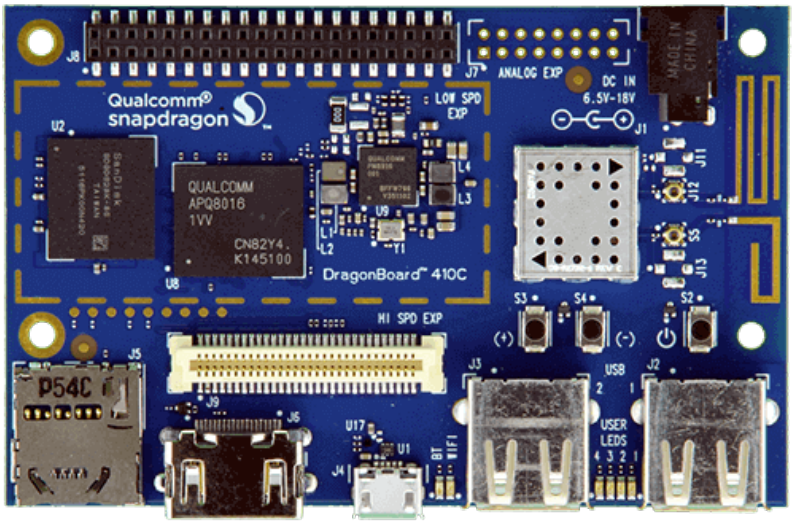


Σχήμα 3.2: Το Raspberry Pi 3 Model B.

Raspberry Foundation, προσφέρεται μια διανομή Linux βασισμένη στο Debian (η Raspbian), που είναι αυτή που χρησιμοποιήσαμε για την αξιολόγηση.
Αξίζει να σημειωθεί ότι παρόλο που το Pi 3 φιλοξενεί έναν 64-bit επεξεργαστή, κατά τη στιγμή της αξιολόγησης δεν υπήρχε κάποιο διαθέσιμο 64-bit λειτουργικό σύστημα. Συνεπώς, οι επεξεργαστές του αξιοποιήθηκαν σε λειτουργία 32-bit.

3.2.2 Dragonboard 410c

Ένα άλλο ενσωματωμένο σύστημα που χρησιμοποιήσαμε είναι το DragonBoard 410c (σχήμα 3.3). Το DragonBoard 410c είναι το πρώτο αναπτυξιακό σύστημα που βασίζεται σε επεξεργαστή της σειράς Qualcomm Snapdragon 400. Διαθέτει προηγμένη ισχύ επεξεργασίας, Wi-Fi, συνδεσιμότητα Bluetooth και GPS, ενώ όλα αυτά είναι συσκευασμένα σε μια πλακέτα μεγέθους πιστωτικής κάρτας. Με βάση τον 64-bit επεξεργαστή Snapdragon 410E, το DragonBoard 410c έχει σχεδιαστεί για να υποστηρίζει την ταχεία ανάπτυξη λογισμικού, την εκπαίδευση καθώς και τη δημιουργία πρωτοτύπων. Όλα αυτά το καθιστούν ιδανικό για τη δημιουργία ενσωματωμένων εφαρμογών και για προϊόντα του Internet of Things, συμπεριλαμβανομένων των φωτογραφικών μηχανών, ιατρικών συσκευών, μηχανημάτων αυτόματης πώλησης, έξυπνων κτηρίων, ψηφιακών πινακίδων, κονσόλων παιχνιδιών κλπ. [30]
Πιο αναλυτικά, το DragonBoard 410c διαθέτει έναν 64-bit quad-core ARM Cortex-A53 επεξεργαστή, χρονισμένο στα 1,2 GHz, 1GB LPDDR3 RAM που λειτουρ-



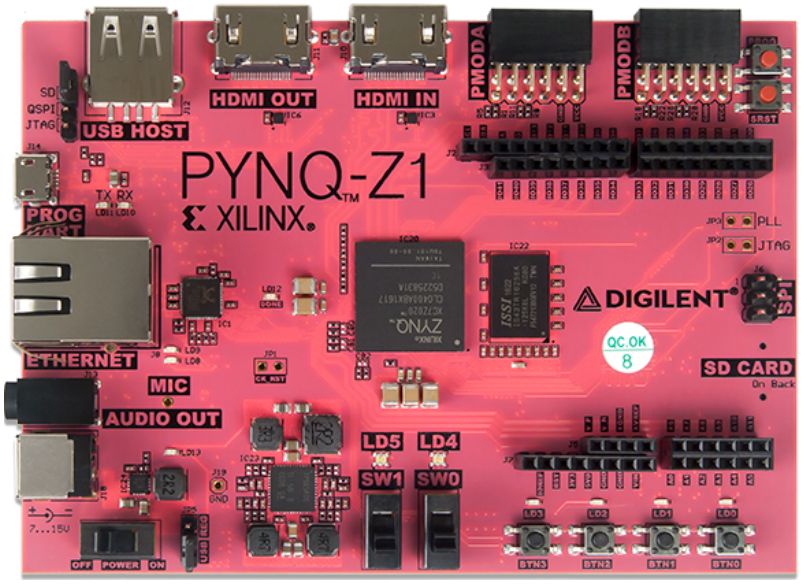
Σχήμα 3.3: Το Dragonboard 410c.

γεί στα 533MHz και 8GB eMMC 4.5 χώρου αποθήκευσης. Όσον αφορά το GPU, διαθέτει το Qualcomm Adreno 306 με υποστήριξη για προηγμένα APIs, συμπεριλαμβανομένων του OpenGL ES 3.0, του OpenCL, του DirectX και της ασφάλειας περιεχομένου.

Σε αυτό το σημείο, είναι σημαντικό να σημειωθεί ότι παρόλο που το Pi 3 και το DragonBoard 410c έχουν τον ίδιο επεξεργαστή, το BCM2837 SoC κατασκευάζεται στην τεχνολογία των 40nm ενώ το Snapdragon 410 SoC κατασκευάζεται στην τεχνολογία των 28nm. Ακόμη, αν και τα δύο συστήματα έχουν 1GB μνήμης RAM, αλλά όπως μπορούμε να δούμε, η μονάδα μνήμης του DragonBoard 410c λειτουργεί σε πολύ χαμηλότερη συχνότητα.

3.2.3 Pynq-Z1

Το PYNQ-Z1 έχει σχεδιαστεί για να χρησιμοποιείται με το PYNQ, ένα σχετικά νέο open source framework της Xilinx, που επιτρέπει στους προγραμματιστές ενσωματωμένων συστημάτων να εκμεταλλεύονται τις δυνατότητες του Zynq All Programmable SoC (APSoC), χωρίς να χρειάζεται να σχεδιάσουν προγραμματιζόμενα λογικά κυκλώματα. Αντίθετα, το APSoC προγραμματίζεται χρησιμοποιώντας την Python και ο κώδικας αναπτύσσεται και ελέγχεται απευθείας στο PYNQ-Z1. Τα προγραμματιζόμενα λογικά κυκλώματα εισάγονται ως βιβλιοθήκες υλικού και προγραμματίζονται μέσω των APIs τους, ακριβώς με τον ίδιο τρόπο που εισάγονται και προγραμματίζονται οι βιβλιοθήκες λογισμικού. [31]



Σχήμα 3.4: Το PYNQ-Z1.

Κύριο συστατικό στοιχείο του είναι το ZYNQ XC7Z020-1CLG400C SoC, το οποίο διαθέτει έναν dual-core Cortex-A9 32-bit επεξεργαστή, χρονισμένο στα 667MHz και έναν ελεγκτή μνήμης DDR3 με 8 κανάλια DMA και 4 AXI3 slave πόρτες υψηλής απόδοσης και μία επαναπρογραμματιζόμενη λογική της οικογένειας Artix-7. Ακόμη, διαθέτει 512MB DDR3 μνήμης και 16MB Quad-SPI Flash.

Αναμφίβολα, τα χαρακτηριστικά του παρόντος συστήματος είναι κατώτερα από αυτά των Pi 3 και του DrangonBoard 410c. Παρόλο που δεν αναμένουμε ότι το PYNQ-Z1 θα ξεπεράσει τις επιδόσεις των άλλων, ο κύριος λόγος για τον οποίο επιλέξαμε να συμπεριληφθεί στην αξιολόγηση μας, είναι ότι στο κεφάλαιο 4 θα αναπτύξουμε ένα framework για την επιτάχυνση αλγορίθμων και εφαρμογών του Spark. Με άλλα λόγια, αυτή η αξιολόγηση θα βοηθήσει στο να διαφανούν καλύτερα τα κέρδη στην απόδοση και την κατανάλωση ενέργειας από τη χρήση επιταχυντών υλικού.

3.3 Εφαρμογές του Spark

Για να αξιολογήσουμε την απόδοση καθώς και την ενεργειακή κατανάλωση των SoCs χαμηλής ισχύος, χρησιμοποιήσαμε ένα σύνολο εφαρμογών του Spark. Πιο συγκεκριμένα, χρησιμοποιήσαμε τρεις αντιπροσωπευτικές εφαρμογές από τον τομέα της μηχανικής εκμάθησης και τρεις από τον τομέα της επεξεργασίας γράφων. Ακολούθως, περιγράψουμε συνοπτικά τη χρήση κάθε εφαρμογής.

3.3.1 Εφαρμογές Μηχανικής Εκμάθησης

- **Linear Regression:** Η γραμμική παλινδρόμηση χρησιμοποιείται για τη μοντελοποίηση της σχέσης μεταξύ μιας βαθμωτής εξαρτώμενης μεταβλητής y και μιας ή περισσότερων ανεξάρτητων μεταβλητών που ορίζονται ως X. Στην περίπτωση που έχουμε μια ανεξάρτητη μεταβλητή, κάνουμε λόγο για απλή γραμμική παλινδρόμηση [32].

- **Logistic Regression:** Η λογιστική παλινδρόμηση, αποτελεί ένα μοντέλο παλινδρόμησης όπου η εξαρτημένη μεταβλητή είναι κατηγορηματική. Μετράει τη σχέση μεταξύ της κατηγορηματικής εξαρτώμενης μεταβλητής και μιας ή περισσοτέρων ανεξάρτητων μεταβλητών, υπολογίζοντας τις πιθανότητες χρησιμοποιώντας μια λογιστική συνάρτηση. [33]

- **K-Means:** Ο K-means είναι ένας αλγόριθμος ομαδοποίησης. Πρόκειται για μια μέθοδο χβαντισμού διανυσμάτων, που χρησιμοποιείται ευρέως στην ανάλυση συστοιχιών και στην εξόρυξη δεδομένων. Επιδιώκει να χωρίσει n παρατηρήσεις σε ομάδες k, στις οποίες κάθε παρατήρηση ανήκει στην ομάδα με τον πλησιέστερο μέσο. [34]

3.3.2 Εφαρμογές Επεξεργασίας Γράφων

- **Connected Components (CC):** Ο CC αλγόριθμος υπολογίζει τα συνδεδεμένα στοιχεία των κορυφών ενός γραφήματος. Αναλυτικότερα, ο αλγόριθμος των συνδεδεμένων στοιχείων επισημαίνει κάθε συνδεδεμένο στοιχείο ενός γράφου με το αναγνωριστικό της χαμηλότερης αριθμημένης κορυφής του. Για παράδειγμα, σε ένα κοινωνικό δίκτυο, τα συνδεδεμένα στοιχεία μπορούν να προσεγγίσουν ομάδες (clusters) [18].

- **PageRank:** Ο PageRank (PR) είναι ένας αλγόριθμος που χρησιμοποιείται για την ταξινόμηση ιστοσελίδων στις μηχανές αναζήτησης. Ο PageRank υπολογίζει τον αριθμό και την ποιότητα των συνδέσμων σε μια σελίδα, για να καθορίσει με μία γρήγορη εκτίμηση πόσο σημαντική είναι αυτή ιστοσελίδα [35].

- **Triangles:** Αυτός ο αλγόριθμος μετρά τα τρίγωνα ενός γράφου. Μια κορυφή είναι μέρος ενός τριγώνου όταν έχει δύο παρακείμενες κορυφές με μια ακμή

μεταξύ τους. Ο αλγόριθμος καθορίζει τον αριθμό των τριγώνων που διέρχονται από κάθε κορυφή, παρέχοντας ένα μέτρο ομαδοποίησης.

3.4 Στήσιμο του Spark

Δεδομένου ότι είδαμε και περιγράψαμε τα χαρακτηριστικά των ενσωματωμένων συστημάτων που χρησιμοποιήσαμε καθώς και των εφαρμογών που χρησιμοποιήσαμε για την αξιολόγηση, είμαστε πλέον έτοιμοι να παρουσιάσουμε τη διαδικασία που ακολουθήσαμε για να στήσουμε το Apache Spark σε αυτά τα συστήματα. Το Spark έγινε deploy αρχικά στις πλατφόρμες Pi3 και DragonBoard 410c, οι οποίες διαθέτουν σχεδόν τους ίδιους πόρους.

Αρχικά, είναι χρήσιμο να αναφέρουμε ότι το Apache Spark μπορεί να μεταφορτωθεί είτε ως pre-built πακέτο ή ως πακέτο πηγαίου κώδικα. Παρόλο που δεν υπάρχει μεγάλη διαφορά μεταξύ των δύο επιλογών, επιλέξαμε τον πηγαίο κώδικα. Με αυτόν τον τρόπο είχαμε τη δυνατότητα να ορίσουμε αναλυτικά τα χαρακτηριστικά του Spark, συμπεριλαμβανομένων της έκδοσης της Scala στην οποία θα κατασκευαστεί το Spark, την ελάχιστης συμβατής έκδοσης του κατανεμημένου συστήματος αρχείων του Hadoop καθώς και τη δυνατότητα ενσωμάτωσης άλλων πακέτων (π.χ. υποστήριξη για το Mesos, για το YARN κ.λπ.). Εκτός από τα παραπάνω, επιλέγοντας να στήσουμε το Spark από πηγαίο κώδικα, είχαμε την ευκαιρία να δοκιμάσουμε και να διαπιστώσουμε εάν είναι δυνατό για το Spark να γίνει build σε αυτά τα ενσωματωμένα συστήματα.

Η διαθέσιμη έκδοση του Apache Spark την περίοδο που ασχοληθήκαμε με την παραπάνω διαδικασία ήταν η 1.6. Για να κάνουμε build το Spark, απαραίτητη προϋπόθεση είναι να υπάρχει ήδη εγκατεστημένη μία έκδοση του Java Development Kit και να έχει διαμορφωθεί ορθά στο σύστημα. Έτσι, το πρώτο πράγμα που έπρεπε να γίνει μετά τη λήψη και την εξαγωγή του πηγαίου κώδικα του Spark, ήταν να μεταφορτώσουμε μια έκδοση του JDK συμβατή με τα συστήματα ARM. Βλέποντας ότι με την Java 7 έπρεπε να ορίσουμε πρόσθετες επιλογές όπως "-XX: MaxPermSize = 512M" για αποφυγή σφαλμάτων και warnings κατά το build, επιλέξαμε την Java 8, όπου δεν είχαμε τους παραπάνω περιορισμούς. Δεδομένου ότι το Raspbian OS είναι ένα 32-bit λειτουργικό σύστημα, ένα 32-bit πακέτο του JDK ελήφθη, ενώ για το Dragonboard 410c το οποίο συνοδεύεται από ένα 64-bit λειτουργικό σύστημα, ένα πακέτο 64-bit

ελήφθη. Μετά τη λήψη και την εξαγωγή των αντίστοιχων πακέτων, οι μεταβλητές $JAVA\_HOME$ και $PATH$ έπρεπε να οριστούν στο *.bashrc* αρχείο (που βρίσκεται στον αρχικό κατάλογο του συνδεδεμένου χρήστη), για να είναι ορατή η Java από το Spark. Οι εντολές που προστέθηκαν στο αρχείο *.bashrc* εμφανίζονται ακολούθως:

```
export JAVA_HOME="<path/to/jdk/folder>"
export PATH="$PATH:$JAVA_HOME/bin"
```

Listing 3.1: Java configuration

Έπειτα, κάνοντας source το .bashrc (π.χ. sudo source .bashrc) για να εφαρμοστούν οι αλλαγές, συνεχίσαμε με τα βήματα που απαιτούνται για το στήσιμο του Spark. Το Spark έρχεται πακεταρισμένο με μια έκδοση του Maven, το οποίο διευκολύνει την στήσιμο και το deploying του πρώτου. Περιλαμβάνει ένα script που μεταφορτώνει τοπικά και ρυθμίζει αυτόματα όλα τα απαραίτητα στοιχεία για το building του Spark (Maven, Scala, και Zinc) [18]. Οι πληροφορίες σχετικά με το project καθώς και όλες οι διαμορφώσεις και τα πακέτα που απαιτούνται για την κατασκευή του project αυτού, περιέχονται σε ένα αρχείο XML το *pom.xml*. Η προεπιλεγμένη έκδοση της Scala ήταν 2.10 και προκειμένου να την αλλάξουμε σε 2.11 εκτελέσαμε την ακόλουθη εντολή:

```
./dev/change-scala-version.sh 2.10
```

Τέλος, ξεκινήσαμε το στήσιμο του Apache Spark, εκτελώντας την ακόλουθη εντολή:

```
build/mvn -Pyarn -Phadoop-2.6 -Dhadoop.version=2.6.0 -Dscala-2.11 -
    DskipTests clean package
```

Με αυτόν τον τρόπο, ορίσαμε την 2.6 ως την ελάχιστη συμβατή έκδοση του Hadoop για το Spark, την έκδοση της Scala σε 2.11 για την μεταγλώττιση όλων των επιμέρους στοιχείων, ενώ τέλος προσθέσαμε υποστήριξη για το YARN.

3.4.1 Προκλήσεις από τη Διαμόρφωση του Spark σε Ενσωματωμένα Συστήματα

Παρόλο που η μεταγλώττιση του πηγαίου κώδικα του Spark με Java 8 δεν απαιτεί πρόσθετες ρυθμίσεις, εμφανίστηκαν σφάλματα σχετικά με το μέγεθος του διαθέσιμου σωρού (heap) που αφορούσαν τα JVMs (Java Virtual Machines). Πιο συγκεκριμένα, η Java δεν μπορούσε να δεσμεύσει το προεπιλεγμένο μέγεθος σωρού και για αυτόν τον λόγο εμφανίζονταν exceptions. Για να ξεπεραστεί αυτό το πρόβλημα, ορίσαμε

ρητά το μέγιστο μέγεθος του σωρού που θα μπορούσε να διατεθεί σε 512MB και το μέγιστο μέγεθος της συνολικής μνήμης που θα μπορούσε να δεσμεύσει στα 800MB, εκτελώντας την ακόλουθη εντολή στο τερματικό:

```
export MAVEN_OPTS="-Xmx=800M -XX: MaxHeapSize=512M"
```

Ένα άλλο πρόβλημα που αντιμετωπίσαμε κατά το στήσιμο του Spark στα ενσωματωμένα συστήματα, ήταν ότι το Snappy, μια βιβλιοθήκη συμπίεσης/αποσυμπίεσης που χρησιμοποιείται για να γίνει build το Spark, δεν υποστήριζε την αρχιτεκτονική ARM. Ως λύση στο πρόβλημα, τροποποιήσαμε το pom.xml αρχείο, το οποίο περιέχει όλες τις εξαρτήσεις για την κατασκευή του Spark και ορίσαμε τη λήψη και τη χρήση της τελευταίας έκδοσης του Snappy, η οποία μέχρι τότε ήταν η 1.1.2.4 και στην οποία προστέθηκε υποστήριξη για την ARM αρχιτεκτονική. Η τροποποίηση που έγινε παρουσιάζεται ακολούθως:

```
### old XML tag value ###
<snappy.version>1.1.2.3</snappy.version>
### new XML tag value ###
<snappy.version>1.1.2.4</snappy.version>
```

Ακολουθώντας τα παραπάνω βήματα, το build του Spark ήταν επιτυχές.

3.4.2 Προκλήσεις από την Εκτέλεση του Spark σε Ενσωματωμένα Συστήματα

Ένας χρήστης μπορεί εύκολα να ελέγξει αν όλα λειτουργούν σωστά, εκτελώντας οποιοδήποτε από τα παραδείγματα που περιλαμβάνονται στο Spark ή πληκτρολογώντας την εντολή: $./bin/spark - shell.sh$ (από τον αρχικό φάκελο του Spark) στο τερματικό, για να ανοίξει ένα διαδραστικό Spark Shell και να αξιολογήσει κώδικα σε Scala ή Java. Στην περίπτωσή μας, εκτελώντας το spark-shell συνειδητοποιήσαμε ότι υπήρξε και πάλι πρόβλημα με την Java. Όπως και πριν, οι διεργασίες των JVMs προσπαθούσαν να δεσμεύσουν περισσότερη από την διαθέσιμη μνήμη. Το πρόβλημα εντοπίζεται ακριβώς στο ότι το Spark ορίζει τη μέγιστη διαθέσιμη μνήμη για κάθε JVM στο 1GB, ως προεπιλογή. Όμως, και τα δύο συστήματα (το Pi 3 και το DragonBoard 410c) έχουν μόνο 1GB (το πραγματικό διαθέσιμο μέγεθος είναι μικρότερο από 1GB) μνήμης RAM ενώ ένα μέρος της μνήμης είναι δεσμευμένο από το λειτουργικό σύστημα. Όπως φαίνεται και στο κεφάλαιο **??**, το Spark διαθέτει μία

πληθώρα επιλογών για τη διαμόρφωση και τον ορισμό των πόρων που θα χρησιμοποιεί κατά την υποβολή μιας εφαρμογής σε ένα cluster. Αναλυτικότερα, στο bash script $/conf/spark-env.sh$ υπάρχουν επιλογές για τη ρύθμιση της μνήμης του driver προγράμματος, του executor, του master, των workers και του daemon, όπως παρουσιάζονται ακολούθως:

Option	Comments
SPARK_EXECUTOR_MEMORY	Memory per Executor (Default: 1G)
SPARK_DRIVER_MEMORY	Memory for Driver (Default: 1G)
SPARK_WORKER_MEMORY	To set how much total memory workers have to give executors
SPARK_DAEMON_MEMORY	To allocate to the master, worker and history server themselves (Default: 1G)

Πίνακας 3.1: Επιλογές διαμόρφωσης μνήμης στο Spark.

Από τις παραπάνω επιλογές, ορίσαμε τη μνήμη του driver στα 800MB για την αξιολόγηση των εφαρμογών του Spark σε local mode και το πρόβλημα επιλύθηκε.

```
export SPARK_DRIVER_MEMORY=800MB
```

Σημειώνεται, ότι η ίδια ρύθμιση θα μπορούσε να γίνει και μέσω του αρχείου $spark-defaults.conf$, όπως φαίνεται στη συνέχεια:

```
spark.driver.memory=800MB
```

Ωστόσο, για την αξιολόγηση των εφαρμογών στο cluster mode του Spark, θα πρέπει επιπλέον να ρυθμίσουμε τη μνήμη των master και worker κόμβων. Τόσο το Pi 3 όσο και το DragonBoard 410c έχουν έναν τετραπύρηνο επεξεργαστή, που σημαίνει ότι μπορούν να φιλοξενήσουν μέχρι και 4 workers (έναν για κάθε CPU core). Επίσης, είναι σημαντικό να σημειωθεί ότι κάθε master και worker κόμβος εκτελείται σε διαφορετικό JVM process. Το Spark, περιορίζει την ελάχιστη ποσότητα μνήμης που διατίθεται σε κάθε process στα 475MB. Με αυτόν τον τρόπο, ακόμα και στην απλούστερη περίπτωση δημιουργίας ενός Spark cluster που αποτελείται από έναν master και έναν worker κόμβο, πρέπει να υπάρχουν διαθέσιμα $475MB + 475MB = 950MB$ μνήμης. Ωστόσο, λόγω του ότι μόνο 800-850 MB μνήμης RAM είναι πραγματικά διαθέσιμα στα ενσωματωμένα συστήματα που χρησιμοποιήσαμε, δεν ήταν δυνατό το στήσιμο ενός Spark cluster και η αξιολόγηση των εφαρμογών σε αυτό το mode.

Τέλος, όσον αφορά την πλατφόρμα PYNQ-Z1, είναι εμφανές ότι οι διαθέσιμοι πόροι σε πυρήνες CPU και μνήμη RAM είναι πολύ λιγότεροι από αυτούς του Pi 3 και του DragonBorad 410c. Για το λόγο αυτό, δεν είχε νόημα να προσπαθήσουμε να στήσουμε το Spark από πηγαίο κώδικα, και αντ' αυτού χρησιμοποιήσαμε μια pre-built έκδοση του Spark με τα ίδια χαρακτηριστικά (πχ. Scala 2.11, Hadoop 2.6 κλπ.).

4 Το Spark σε ένα Pynq Cluster

4.1 Εισαγωγή

Οι αυξανόμενες απαιτήσεις τόσο στη συνολική απόδοση όσο και στην ενεργειακή αποδοτικότητα έχουν οδηγήσει τις εταιρείες στη υιοθέτηση νέων πρακτικών μέσω της χρήσης επιταχυντών FPGA σε επίπεδο datacenter. Πολλές κολοσσιαίες εταιρείες, όπως η Amazon και η Intel, έχουν ήδη κάνει κινήσεις στην αγορά των FPGA. Για παράδειγμα, η Intel εξαγόρασε πρόσφατα την Altera [36], μια πολύ γνωστή εταιρεία κατασκευής FPGAs, ενώ η Amazon παρέχει ήδη εμπορικά τις εικονικές μηχανές F1 που περιλαμβάνουν εκτός του κλασικού CPU και μία προγραμματιζόμενη λογική [14]. Για το λόγο αυτό, σκοπός μας είναι να κατασκευάσουμε ένα ετερογενές σύμπλεγμα υπολογιστών που θα ήταν ικανό να επιταχύνει τα υπολογιστικά απαιτητικά τμήματα των αλγορίθμων. Έτσι, σε αυτό το κεφάλαιο θα παρουσιάσουμε ολόκληρη τη διαδικασία ρύθμισης, δημιουργίας και διαμόρφωσης αυτού του συμπλέγματος που αποτελείται κυρίως από PYNQ-Z1 κόμβους, οι οποίοι ενσωματώνουν και μία επαναπρογραμματιζόμενη λογική, με έναν master κόμβο βασισμένο στην αρχιτεκτονική της Intel. Το Spark θα εγκατασταθεί σε όλους τους κόμβους του cluster και θα διαμορφωθεί για να εκτελείται σε αυτούς. Επιπλέον, θα παρουσιαστεί ένα αποτελεσματικό framework για την απρόσκοπτη χρήση επιταχυντών υλικού στο Spark, μαζί με τα διαθέσιμα APIs που θα παρέχει. Τέλος, θα πραγματοποιηθούν μετρήσεις για την αξιολόγηση του συνολικού έργου.

4.2 PYNQ-Z1

Ένα από τα σημαντικότερα ερωτήματα που προκύπτουν είναι το γιατί επελέγη το PYNQ-Z1 για τη δημιουργία του cluster. Η απάντηση είναι ότι τα χαρακτηριστικά

του πληρούν τις απαιτήσεις ενός τέτοιου project. Αναλυτικότερα, τα χαρακτηριστικά βάσει των οποίων λάβαμε την απόφαση είναι τα εξής:

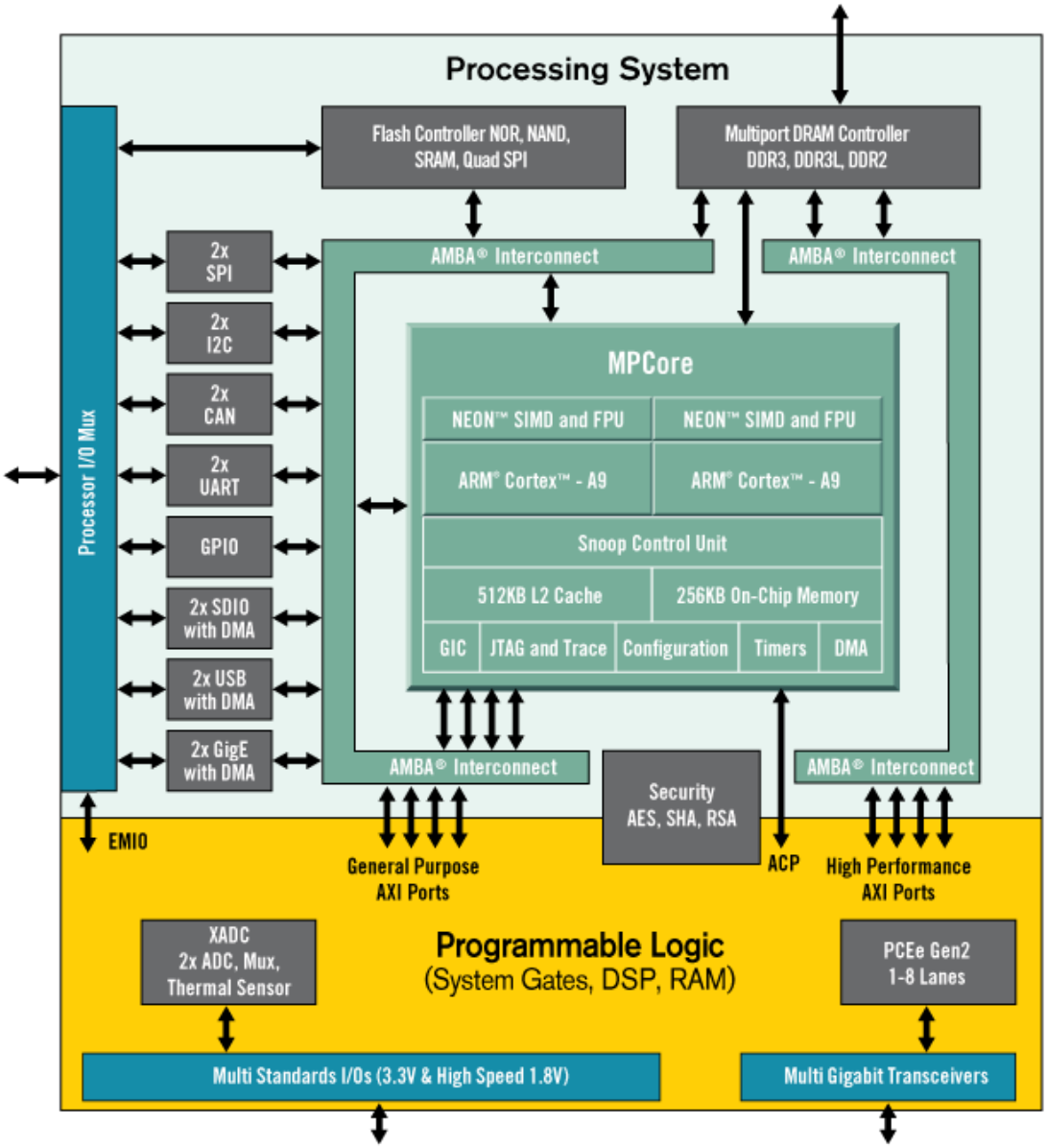
- **Zynq APSoC:** Αρχικά, το PYNQ είναι βασισμένο στο Zynq All Programmable SoC και συγκεκριμένα στη σειρά Zynq-7000. Όλες οι συσκευές Zynq-7000 είναι εξοπλισμένες με έναν διπύρηνο επεξεργαστή Cortex-A9, ο οποίος είναι στενά συνδεδεμένος με μία Artix-7 ή Kintex-7 επαναπρογραμματιζόμενη λογική, για εξαιρετικές επιδόσεις ανά watt και μέγιστη ευελιξία σχεδιασμού. Με τον τρόπο αυτό, ένας χρήστης μπορεί να συνδυάζει τον προγραμματισμό σε επίπεδο λογισμικού που παρέχουν οι ARM επεξεργαστές με τον προγραμματισμό σε επίπεδο υλικού που παρέχεται από την επαναπρογραμματιζόμενη λογική. Στην εικόνα 4.1 φαίνεται το μπλοκ διάγραμμα της σειράς Zynq-7000.

  Ωστόσο, παρατηρείται ότι μόνο τα υπολογιστικά απαιτητικά μέρη ενός αλγορίθμου μπορούν πραγματικά να επιταχυνθούν. Συνεπώς, η ιδανική πλατφόρμα για την επιτάχυνση ενός αλγορίθμου πρέπει να υποστηρίζει τον προγραμματισμό τόσο σε επίπεδο λογισμικού όσο και σε επίπεδο υλικού, όπως αναφέρθηκε προηγούμενως. Είναι προφανές ότι η πλατφόρμα Zynq είναι ιδανική για το σκοπό αυτό και γίνεται εύκολα κατανοητό ότι αυτός ήταν ένας από τους κύριους λόγους χρησιμοποίησης της πλατφόρμας PYNQ.

- **Low power:** Όπως έχουμε ήδη δει στο κεφαλαίο 3, το PYNQ-Z1 ανήκει στις ενσωματωμένες πλατφόρμες χαμηλής κατανάλωσης. Η κατανάλωση ενέργειάς του δεν ξεπερνά το όριο των 3.2W, ενώ φιλοξενεί μία πλήρη διανομή του λειτουργικού συστήματος Ubuntu.

  Κύριος σκοπός της εργασίας αυτής είναι η διερεύνηση εναλλακτικών για τη βελτίωση της απόδοσης των datacenters με παράλληλη μείωση στη συνολική κατανάλωση ενέργειας. Συνεπώς, μέχρι στιγμής, υπάρχουν δύο λόγοι που καθιστούν το PYNQ-Z1 ιδανικό για το σκοπό μας.

- **Small size:** Όλα τα συστατικά στοιχεία του PYNQ, συμπεριλαμβανομένων των εξωτερικών διεπαφών εισόδου/εξόδου, καθώς και του APSoC είναι πακεταρισμένα σε μία μικρή πλακέτα (printed board circuit – PCB) μεγέθους σχεδόν όσο τρεις φορές το μέγεθος μιας πιστωτικής κάρτας. Με τον τρόπο



Σχήμα 4.1: Μπλοκ διάγραμμα της σειράς Zynq-7000.

αυτό, όχι μόνο εξοικονομεί χώρο, αλλά μπορεί επίσης να ανταπεξέλθει στις σημερινές ανάγκες κλιμάκωσης.

- **Python libraries:** Το PYNQ-Z1 είναι μία μια πλατφόρμα βασισμένη σε ένα νέο project ανοιχτού κώδικα της Xilinx, ονόματι PYNQ. Το κύριο πλεονέκτημα αυτού είναι ότι επιτρέπει στους χρήστες να σχεδιάζουν εύκολα ενσωματωμένα συστήματα χρησιμοποιώντας τα Zynq APSoCs της Xilinx. Οι επιταχυντές υλικού εισάγονται ως βιβλιοθήκες υλικού (γνωστές ως overlays), ενώ

ολόκληρη η εφαρμογή συμπεριλαμβανομένης και της διεπαφής των επιταχυντών είναι γραμμένες σε Python.

Η Python χρησιμοποιείται όλο και περισσότερο στον τομέα της ανάλυσης μεγάλων δεδομένων ως μια ισχυρή, ευέλικτη γλώσσα ανοιχτού κώδικα που είναι εύκολη στη μάθηση, στη χρήση και έχει ισχυρές βιβλιοθήκες για τον χειρισμό και την ανάλυση δεδομένων [38]. Ως εκ τούτου, πιστεύουμε στη μελλοντική οδήγηση επιταχυντών υλικού μέσω βιβλιοθηκών της Python.

4.3 Το Spark στο Pynq (SPynq) Cluster

Σε αυτή την ενότητα θα παρουσιάσουμε όλη την διαδικασία δημιουργίας και διαμόρφωσης του SPynq cluster. Για το cluster, προμηθευτήκαμε τέσσερεις PYNQ-Z1 υπολογιστές καθώς και ένα PC με έναν Intel i5 επεξεργαστή, ενώ για τη δικτύωση των κόμβων χρειάστηκε να προμηθευτούμε και ένα μικρό switch.
Για να ρυθμίσετε τους PYNQ-Z1 κόμβους, συμβουλευτήκαμε τον οδηγό γρήγορης εκκίνησης του Pynq. Έπειτα, κατεβάσαμε το PYNQ image της Xilinx μεταβαίνοντας στο GitHub. Αφού μεταφορτώσαμε τα δεδομένα του image σε τέσσερις SD κάρτες, μπορέσαμε τελικά να εκκινήσουμε τους κόμβους PYNQ-Z1. Στη συνέχεια θα παρουσιάσουμε αναλυτικότερα τις ρυθμίσεις στις οποίες προβήκαμε:

4.3.1 Ρυθμίσεις δικτύου

Οι PYNQ κόμβοι, από προεπιλογή, λαμβάνουν τη στατική *192.168.2.99* IPv4 διεύθυνση. Για να αποφευχθεί το ενδεχόμενο δύο ή περισσότεροι κόμβοι να έχουν την ίδια IP, τροποποιήσαμε τη διαμόρφωση δικτύου κάθε κόμβου και ορίσαμε διαφορετικές IP διευθύνσεις για κάθε ένα από αυτούς. Αναλυτικότερα, τροποποιήσαμε το αρχείο *static* κάθε κόμβου το οποίο βρίσκεται στο path */etc/network/interfaces.d* των Linux. Οι τροποποιήσεις παρουσιάζονται ακολούθως:

```
### Old configuration for the static IP ###

address 192.168.2.99

### New configuration for the static IP ###

```

```
address 192.168.1.<231-234 depending on the pynq node and 203 for the
    master>
gateway 192.168.1.1
```

Listing 4.1: Ip configuration

Ωστόσο, οι κόμβοι PYNQ-Z1 παρέχουν επιπλέον την πληροφορία για την προεπιλεγμένη στατική διεύθυνση IP στο αρχείο *dhclient.conf*, το οποίο βρίσκεται κάτω από τον κατάλογο */etc/dhcp*. Στη συνέχεια, παρουσιάζεται η τελική διαμόρφωση του DHCP:

```
### Old fixed-address value ###

alias {   interface "eth0";
  fixed-address 192.168.99;
  option subnet-mask 255.255.255.0; }

### New DHCP configuration ###

option rfc3442-classless-static-routes code 121 = array of unsigned
    integer 8;

send host-name = gethostname();

request subnet-mask, broadcast-address, time-offset, routers,
domain-name, domain-name-servers, domain-search, host-name,
dhcp6.name-servers, dhcp6.domain-search,
netbios-name-servers, netbios-scope, interface-mtu,
rfc3442-classless-static-routes, ntp-servers,
dhcp6.fqdn, dhcp6.sntp-servers;

timeout 30;
retry 10;
reboot 3;
select-timeout 0;

alias {   interface "eth0";
  fixed-address 192.168.231;
  option subnet-mask 255.255.255.0; }
```

Listing 4.2: DHCP configuration

Επιπλέον, ως προαιρετικό βήμα, δημιουργήσαμε DNS εγγραφές για τους κόμβους του cluster, προκειμένου να μπορούμε να αναφερόμαστε σε αυτούς με βάση τα hostnames τους και όχι τις IP διευθύνσεις τους. Η διαδικασία δημιουργίας των εγγραφών DNS παραλείπεται καθώς είναι προαιρετική και δεν έχει κάποια σχέση με την υλοποίηση του cluster. Ωστόσο, θα αναδείξουμε τη διαδικασία που ακολουθήσαμε για να αλλάξουμε το hostname κάθε PYNQ κόμβου. Η Xilinx παρέχει ένα script που ονομάζεται *hostname.sh* και χρησιμοποιείται ακριβώς για την τροποποίηση του hostname ενός PYNQ κόμβου. Το script αυτό, βρίσκεται κάτω από τον κατάλογο */home/xilinx/scripts* του λειτουργικού συστήματος. Εκτελώντας την ακόλουθη εντολή στο τερματικό κάθε κόμβου, τροποποιήσαμε τα hostnames των PYNQ-Z1, ενώ προκειμένου να εφαρμοστούν οι αλλαγές ήταν απαραίτητη επανεκκίνηση κάθε συστήματος.

```
$ sudo ./hostname.sh <new hostname>
```

Listing 4.3: The command for altering PYNQs' hostnames

Τέλος, παρουσιάζουμε τη διαμόρφωση δικτύου του cluster μας:

Hostname	**IPv4 address**	**Platform**
vinemaster	192.168.1.203	Intel i5 based
pynq1	192.168.1.231	PYNQ-Z1
pynq2	192.168.1.232	PYNQ-Z1
pynq3	192.168.1.233	PYNQ-Z1
pynq4	192.168.1.234	PYNQ-Z1

Πίνακας 4.1: Διαμόρφωση δικτύου του cluster.

4.3.2 Ρυθμίσεις ασφαλείας

Δεδομένου ότι σκοπός μας ήταν να δημιουργήσουμε ένα cluster που θα είναι προσβάσιμο μέσω του Διαδικτύου, έπρεπε να διασφαλίσουμε ότι πληρούνται ορισμένα πρότυπα ασφαλείας. Για το λόγο αυτό, τροποποιήσαμε την πολιτική του τείχους προστασίας κάθε μεμονωμένου κόμβου από τον οποίο αποτελείται το cluster, χρησιμοποιώντας την *iptables* εντολή. Επιπλέον, τροποποιήσαμε το αρχείο *sshd_config* που βρίσκεται κάτω από τον κατάλογο */etc/ssh* των Linux, προκειμένου να επιτρέπεται ο έλεγχος ταυτότητας μέσω του πρωτοκόλλου SSH μόνο για χρήστες που παρέχουν ένα ιδιωτικό κλειδί. Το αρχείο τροποποιήθηκε έτσι ώστε να συμπεριληφθούν τελικά οι παρακάτω επιλογές:

```
1 ###### sshd_config #######
2
3 PermitRootLogin without-password
4 PubkeyAuthentication yes
5 PasswordAuthentication no
```

Listing 4.4: Modifications done in the *sshd_config* file

Είναι σημαντικό να σημειωθεί ότι επιτρέψαμε τις κενές «φράσεις κλειδιά» για τη σύνδεση μέσω δημόσιου κλειδιού. Ο κύριος σκοπός αυτού θα εξηγηθεί αργότερα, στο κομμάτι διαμόρφωσης του Spark.
Τέλος, δημιουργήσαμε ένα ζεύγος δημόσιου/ιδιωτικού κλειδιού rsa με κενή «φράση πρόσβασης» χρησιμοποιώντας την ακόλουθη εντολή:

```
1 #### linux command that generates a private/public rsa key pair ####
2
3 $ ssh-keygen
```

Listing 4.5: Generating a private/public rsa key pair

Στη συνέχεια, κρατήσαμε το ιδιωτικό μέρος του κλειδιού στον κόμβο που βασίζεται στον Intel επεξεργαστή, ο οποίος, όπως θα δούμε αργότερα, θα είναι ο master κόμβος του cluster (όσον αφορά το Spark) και τέλος αντιγράψαμε το δημόσιο μέρος σε κάθε έναν PYNQ-Z1 κόμβο στον κατάλογο */root/.ssh*, προσθέτοντάς το στο αρχείο *authorized_keys*.
Αξίζει να σημειωθεί ότι όλες οι παραπάνω διαμορφώσεις σχετίζονται με τον root χρήστη των Linux και φυσικά ήταν κάτι το οποίο έγινε σκοπίμως. Ο κύριος λόγος πίσω από αυτό, είναι ότι οι ενσωματωμένες βιβλιοθήκες της Xilinx, για τη χρήση της επαναπρογραμματιζόμενης λογικής, απαιτούν επαυξημένα δικαιώματα. Με αυτόν τον τρόπο, η εκτέλεση όλων των εντολών πρέπει να γίνεται από τον root χρήστη.

4.3.3 Ρυθμίσεις στο Spark

Πριν ξεκινήσουμε με το στήσιμο του Spark, κατεβάσαμε και εγκαταστήσαμε το JDK framework σε όλους τους κόμβους του cluster. Έπειτα, κατεβάσαμε μια pre-built έκδοση του Apache Spark, το οποίο αντιγράψαμε και εξήγαμε κάτω από το φάκελο /home/xilinx κάθε κόμβου. Επιπλέον, για να εξάγουμε όλες τις απαραίτητες μεταβλητές περιβάλλοντος, αρχικά προσθέσαμε τις μεταβλητές *$JAVA_HOME*,

$SPARK_HOME και *$PATH* στο αρχείο */etc/environment* και στη συνέχεια επανεκκινήσαμε κάθε σύστημα προκειμένου να εφαρμοστούν οι αλλαγές. Με την προσθήκη όλων των παραπάνω, το αρχείο */etc/environment* των πλακετών PYNQ-Z1 απέκτησε την ακόλουθη μορφή (δεν τροποποιήσαμε ολόκληρο το αρχείο καθώς άλλες καταχωρήσεις θα μπορούσαν να επηρεάσουν τη λειτουργία του συστήματος):

```
###### Contents of /etc/environment file ######

#Java
JAVA_HOME="/usr/lib/jvm/jdk1.8.0_111"

#Spark
SPARK_HOME="/home/xilinx/spark-2.1.1"

#PATH
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr
    /games:/usr/local/games:/home/xilinx/spark-2.1.1/bin:/home/xilinx/
    spark-2.1.1/sbin:/usr/lib/jvm/jdk1.8.0_111/bin"
```

Listing 4.6: Setting environment variables on the PYNQ-Z1 nodes

Ακολούθως παραθέτουμε το ίδιο αρχείο ρυθμίσεων του κόμβου που βασίζεται στην Intel αρχιτεκτονική:

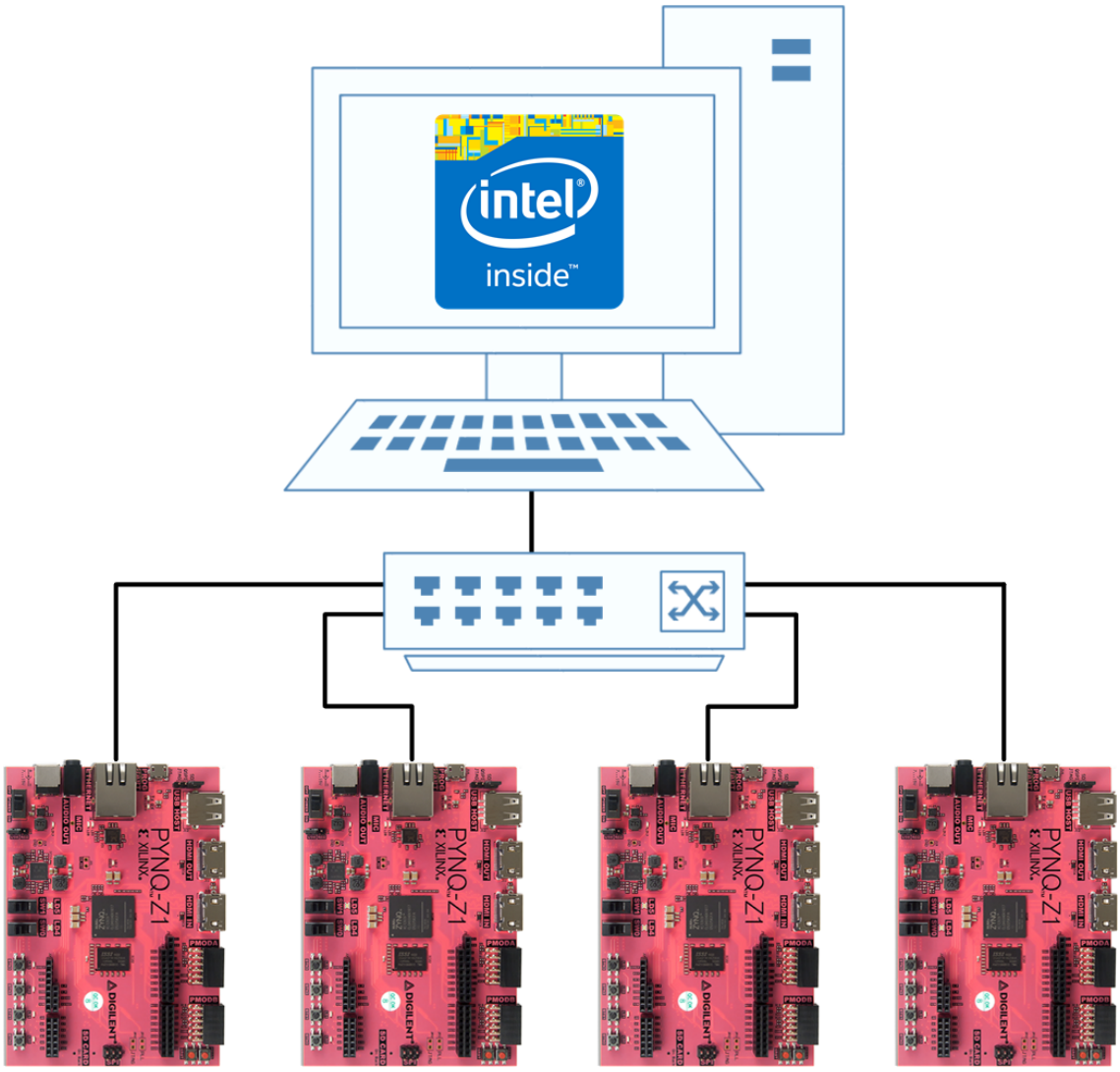
```
###### Contents of /etc/environment file ######

#Spark
export SPARK_HOME=/home/xilinx/spark-2.1.1

#Java
export JAVA_HOME=/usr/lib/jvm/jdk1.8.0

#PATH
export PATH=$PATH:/home/xilinx/spark-2.1.1/bin:/home/xilinx/spark
    -2.1.1/sbin:/usr/lib/jvm/jdk1.8.0/bin
```

Listing 4.7: Setting environment variables on the Intel i5 based node

Όπως μπορεί κανείς να παρατηρήσει, ο κατάλογος κάτω από τον οποίο βρίσκεται το Spark είναι ο ίδιος για όλους τους εμπλεκόμενους κόμβους. Όσον αφορά την εκτέλεση εφαρμογών στο Standalone cluster mode του Spark, ο Spark Master,

αφού συνδεθεί με τους Workers, ψάχνει το αντίστοιχο πακέτο Spark σε κάθε Worker κόμβο, στην ίδια διαδρομή με τη δική του. Επομένως, το deployment του Spark σε διαφορετικό path, θα προκαλούσε προβλήματα κατά την εκκίνηση των Worker διεργασιών.

Με βάση τα παραπάνω, είμαστε έτοιμοι να προχωρήσουμε στις συγκεκριμένες ρυθμίσεις του Spark για το επιθυμητό cluster. Η εικόνα 4.2 απεικονίζει το τελικό επιθυμητό σχήμα του cluster. Ρυθμίσεις θα γίνουν στον κόμβο που βασίζεται στην Intel αρχιτεκτονική, προκειμένου να φιλοξενεί τον Spark Master, ενώ κάθε PYNQ-Z1 κόμβος θα διαμορφωθεί έτσι ώστε να εκκινείται ένας Spark Worker. Τυπικά, όλες οι εφαρμογές θα υποβάλλονται στον Master κόμβο, ενώ οι PYNQ κόμβοι θα αναλαμβάνουν την εκτέλεση των εργασιών που θα τους ανατίθενται.



Σχήμα 4.2: Το προτεινόμενο σχήμα του cluster

Για το σκοπό αυτό, έπρεπε να προβούμε στις απαραίτητες ρυθμίσεις ώστε ο Master κόμβος να συνδέεται με τους Slave κόμβους και αντίστροφα. Ακολούθως, παρουσιάζουμε κάθε περίπτωση χωριστά:

Ρυθμίσεις στον Spark Master Κόμβο

Στον Master κόμβο τροποποιήσαμε τα ακόλουθα τέσσερα αρχεία: *slaves*, *spark-env.sh*, *spark-defaults.conf* και *log4j.properties*.

- **slaves:** Στο Spark 2.0 και σε όλες τις νεότερες εκδόσεις του, κάτω από τον κατάλογο *conf*, υπάρχει το αρχείο *slaves*. Αυτό το αρχείο περιέχει τα hostnames και τις IP διευθύνσεις όλων των κόμβων στους οποίους θα εκκινείται ένας οι περισσότεροι Spark Workers. Έχοντας διαμορφώσει το αρχείο *slaves*, είμαστε σε θέση να εκκινήσουμε ή να σταματήσουμε το Spark cluster, χρησιμοποιώντας οποιοδήποτε από τα διαθέσιμα scripts του Spark, που βρίσκεται κάτω από τον κατάλογο SPARK_HOME/sbin. Ο πίνακας **??** εμφανίζει όλα τα διαθέσιμα scripts του Spark.

Script name	Description
sbin/start-master.sh	Starts a master instance on the machine the script is executed on.
sbin/start-slaves.sh	Starts a slave instance on each machine specified in the conf/slaves file.
sbin/start-slave.sh	Starts a slave instance on the machine the script is executed on.
sbin/start-all.sh	Starts both a master and a number of slaves as described above.
sbin/stop-master.sh	Stops the master that was started via the bin/start-master.sh script.
sbin/stop-slaves.sh	Stops all slave instances on the machines specified in the conf/slaves file.
sbin/stop-all.sh	Stops both the master and the slaves as described above.

Πίνακας 4.2: Διαθέσιμα scripts για έναρξη/διακοπή λειτουργίας του cluster στο Spark

Συνεπώς στην περίπτωσή μας, στον κόμβο που βασίζεται στην Intel αρχιτεκτονική, τροποποιήσαμε το αρχείο slaves του Spark ώστε να περιέχει τα hostnames

όλων των PYNQ-Z1 κόμβων. Τα περιεχόμενα του αρχείου παρουσιάζονται ακολούθως:

```
1 ##### Contents of conf/slaves file #####
2
3 pynq1
4 pynq2
5 pynq3
6 pynq4
```

Listing 4.8: Setting the Worker nodes of the Spark Cluster

- **spark-env.sh:** Έπειτα, τροποποιήσαμε το αρχείο *spark-env.sh* ώστε τελικά να περιέχει τις ακόλουθες εντολές:

```
1 ##### Master - conf/spark-env.sh #####
2
3 export SPARK_DRIVER_MEMORY=505m
4 export SPARK_DAEMON_MEMORY=505m
5 export SPARK_MASTER_HOST=192.168.1.203
6 export SPARK_LOCAL_IP=192.168.1.203
```

Listing 4.9: The spark-env.sh file of the Master Node

Αρχικά, η μνήμη του προγράμματος οδήγησης και του δαίμονα του Spark ορίστηκε στα 505MB. Η ποσότητα της μνήμης που δόθηκε, ορίστηκε μετά από πολλές δοκιμές και είναι τόσο περιορισμένη λόγω του αντίστοιχου περιορισμού στη μνήμη των Worker κόμβων. Οι PYNQ-Z1 κόμβοι έχουν μόλις 512MB μνήμης RAM και συνεπώς η μέγιστη μνήμη που μπορούμε να διαθέσουμε για στους Spark workers οριοθετείται από αυτό το μέγεθος. Ακόμη, η ποσότητα της μνήμης του προγράμματος οδήγησης του Spark δεν μπορεί να υπερβαίνει την προαναφερθείσα τιμή. Κατά την υποβολή μιας εφαρμογής, η διαθέσιμη μνήμη της είναι αυτή που έχει οριστεί ως μνήμη του προγράμματος οδήγησης. Έτσι, σε περίπτωση όπου η μνήμη του προγράμματος οδήγησης είναι μεγαλύτερη από τη μνήμη των workers, ο διαχειριστής συμπλέγματος δεν θα μπορεί να αποδεχθεί τους απαιτούμενους πόρους, δεδομένου ότι δεν υπάρχουν worker κόμβοι με αυτούς τους διαθέσιμους πόρους.

Τέλος, ορίστηκαν οι μεταβλητές Master Host και Master IP στην τιμή της IP του Intel κόμβου 4.3.1.

- **spark-defaults.conf:** Σε αυτό το αρχείο, αρχικά ορίστηκε το url του Spark master. Επισημαίνεται ότι δεν ήταν απαραίτητο να προστεθεί στο αρχείο διαμόρφωσης, ωστόσο, σε διαφορετική περίπτωση θα έπρεπε να ορίζεται σε κάθε υποβολή μίας εφαρμογής. Έπειτα, ενεργοποιήσαμε την καταγραφή των συμβάντων εκτέλεσης και τελικά ορίσαμε τον κατάλογο και το αρχείο στο οποίο θα αποθηκεύονται. Ο ίδιος κατάλογος χρησιμοποιήθηκε και ως κατάλογος καταγραφής του Spark History Server, όπως παρουσιάζεται στη συνέχεια.

  Αξίζει να σημειωθεί ότι με τη χρήση του history server, καθίσταται δυνατή η περιήγηση και η ανακατασκευή του UI περιβάλλοντος προηγούμενων υποβληθέντων εφαρμογών.

```
##### Master conf/spark-defaults.conf #####

spark.master                     spark://192.168.1.203:7077
spark.eventLog.enabled           true
spark.eventLog.dir               /home/xilinx/spark-2.1.1/work
spark.driver.memory              505m
spark.history.fs.logDirectory    /home/xilinx/spark-2.1.1/work
```

Listing 4.10: The spark-defaults.conf file of the Master Node

- **log4j.properties:** Αυτό το βήμα είναι προαιρετικό και αποσκοπεί στην απόκρυψη οποιωνδήποτε περιττών ενημερωτικών logs κατά την υποβολή μίας εφαρμογής. Τα περιεχόμενα του αρχείου log4.properties παρουσιάζονται στη συνέχεια.

```
################SPARK log4j ocnfiguration ####################

# Set everything to be logged to the console, file
log4j.rootCategory=INFO, console, file
log4j.appender.console.threshold=ERROR
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
```

```
9  log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:
       ss} %p %c{1}: %m%n
10
11 log4j.appender.file=org.apache.log4j.RollingFileAppender
12 log4j.appender.file.File=/home/xilinx/spark-2.1.1/logs/logging.log
13 log4j.appender.file.MaxFileSize=5MB
14 log4j.appender.file.MaxBackupIndex=10
15 log4j.appender.file.layout=org.apache.log4j.PatternLayout
16 log4j.appender.file.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss}
       %p %c{1}: %m%n
17
18 # Set the default spark-shell log level to WARN. When running the
       spark-shell, the
19 # log level for this class is used to overwrite the root logger's
       log level, so that
20 # the user can have different defaults for the shell and regular
       Spark apps.
21 log4j.logger.org.apache.spark.repl.Main=WARN
22
23 # Settings to quiet third party logs that are too verbose
24 log4j.logger.org.spark_project.jetty=WARN
25 log4j.logger.org.spark_project.jetty.util.component.
       AbstractLifeCycle=ERROR
26 log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=INFO
27 log4j.logger.org.apache.spark.repl.
       SparkILoop$SparkILoopInterpreter=INFO
28 log4j.logger.org.apache.parquet=ERROR
29 log4j.logger.parquet=ERROR
30
31 # SPARK-9183: Settings to avoid annoying messages when looking up
       nonexistent UDFs in SparkSQL with Hive support
32 log4j.logger.org.apache.hadoop.hive.metastore.RetryingHMSHandler=
       FATAL
33 log4j.logger.org.apache.hadoop.hive.ql.exec.FunctionRegistry=ERROR
```

Listing 4.11: The log4.properties file of the Master Node

Ρυθμίσεις στους Spark Worker Κόμβους:

Όσον αφορά τους worker κόμβους, έπρεπε μόνο να διαμορφώσουμε κατάλληλα τα αρχεία *spark-env.sh* και *spark-defaults.conf*. Ακολούθως παραθέτουμε τα περιεχόμενα των αντίστοιχων αρχείων, επισημαίνοντας τις διαφορές με τα ίδια αρχεία του master κόμβου.

- **spark-env.sh:** Συγκριτικά με το αντίστοιχο αρχείο του master κόμβου, σε αυτή την περίπτωση έχουμε συμπεριλάβει όλες τις απαραίτητες μεταβλητές περιβάλλοντος που σχετίζονται με τις ρυθμίσεις των workers και των executors. Πιο συγκεκριμένα, ορίσαμε τα worker instances κάθε PYNQ-Z1 κόμβου σε ένα (1) και τη διαθέσιμη μνήμη τους στα 505MB. Επιπλέον, ορίσαμε ότι κάθε worker κόμβος θα μπορεί να φιλοξενεί ένα μόνο executor instance, που θα έχει επίσης 505MB διαθέσιμης μνήμης RAM.

  Κάθε executor έχει διαθέσιμο μόνο ένα CPU core. Στην πραγματικότητα, και οι δύο επεξεργαστές του Zynq APSoC θα μπορούσαν να χρησιμοποιηθούν σε κάθε worker. Ωστόσο, κάτι τέτοιο θα προκαλούσε προβλήματα στην μετέπειτα επιτάχυνση των εφαρμογών του Spark μέσω επιταχυντών υλικού. Δεδομένου ότι τα tasks των Spark executors τρέχουν παράλληλα, η απόδοση και των δύο επεξεργαστικών πυρήνων στο Spark, θα προκαλούσε μια κατάσταση ανταγωνισμού μεταξύ των δύο πυρήνων για ποιος θα είναι αυτός που θα έχει πρόσβαση στην επαναπρογραμματιζόμενη λογική. Φυσικά, το πρόβλημα αυτό συνδέεται στενά με τα ειδικά χαρακτηριστικά του επιταχυντή υλικού που δημιουργήσαμε στα πλαίσια αυτής της εργασίας. Επομένως, αυτό δεν σημαίνει απαραίτητα ότι η επαναπρογραμματιζόμενη λογική δεν μπορεί να προσπελαστεί και να επικοινωνήσει και με τους δύο επεξεργαστές ταυτόχρονα σε μια δεδομένη στιγμή, αλλά παραμένει ένα χαρακτηριστικό που απαιτεί περισσότερη έρευνα για να ενσωματωθεί στις εφαρμογές Spark.

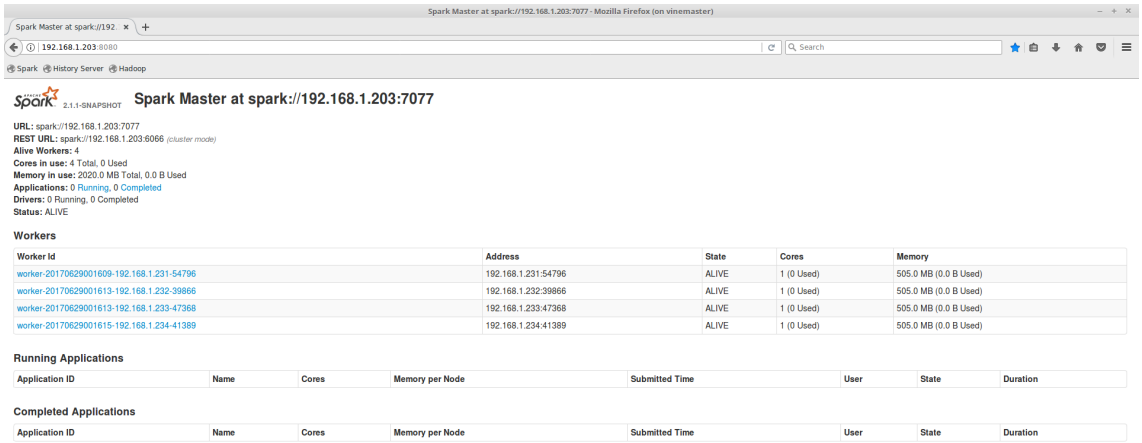
```
1 ##### Worker - conf/spark-env.sh #####
2
3 export SPARK_DRIVER_MEMORY=505m
4 export SPARK_MASTER_HOST=192.168.1.203
5 export SPARK_LOCAL_IP=192.168.1.203
6
```

```
7  ##### Worker and Executor configurations ####
8
9  export SPARK_EXECUTOR_INSTANCES=1
10 export SPARK_EXECUTOR_CORES=1
11 export SPARK_EXECUTOR_MEMORY=505m
12 export SPARK_WORKER_CORES=1
13 export SPARK_WORKER_INSTANCES=1
```

Listing 4.12: The spark-env.sh file of the Worker Nodes

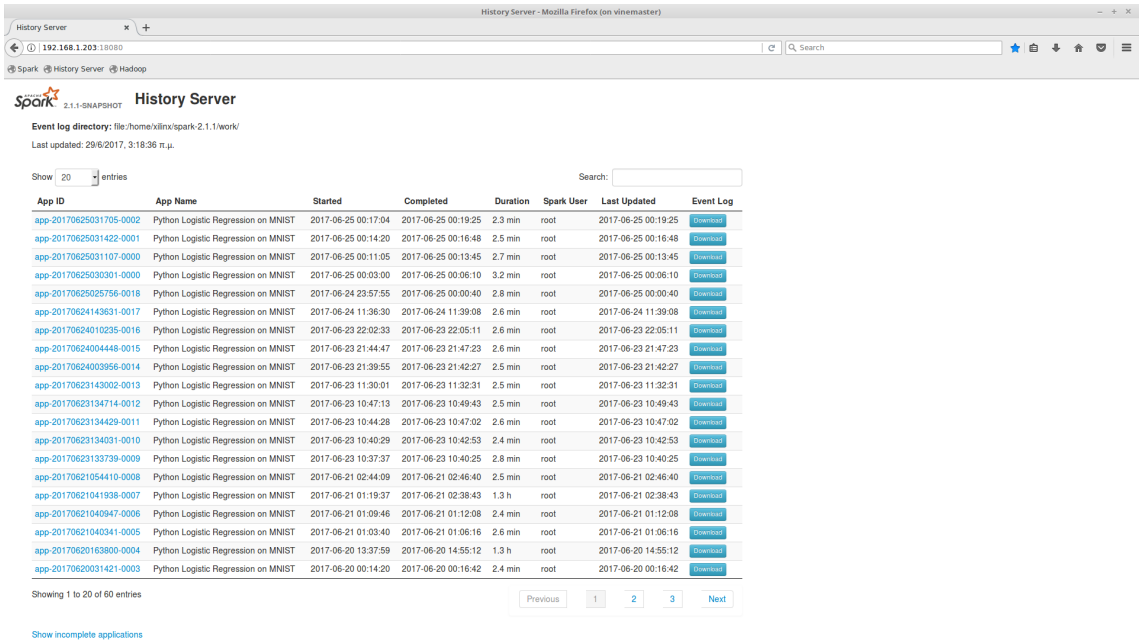
- **spark-defaults.conf:** Σε αυτό το αρχείο δεν προστέθηκαν επιπλέον ρυθμίσεις σε σύγκριση με αυτό του master κόμβου. Στην πραγματικότητα, σε αυτό το αρχείο δεν χρειάζεται να οριστεί καμία μεταβλητή, δεδομένων ότι οι εφαρμογές μας θα υποβάλλονται από τον master κόμβο και ότι το αρχείο *spark-defaults.conf* αφορά ρυθμίσεις για την εκτέλεση των εφαρμογών που υποβάλλονται.

Σε αυτό το σημείο έχουμε ολοκληρώσει τις ρυθμίσεις του Spark για το Pynq cluster. Το σχήμα 4.3, δείχνει το web UI του Spark Cluster που εκτελείται καθώς και τους PYNQ workers μαζί με τους διαθέσιμους πόρους τους, ενώ η εικόνα 4.4 παρουσιάζει τη διεπαφή χρήστη του Spark History Server.



Σχήμα 4.3: PYNQ Cluster - Spark Web UI

Πριν προχωρήσουμε στην ενσωμάτωση των επιταχυντών υλικού στις εφαρμογές του Spark, είναι σημαντικό να σημειωθεί ότι το Hadoop framework εγκαταστάθηκε στον



Σχήμα 4.4: Spark History Server Web UI

master κόμβο του συμπλέγματος. Σκοπός της εγκατάστασης, ήταν να δημιουργηθεί ένα κατανεμημένο σύστημα αρχείων του Hadoop. Πριν τη δημιουργία του (του HDFS), έπρεπε να τοποθετούμε όλα τα αρχεία εισόδου μίας εφαρμογής του Spark, στην ίδια διαδρομή (path) και στον ίδιο φάκελο με εκείνο του master (επειδή οι εφαρμογές υποβάλλονται από τον master κόμβο) σε κάθε worker κόμβο. Δεδομένου του HDFS, το μόνο απαραίτητο βήμα είναι η τοποθέτηση των αρχείων εισόδου στο κατανεμημένο σύστημα αρχείων και στη συνέχεια ο ορισμός, στο περιεχόμενο της εφαρμογής, της ανάγνωσης οποιωνδήποτε δεδομένων από αυτό. Ακολούθως θα εξετάσουμε αναλυτικότερα όλες τις απαραίτητες διαμορφώσεις για τη ρύθμιση Hadoop σε αυτήν την ψευδο-κατανεμημένη λειτουργία. Τα βήματα που ακολουθήσαμε βασίζονται στην τεκμηρίωση του Hadoop.

4.3.4 Ρυθμίσεις στο Hadoop

Αρχικά, έπρεπε να οριστεί η μεταβλητή *HADOOP_HOME* στο αρχείο */etc/environment* και στη συνέχεια να προστεθούν στη μεταβλητή *PATH* οι *bin* και textitsbin υπο-φάκελοι αυτού. Στη συνέχεια, προστέθηκαν οι ακόλουθες γραμμές κώδικα στο αρχείο *etc/hadoop/core-site.xml* που βρίσκεται στον κατάλογο του

Hadoop.

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://192.168.1.203:9000</value>
  </property>
</configuration>
```

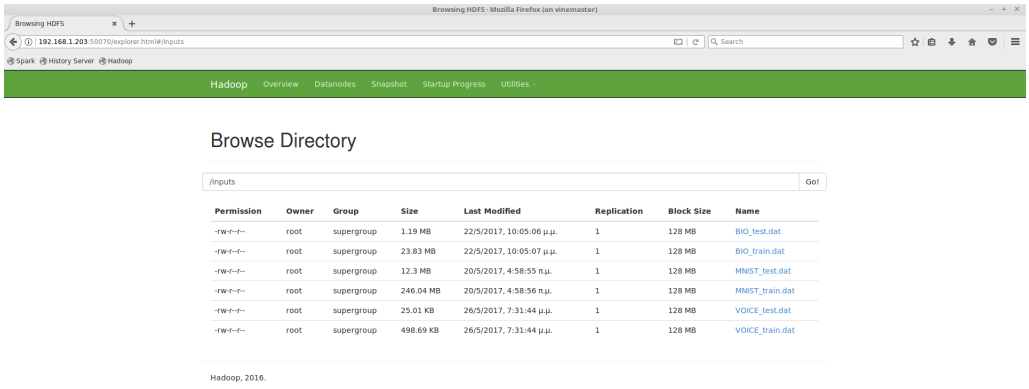
Listing 4.13: Hadoop core-site.xml configuration

Ακόμη, ορίσαμε ρητά τη μεταβλητή *JAVA_HOME* προσθέτοντάς την στο αρχείο *etc/hadoop/hadoop-env.sh* και τέλος ορίσαμε τη διαμόρφωση του κατανεμημένου συστήματος αρχείων μας στο αρχείο *Hadoop/hadoop-site.xml* αρχείο, ως εξής:

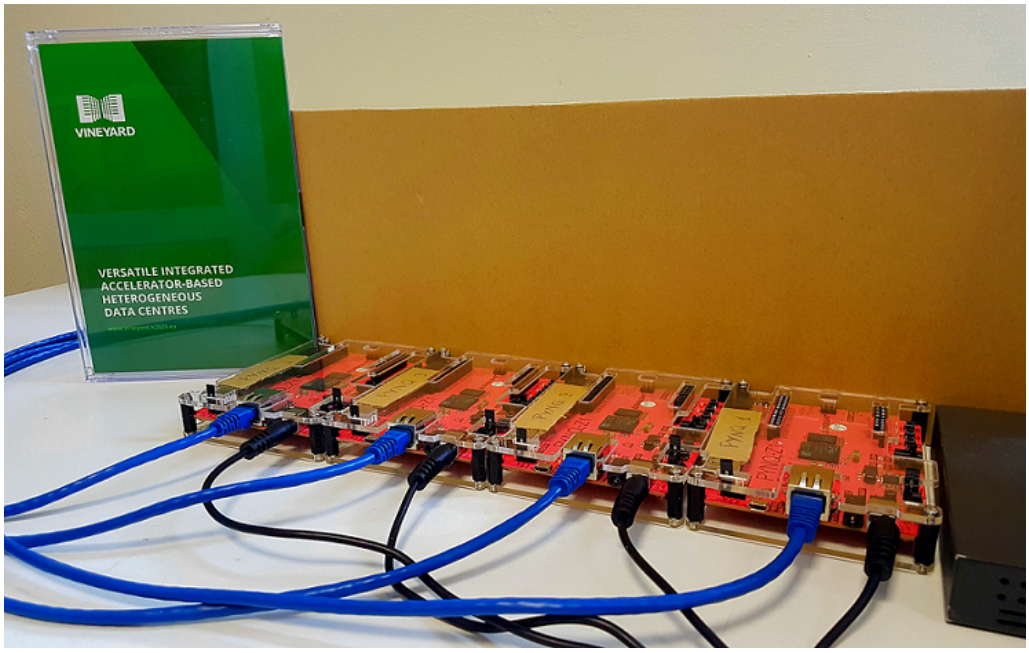
```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>/home/xilinx/hadoop-2.7.3/hdfs/namenode</value>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>/home/xilinx/hadoop-2.7.3/hdfs/datanode</value>
  </property>
</configuration>
```

Listing 4.14: Hadoop hadoop-site.xml configuration

Το σχήμα 4.5 απεικονίζει το web UI του Hadoop μαζί με τα αρχεία που έχουμε εναποθέσει στο HDFS.



Σχήμα 4.5: Hadoop Web UI



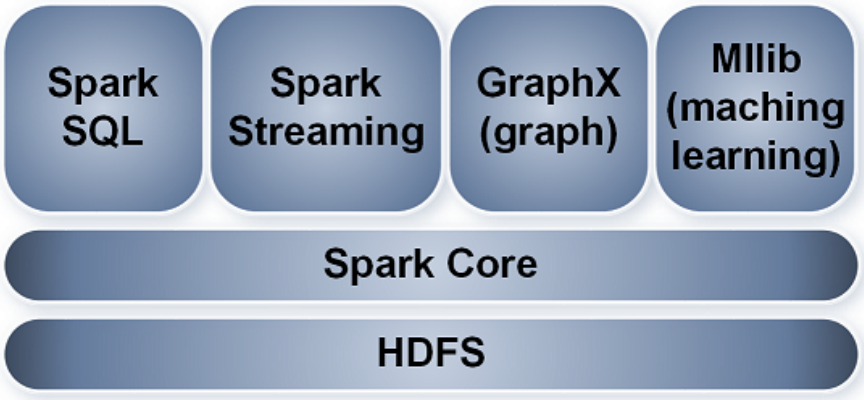
Σχήμα 4.6: Φωτογραφία του πραγματικού PYNQ-Z1 cluster

4.4 Επιτάχυνση Εφαρμογών στο Spark

Σε αυτή την ενότητα θα περιγράψουμε το προτεινόμενο framwework για την απρόσκοπτη χρήση επιταχυντών υλικού από τις εφαρμογές του Spark σε ετερογενή MPSoCs βασισμένα σε FPGA, καθώς και για την ανάπτυξη ενός συνόλου βιβλιοθηκών που θα αποκρύπτουν τις λεπτομέρειες των επιταχυντών για την απλοποιημένη ενσωμάτωση αυτών στο Spark. Επιπλέον, θα ενσωματώσουμε τις νέες βιβλιοθήκες στο SPynq cluster και τέλος θα αξιολογήσουμε τα κέρδη της χρήσης επιταχυντών υλικού, για ένα σενάριο χρήσης του αλγορίθμου λογιστικής παλινδρόμησης.

4.4.1 To Spark on Pynq (SPynq) Framework

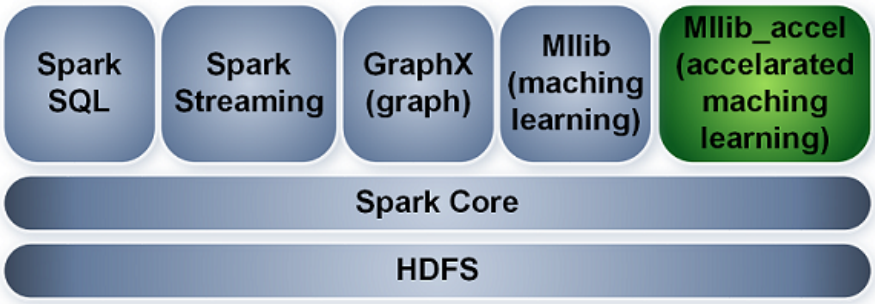
Εφόσον έχουμε εγκαταστήσει το Spark στο PYNQ-Z1 cluster, είμαστε πλέον έτοιμοι να προχωρήσουμε στα απαραίτητα βήματα για την επίτευξη επικοινωνίας με τους επιταχυντές υλικού της προγραμματιζόμενης λογικής (PL) του Zynq.
Το σχήμα 4.7 απεικονίζει τη στοίβα λογισμικού του σχήματος που υλοποιήσαμε. Το Hadoop DFS βρίσκεται στο κατώτερο επίπεδο, ενώ πάνω από αυτό είναι «χτισμένο» το Apache Spark, μαζί με τα APIs που προσφέρει για μηχανική εκμάθηση, επεξεργασία γράφων κ.α.



Σχήμα 4.7: Η στοίβα λογισμικού της υλοποιημένης διάταξης

Στην τυπική περίπτωση εκτέλεσης μιας εφαρμογής μηχανικής εκμάθησης, η εφαρμογή καλεί τη βιβλιοθήκη MLlib του Spark, η οποία χρησιμοποιεί τη βιβλιοθήκη Breeze. Η βιβλιοθήκη Breeze επικαλείται το Java Netlib framework, το οποίο αποτελεί έναν wrapper για τα χαμηλότερου επιπέδου εργαλεία γραμμικής άλγεβρας που είναι υλοποιημένα σε C ή σε Fortran. Το Netlib Java εκτελείται μέσω του Java Virtual Machine (JVM) και τα εργαλεία γραμμικής άλγεβρας (BLAS - Basic Linear Algebra Subprograms) εκτελούνται μέσω του Java Native Interface (JNI). Όλα αυτά τα επίπεδα προσθέτουν σημαντική επιβάρυνση στις εφαρμογές Spark. Συνεπώς, η βασική ιδέα του SPynq framework είναι η δημιουργία νέων πακέτων που προσφέρουν επιτάχυνση υλικού στις εφαρμογές του Spark. Με αυτόν τον τρόπο, η μόνη τροποποίηση που απαιτείται για οποιαδήποτε εφαρμογή του Spark, είναι η αντικατάσταση των «παλιών» συναρτήσεων της βιβλιοθήκης MLlib με τις νέες που επικαλούνται τον επιταχυντή υλικού. Το σχήμα 4.8 απεικονίζει το προτεινόμενο σχήμα για την επιτάχυνση εφαρμογών του Spark, όπου έχουμε υλοποιήσει μία νέα βιβλιοθήκη, την

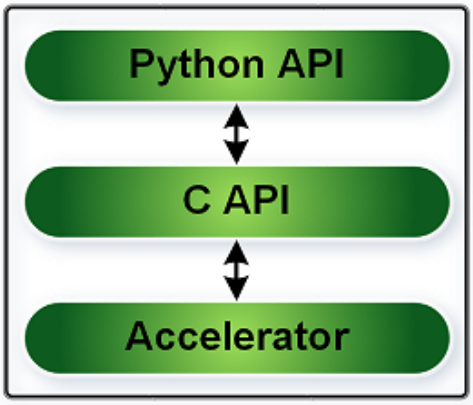
MLlib_accel, για την επιτάχυνση των αλγορίθμων μηχανικής εκμάθησης.



Σχήμα 4.8: Η στοίβα λογισμικού της προτεινόμενης διάταξης για την επιτάχυνση εφαρμογών στο Spark

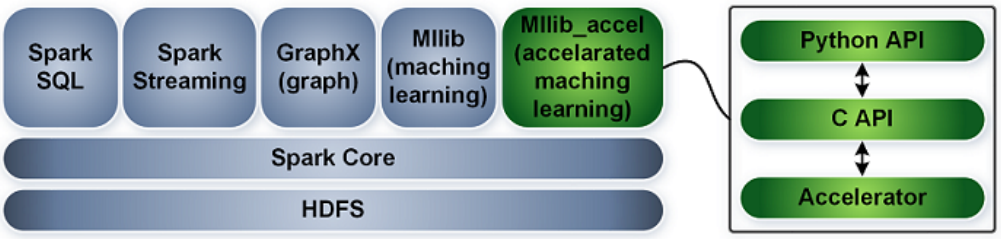
Όσον αφορά το PYNQ τώρα, όπως έχει ήδη αναφερθεί στην ενότητα 4.2, το PYNQ project διαθέτει ένα σύνολο βιβλιοθηκών, γραμμένων σε Python, για την επικοινωνία με την προγραμματιζόμενη λογική. Οι βιβλιοθήκες αυτές περιλαμβάνουν μεθόδους για τη δημιουργία και τον προγραμματισμό επιταχυντών υλικού, καθώς και ολόκληρες δομές και συναρτήσεις για τον χειρισμού των αντίστοιχων δομών και συστατικών στοιχείων κάθε επιταχυντή. Για παράδειγμα, Python βιβλιοθήκες παρέχονται για τη δημιουργία και την καταστροφή αντικειμένων DMA (Direct Memory Access), καθώς και περαιτέρω μέθοδοι για τη δέσμευση συνεχόμενων buffers μνήμης που χρησιμεύουν ως buffers των δεδομένων εισόδου και εξόδου ενός επιταχυντή υλικού. Στην πραγματικότητα, το Python API χρησιμοποιεί ένα C API, το οποίο και τελικά επικοινωνεί με τον επιταχυντή υλικού. Με άλλα λόγια, το PYNQ παρέχει έναν εύκολο και αποδοτικό τρόπο για τον χειρισμού επιταχυντών υλικού που βασίζονται σε FGPA, χωρίς να απαιτεί από τον χρήστη την ανάλογη τεχνογνωσία και εμπειρία στον συγκεκριμένο τομέα. Έτσι, για κάθε νέο επιταχυντή υλικού, αρκεί η δημιουργία μιας νέας Python βιβλιοθήκης, η οποία θα φιλοξενεί τις χαμηλότερου επιπέδου κλήσεις συστήματος για την επικοινωνία με την επαναπρογραμματιζόμενη λογική. Σημειώνεται ότι η βιβλιοθήκη αυτή είναι ανεξάρτητη από οποιοδήποτε framework (π.χ. Apache Spark, Hadoop κ.λπ.), και επομένως θα μπορούσε να ενσωματωθεί σε ένα ευρύ πλήθος εφαρμογών. Το σχήμα 4.9 παρουσιάζει όλα τα ενδιάμεσα στάδια για την επικοινωνία με τον επιταχυντή υλικού.

Εφόσον έχει δημιουργηθεί το Python API για την επικοινωνία με τον επιταχυντή, είναι δυνατή η υλοποίηση της αντίστοιχης βιβλιοθήκης που θα επικαλείται το API αυτό



Σχήμα 4.9: Διάγραμμα ροής των ενδιάμεσων σταδίων για την επικοινωνία με τον επιταχυντή υλικού

και θα επιταχύνει την εκτέλεση των εφαρμογών στο Spark. Ολόκληρο το σχήμα της προτεινόμενης στοίβας παρουσιάζεται ακολούθως.



Σχήμα 4.10: Η τελική στοίβα λογισμικού με τις νέες βιβλιοθήκες

Επιπλέον Ρυθμίσεις

Για να αξιολογήσουμε το προτεινόμενο πλαίσιο, θα πρέπει πρώτα να γίνουν κάποιες επιπλέον ρυθμίσεις. Αυτές οι ρυθμίσεις αφορούν τόσο το Spark framework όσο και τις υπάρχουσες βιβλιοθήκες του PYNQ.

Όσον αφορά το Spark, δεδομένου ότι οι νέες βιβλιοθήκες που επικαλούνται τους επιταχυντές υλικού είναι γραμμένες στη γλώσσα προγραμματισμού Python, το PySpark πρόκειται να χρησιμοποιηθεί για οποιαδήποτε υποβολή εφαρμογής. Από προεπιλογή, PySpark χρησιμοποιεί την Python 2. Ωστόσο, οι βιβλιοθήκες που του PYNQ είναι γραμμένες στην Python 3. Για το σκοπό αυτό, πρέπει να τροποποιήσουμε τις ρυθμίσεις στο Spark ώστε το PySpark να χρησιμοποιεί Python 3. Αυτό γίνεται προσθέτοντας στο αρχείο spark-env.sh κάθε κόμβου την ακόλουθες γραμμές κώδικα:

```
1 export PYSPARK_PYTHON=python3
```

```
export PYTHONHASHSEED=0

# SPARK_WORKER_TYPE = 0 for CPU only, SPARK_WORKER_TYPE = 1 for CPU + FPGA
export SPARK_WORKER_TPYE={0,1}
```

Listing 4.15: Configuring PySpark to use Python 3

Έπειτα, έπρεπε να βεβαιωθούμε ότι κάθε κόμβος του συμπλέγματος είχε την ίδια δευτερεύουσα έκδοση της Python 3 (π.χ. Python 3.**4**), η οποία είναι απαραίτητη για να λειτουργεί το Spark στο cluster mode.

Επιπλέον, ορίσαμε τη μεταβλητή PYTHONHASHSEED. Αυτή η μεταβλητή αντιπροσωπεύει την λειτουργία τυχαίου κατακερματισμό της Python και στην περίπτωση που έχει οριστεί σε μια ακέραια τιμή, αυτή η τιμή χρησιμοποιείται ως μία σταθερή φύτρα για τη δημιουργία του hash(). Στη συγκεκριμένη περίπτωση, η τιμή του έπρεπε να οριστεί στο μηδέν για να λειτουργήσει σωστά το cluster.

Ακολούθως, θέσαμε μια νέα μεταβλητή, την SPARK_WORKER_TYPE. Αυτή η μεταβλητή χρησιμοποιείται για τον καθορισμό των πόρων υλικού ενός συστήματος. Για πλατφόρμες που φιλοξενούν προγραμματιζόμενη λογική (όπως το PYNQ-Z1) θα η τιμή της μεταβλητής θα ορίζεται σε ”1”, ενώ για πλατφόρμες που δεν διαθέτουν, η τιμή του θα ορίζεται στο μηδέν. Με αυτό τον τρόπο, εκκινώντας ένα ετερογενές, σε επίπεδο worker κόμβων Spark cluster, μπορούμε να επικαλούμαστε τους επιταχυντές υλικού στους κόμβους που τους διαθέτουν (πχ. τους κόμβους PYNQ-Z1) ενώ αντίθετα να χρησιμοποιήσουμε τις προεπιλεγμένες βιβλιοθήκες του Spark για τους κόμβους που δεν διαθέτουν επιταχυντές.

Ακόμη, όλες οι νέες βιβλιοθήκες που υλοποιήθηκαν για την επιτάχυνση των εφαρμογών του Spark, έπρεπε να μεταφερθούν στον κατάλογο */python/ pyspark* ο οποίος περιλαμβάνεται ήδη στη μεταβλητή *PYTHONPATH*. Ωστόσο, το Spark διαθέτει επίσης ένα αρχείο pyspark.zip κάτω από τον φάκελο */python/lib* που χρησιμοποιείται όταν καλείται κάποια βιβλιοθήκη από το πακέτο PySPark. Έτσι, για να είναι ορατές και να μπορούν να χρησιμοποιηθούν οι νέες βιβλιοθήκες, το αρχείο *pyspark.zip* έπρεπε να μετονομαστεί ή να διαγραφεί.

Επιπλέον, ήταν αναγκαίο να μεταφέρουμε στον master κόμβο τις βιβλιοθήκες του PYNQ για την επικοινωνία με την επαναπρογραμματιζόμενη λογική. Κατά την υποβολή μιας εφαρμογής στο PySpark, όλο το περιβάλλον της εφαρμογής κατασκευά-

ζεται στον κόμβο από τον οποίο υποβλήθηκε και έπειτα το κομμάτι του κώδικα της εφαρμογής που περιέχει μετασχηματισμούς ή ενέργειες σε RDDs εκτελείται στους worker κόμβους. Είναι πλέον κατανοητό ότι αυτός είναι και ο λόγος για τον οποίο οι βιβλιοθήκες του PYNQ έπρεπε να βρίσκονται και στον master κόμβο. Όπως όμως έχουμε αναφέρει, το Python API χρησιμοποιεί ένα C API για την επικοινωνία με την επαναπρογραμματιζόμενη λογική. Το PYNQ, παρέχει αυτό το API συμπεριλαμβάνοντας κάποιες βιβλιοθήκες σε μορφή shared object file (.so) (πχ. libdma.so). Το πρόβλημα εντοπίζεται στο ότι οι βιβλιοθήκες αυτές είναι μεταγλωττισμένες για την αρχιτεκτονική 32-bit της ARM. Δεδομένου ότι οι εφαρμογές στο Spark θα υποβάλλονται από τον master κόμβο, μεταγλωττίσαμε εκ νέου τα αρχεία *libdma.so* και *libsds_lib.so* που χρησιμοποιούνται από τη βιβλιοθήκη *dma.py* του PYNQ, για τις αρχιτεκτονικές x86 και x86_64 της Intel. Τέλος, μετά τη δημιουργία των νέων μεταγλωττισμένων βιβλιοθηκών (μία για κάθε διαφορετική αρχιτεκτονική συστήματος), τροποποιήσαμε τον κώδικα του αρχείου dma.py ώστε να επιλέγεται το σωστό *.so* αρχείο ανάλογα με το σύστημα στο οποίο γίνεται η εκτέλεση. Η αλλαγή στο αρχείο *dma.py* παρουσιάζεται ακολούθως.

```
LIB_SEARCH_PATH = os.path.dirname(os.path.realpath(__file__))

if((platform.machine()=='x86_64')):
  # load 64bit ELF
  libdma = ffi.dlopen(LIB_SEARCH_PATH + "/libdma64.so")
  libxlnk = memapi.dlopen(LIB_SEARCH_PATH + "/libsds_lib64.so")
elif (platform.machine()=='armv7l'):
  #load 32bit ELF compiled for ARM
  libdma = ffi.dlopen(LIB_SEARCH_PATH + "/libdma.so")
  libxlnk = memapi.dlopen(LIB_SEARCH_PATH + "/libsds_lib.so")
elif (platform.machine()=="i686"):
  libdma = ffi.dlopen(LIB_SEARCH_PATH + "/libdma32.so")
  libxlnk = memapi.dlopen(LIB_SEARCH_PATH + "/libsds_lib32.so")
else:
  print("Machine type not supported. Exiting!\n")
exit(1)
```

Listing 4.16: Modification in dma.py (1) - Support for Intel x86 and x86_64 architectures

Ο πίνακας 4.3, εμφανίζει όλα τα διαθέσιμα .so αρχεία μετά τη διαδικασία μεταγλώττισης που ακολουθήσαμε, καθώς και την αρχιτεκτονική για την οποία δημιουργήθηκαν. Το αρχείο libsds_lib.so περιέχει το C API για τη διαχείριση των αιτημάτων μνήμης (πχ. δέσμευση ή απελευθέρωση συνεχούς μέρους μνήμης) ενώ το αρχείο libdma.so περιέχει το C API για την επικοινωνία με οποιονδήποτε DMA ενός επιταχυντή υλικού.

.so File Name	**System Architecture**
libdma.so	ARM 32-bit
libdma32.so	Intel x86
libdma64.so	Intel x86_64
libsds_lib.so	ARM 32-bit
libsds_lib32.so	Intel x86
libsds_lib64.so	Intel x86_64

Πίνακας 4.3: Available .so files after the recompilation process

Επιπλέον, αντιμετωπίσαμε ακόμη μία πρόκληση σχετικά με το αρχείο dma.py. Με τη δημιουργία ενός αντικειμένου DMA, ένας buffer μπορεί ακολούθως να δημιουργηθεί για την αποθήκευση των δεδομένων που θα μεταφέρει. Η δημιουργία ενός buffer γίνεται μέσω της μεθόδου get_buf(), η οποία επιστρέφει έναν δείκτη *CFFI* σε δεδομένα τύπου *integer* ή *long long*. Ωστόσο, οι περισσότερες εφαρμογές στο Spark, χρησιμοποιούν αριθμούς κινητής υποδιαστολής. Έτσι, τροποποιήσαμε τη μέθοδο *get_buf()* ώστε να δύναται να επιστρέφει έναν CFFI δείκτη σε δεδομένα κινητής υποδιαστολής. Η τελική μορφή της μεθόδου *get_buf()* παρουσιάζεται ακολούθως.

```
def get_buf(self, width=32, data_type = 'int'):
  """Get a CFFI pointer to object's internal buffer.

  This can be accessed like a regular array in python. The width can be
    either 32 or 64.

  Parameters
  ----------
  width : int
  The data width in the buffer.
  data_type : string
  The type of the returned pointer

```

```
13  Returns
14  _______
15  cffi.FFI.CData
16  An CFFI object which can be accessed similar to arrays in C.
17
18  """
19  if self.buf is not None:
20    if width == 32:
21      if data_type == 'int':
22      return ffi.cast("int *",self.buf)
23    elif data_type == 'float':
24      return ffi.cast("float *",self.buf)
25    else:
26      raise RuntimeError ("Not supported type")
27  elif width == 64:
28    return ffi.cast("long long *",self.buf)
29  else:
30    raise RuntimeError("Buffer not created.")
```

Listing 4.17: Modification in dma.py (2) - Add functionality to retun a pointer of float data type

Τέλος, το Spark χρησιμοποιεί σειριοποιητές (serializers) για την αποστολή ή λήψη δεδομένων από του worker κόμβους. Σειριοποίηση ενός αντικειμένου ορίζεται η μετατροπή της κατάστασής του σε μια ροή από bytes έτσι ώστε η ροή αυτή να μπορεί να επανέλθει, ως ένα αντίγραφο, στην μορφή του αρχικού αντικειμένου [41]. Από προεπιλογή, το Spark χρησιμοποιεί τον Pickle σειριοποιητή για το PySpark, ενώ υπάρχουν και μερικοί άλλοι διαθέσιμοι (πχ. Marshall, και Cloudpickle). Φυσικά, ένας χρήστης μπορεί να γράψει τον δικό του σειριοποιητή και να τον μεταφέρει στο Spark. Η πρόκληση εδώ, σχετίζεται και πάλι με τη βιβλιοθήκη DMA. Τα αντικείμενα DMA που δημιουργούνται από τη βιβλιοθήκη του PYNQ έχουν πολύπλοκη δομή που το Spark δεν μπορεί σειριοποιήσει ή να απο-σειριοποιήσει. Συγκεκριμένα, το Spark δεν μπορεί να σειριοποιήσει ή να απο-σειριοποιήσει τα αντικείμενα CFFI που χρησιμοποιούνται για τους buffers των DMAs. Σε αυτό το σημείο, έπρεπε να επιλέξουμε μεταξύ δύο πιθανών λύσεων: τη δημιουργία ενός νέου serializer ή την εύρεση ενός νέου σχήματος δημιουργίας των buffers ανεξάρτητα από τα DMA αντικείμενα.

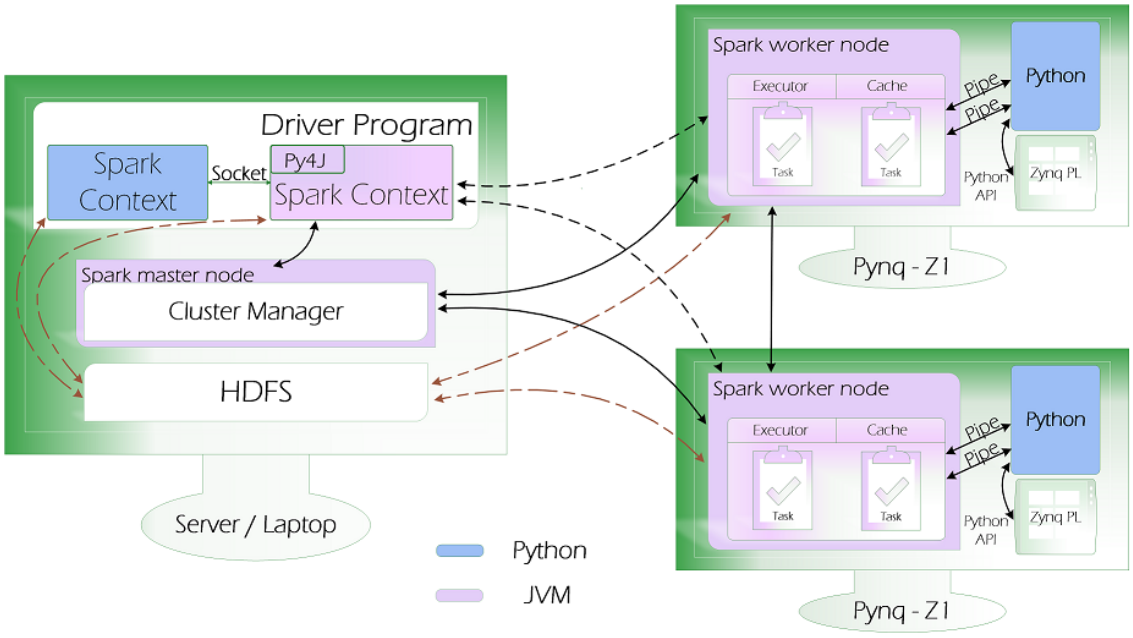
Η πρώτη λύση είχε τα περισσότερα μειονεκτήματα. Εκτός από το ότι θα έπρεπε να

σπαταλήσουμε περισσότερο χρόνο για τη δημιουργία ενός νέου serializer, ένας χρήστης θα έπρεπε πάντα να ορίζει ρητά τον σειριοποιητή μας σε κάθε νέα εφαρμογή του Spark. Εκτός αυτού, η διαδικασία σειριοποίησης και απο-σειριοποίησης των αντικειμένων DMA μπορεί να είναι χρονοβόρα, ενώ τα σειριοποιημένα δεδομένα θα έπρεπε να μεταφέρονται μέσω δικτύου στον master κόμβο. Είναι σαφές, ότι μία τέτοια λύση δεν θα βοηθούσε στο σχήμα επιτάχυνσης που θέλουμε να πετύχουμε, καθώς θα προσέθετε σημαντική επιβάρυνση στον χρόνο εκτέλεσης μιας εφαρμογής.

Αντί αυτού, η λύση της δημιουργίας των buffers ανεξάρτητα από τα DMA αντικείμενα, ήταν αρκετά υποσχόμενη. Η βασική ιδέα ήταν αρκετά απλή. Αρχικά, θα δημιουργούνται οι buffers για τα δεδομένα των αντικειμένων DMA και στη συνέχεια, με τη δημιουργία των αντικειμένων DMA οι buffers θα αντιστοιχίζονται σε αυτά. Πριν την επιστροφή οποιωνδήποτε δεδομένων στον master κόμβο, οι buffers θα αποδεσμεύονται από τα DMA αντικείμενα και αντί να επιστρέφεται ολόκληρη ή δομή του αντικειμένου, θα επιστρέφονται μόνο οι εικονικές και φυσικές διευθύνσεις καθώς και το μέγεθος των buffers σε ένα νέο RDD. Με αυτόν τον τρόπο, τα δεδομένα των buffers παραμένουν στη μνήμη των worker κόμβων, υλοποιώντας έτσι μία ανάλογη με του Spark μέθοδο για την προσωρινή παραμονή και αποθήκευση των δεδομένων ενός RDD στους workers. Ένα πιο λεπτομερές παράδειγμα θα δοθεί στην επόμενη ενότητα, όπου θα παρουσιαστεί ένα σενάριο εφαρμογής της παραπάνω διαδικασίας για την περίπτωση εκτέλεσης του αλγορίθμου λογιστικής παλινδρόμησης.

Έχοντας ολοκληρώσει τα βήματα που απαιτούνται για την επιτάχυνση μιας εφαρμογής στο Spark, μπορούμε να ρίξουμε μια ματιά στην συνολική αρχιτεκτονική του cluster μας (figure 4.11).

Όπως έχει ήδη αναφερθεί, ο Spark master στεγάζεται στον κόμβο της Intel αρχιτεκτονικής, ο οποίος φιλοξενεί επίσης το κατανεμημένο σύστημα αρχείων Hadoop. Αντίστοιχα, οι Spark workers φιλοξενούνται στις πλατφόρμες PYNQ-Z1. Δεδομένου ότι οι εφαρμογές μας είναι γραμμένες σε Python, το PySpark χρησιμοποιείται για την υποβολή τους στο cluster. Το PySpark είναι υλοποιημένο πάνω από το Java API του Spark. Με αυτόν τον τρόπο, τα δεδομένα επεξεργάζονται στο περιβάλλον της Python και γίνονται cache ή shuffle στο περιβάλλον του JVM. Με την υποβολή μιας Python εφαρμογής, στο πρόγραμμα οδήγησης δημιουργείται ένα Python SparkContext όπου το πακέτο Py4J χρησιμοποιείται για την εκκίνηση ενός JVM και

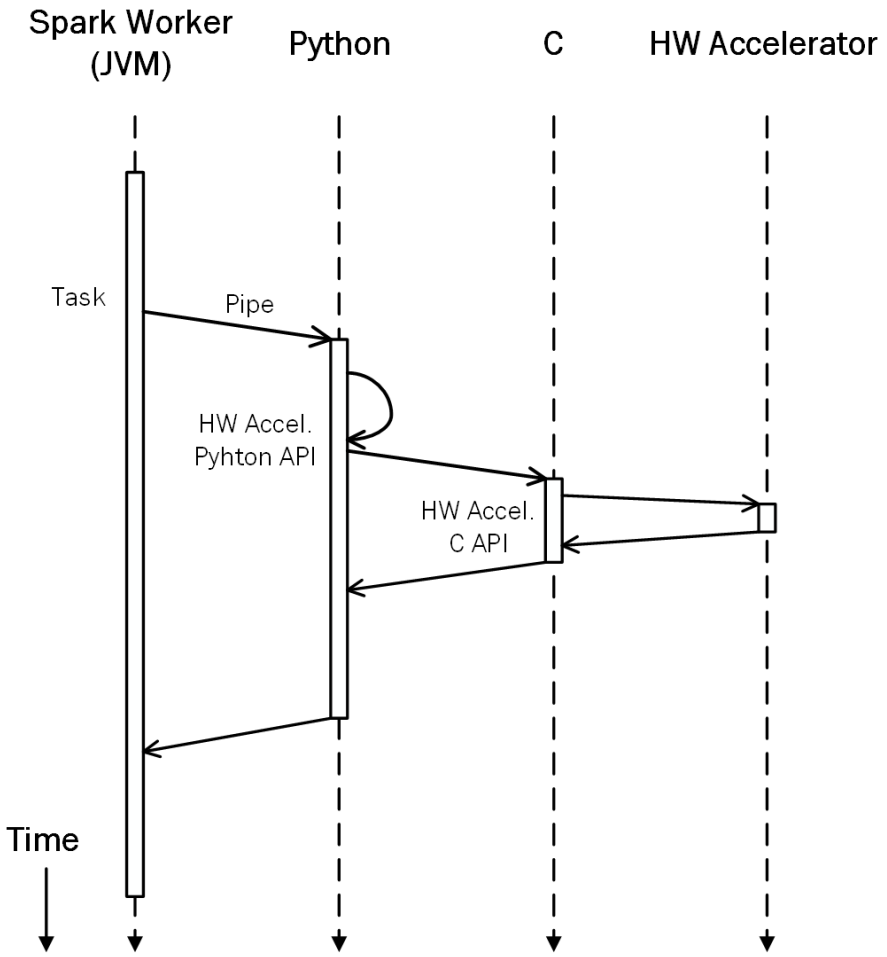


Σχήμα 4.11: Η τελική αρχιτεκτονική του υλοποιημένου cluster

τη μετέπειτα δημιουργία ενός JavaSparkContext. Το Py4J χρησιμοποιείται μόνο στο πρόγραμμα οδήγησης για την τοπική επικοινωνία μεταξύ των αντικειμένων Python και Java SparkContext.

Οι μετασχηματισμοί ενός RDD στην Python αντιστοιχίζονται σε μετασχηματισμούς αντικειμένων PythonRDD της Java. Στους worker κόμβους, τα PythonRDD αντικείμενα εκκινούν Python υπο-διεργασίες και επικοινωνούν με αυτές χρησιμοποιώντας pipes, για την αποστολή του κώδικα του χρήστη και των προς επεξεργασία δεδομένων. Στην περίπτωσή μας, όταν τα δεδομένα λαμβάνονται από τις Python υπο-διεργασίες, ο επιταχυντής υλικού επικαλείται μέσω του Python API. Το σχήμα 4.12 παρουσιάζει με περισσότερη λεπτομέρεια την επικοινωνία και τη ροή των δεδομένων για μια δεδομένη εργασία στην πλευρά των worker κόμβων.

Ο worker είναι ενεργός καθ' όλη τη διάρκεια και αναμένει να λάβει κάποιο κομμάτι εργασίας. Μόλις λάβει μια εργασία, ένας executor εκκινείται και στη συνέχεια τα δεδομένα αποστέλλονται σε Python υπο-διεργασίες. Στην περίπτωση χρήσης επιταχυντών υλικού, το Python API καλείται το οποίο στη συνέχεια χρησιμοποιεί το C API για να επικοινωνήσει με την προγραμματιζόμενη λογική. Τα δεδομένα επιστρέφονται στις Python υπο-διεργασίες μέσω της ίδιας ακριβώς οδού και αποστέλλονται πίσω στον worker.



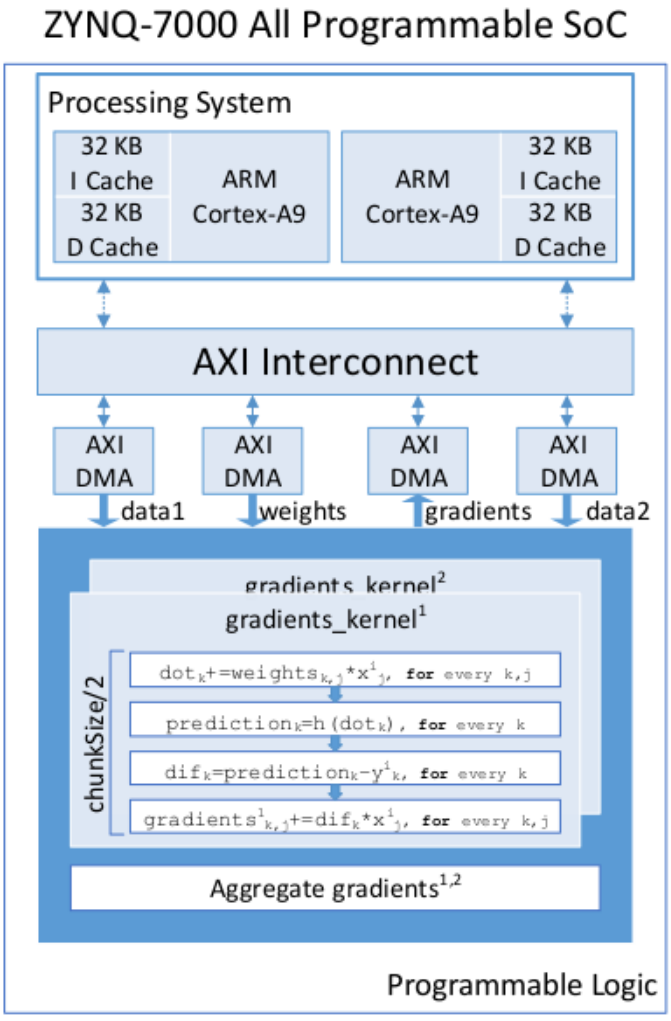
Σχήμα 4.12: Η διαδρομή των δεδομένων στην πλευρά του worker

4.4.2 Σενάριο Χρήσης του Αλγορίθμου Λογιστικής Παλινδρόμισης

Για την αξιολόγηση του προτεινόμενου framework, αναπτύχθηκε ένας επιταχυντής υλικού για την εκπαίδευση ενός μοντέλου λογιστικής παλινδρόμησης βασισμένο στον αλγόριθμο Batch Gradient Descent (BGD) και πιο συγκεκριμένα για τον υπολογισμό των κλίσεων του αλγορίθμου (gradients kernel). Ο επιταχυντής υλοποιήθηκε ως μέρος της επονομαζόμενης "*FPGA-Acceleration of Machine Learning in Cloud Computing, a case study using Logistic Regression*" διπλωματικής εργασίας [42]. Για οποιαδήποτε τεχνική ή περαιτέρω πληροφορία σχετικά με τον επιταχυντή αυτόν, μπορείτε να ανατρέξτε στην αντίστοιχη εργασία.

Το σχήμα 4.13 απεικονίζει το μπλοκ διάγραμμα του επιταχυντή για τον αλγόριθμο της λογιστικής παλινδρόμησης. Το πρόγραμμα οδήγησης χρησιμοποιείται για την

αποστολή των απαιτούμενων παραμέτρων μέσω της διασύνδεσης AXI στον επιταχυντή υλικού. Στη συγκεκριμένη εφαρμογή, χρησιμοποιούνται τέσσερα διαφορετικά κανάλια για την επικοινωνία μεταξύ των ARM επεξεργαστών και του επιταχυντή. Δύο από αυτά χρησιμοποιούνται για την αποστολή των δεδομένων και ένα τρίτο κανάλι χρησιμοποιείται για την αποστολή των βαρών του αλγορίθμου. Το τελευταίο κανάλι χρησιμοποιείται για την λήψη των αποτελεσμάτων του επιταχυντή. Ακόμη, χρησιμοποιείται και μία δομή Accelerator Adapter (η οποία δεν παρουσιάζεται στο σχήμα), προκειμένου να υπάρχει δυνατότητα αποστολής μεταβλητού μεγέθους δεδομένων στον επιταχυντή.



Σχήμα 4.13: Επιτάχυνση του αλγορίθμου λογιστικής παλινδρόμησης στο Spark χρησιμοποιώντας το FPGA του PYNQ-Z1 [42]

Τέλος, για να επιταχυνθεί ο χρόνος εκτέλεσης, η προγραμματιζόμενη λογική φιλοξενεί δύο αντίγραφα των πυρήνων που μπορούν να εκτελεστούν παράλληλα. Κάθε

πυρήνας αποτελείται από τέσσερα μπλοκ που χρησιμοποιούνται για τον υπολογισμό των gradients και στα οποία εφαρμόζεται ένα σχήμα διοχέτευσης για την αύξηση της συνολικής απόδοσης.

Στο Spark, η μέθοδος *gradients_kernel* μπορεί να παραλληλιστεί χρησιμοποιώντας το σχήμα Map-Reduce. Έτσι, οι μερικές κλίσεις του αλγορίθμου υπολογίζονται σε κάθε worker κόμβο, χρησιμοποιώντας διαφορετικά κομμάτια του αρχείου εισόδου εκπαίδευσης του μοντέλου, ενώ στη συνέχεια ο master κόμβος συγκεντρώνει, αθροίζει και ενημερώνει τα βάρη του αλγορίθμου.

Όταν ένας χρήστης Spark θέλει να χρησιμοποιήσει τον επιταχυντή υλικού, η μόνη αλλαγή στην οποία χρειάζεται να προβεί, είναι η αντικατάσταση της βιβλιοθήκης *mllib* του Spark με την *mllib_accel*. Με αυτόν τον τρόπο, ένας χρήστης μπορεί να επιταχύνει τον χρόνο εκτέλεσης μίας εφαρμογής στο Spark με μια απλή αντικατάσταση των βιβλιοθηκών.

```
from pyspark import SparkContext
from pyspark.mllib.regression import LabeledPoint
from mllib_accel.classification import LogisticRegression

...

sc = SparkContext(appName = "Python Logistic Regression")

trainRDD = sc.textFile(train_file, numPartitions).map(parsePoint)
testRDD = sc.textFile(test_file, numPartitions).map(parsePoint)

LR = LogisticRegression(numClasses, numFeatures).train(trainRDD, alpha,
    numIterations, _accel_)
LR.test(testRDD)

sc.stop()
```

Listing 4.18: Spark code for the utilization of the hardware accelerator

Πιο συγκεκριμένα, στο περιβάλλον της Python, δημιουργείται ένα LogisticRegression αντικείμενο το οποίο υποστηρίζει διάφορες μεθόδους (train, test, predict κλπ.). Κάθε απαιτούμενη ενέργεια διαβιβάζεται μέσω μίας map εντολής η οποία ακολουθείται από μια αντίστοιχη reduce ή collect ενέργεια. Για παράδειγμα, στη μέθοδο train του

LogisticRegression αντικειμένου, ή μέθοδος *gradients_kernel* εκτελείται σε όλους τους διαθέσιμους workers και στη συνέχεια, στην πλευρά των workers, η βιβλιοθήκη του συγκεκριμένου επιταχυντή καλείται για την επικοινωνία με την προγραμματιζόμενη λογική.

Μετά από profiling της εφαρμογής, καταλήξαμε στο συμπέρασμα ότι ο περισσότερος χρόνος (99,2%) δαπανάται στην εγγραφή των δεδομένων εισόδου στους προσωρινούς buffers των DMAs. Εφόσον τα δεδομένα αυτά παραμένουν ίδια καθ' όλη τη διάρκεια της εκτέλεσης του αλγορίθμου, καταφέραμε και υλοποιήσαμε το σχήμα που περιεγράφηκε ανωτέρω, όπου τα δεδομένα των buffers παραμένουν στη μνήμη καθ' όλη τη διάρκεια της κλήσης του επιταχυντή. Υλοποιήσαμε μία νέα μέθοδο όπου τα δεδομένα εισόδου εγγράφονται στους αντίστοιχους buffers και παραμένουν εκεί για την υπόλοιπη εκτέλεση της εφαρμογής. Σε κάθε επανάληψη του αλγορίθμου, δημιουργούνται αντικείμενα DMA στα οποία εκχωρούνται οι buffers που δημιουργήθηκαν προηγουμένως . Πριν από την εντολή επιστροφής σε κάθε επανάληψη, οι buffers αποδεσμεύονται από τα DMA αντικείμενα και τα τελευταία καταστρέφονται.

Με βάση τα παραπάνω, δημιουργήσαμε ένα Python API το οποίο αποτελείται από τρεις μεθόδους:

- **cma (contiguous memory allocate):** Αυτή η μέθοδος χρησιμοποιείται για τη δημιουργία των buffers και την περαιτέρω δέσμευση συνεχούς μνήμης. Επίσης σε αυτό το σημείο προγραμματίζεται η μονάδα FPGA και εγγράφονται τα δεδομένα εισόδου στους αντίστοιχους buffers. Χρησιμοποιώντας την cma, ένα νέο RDD δημιουργείται και διατηρείται, το οποίο περιέχει μόνο πληροφορίες για τους buffers (διευθύνσεις μνήμης, μεγέθη κλπ.).

- **gradients_kernel_accel**: Σε αυτή τη μέθοδο, δημιουργούνται τα αντικείμενα DMA χρησιμοποιώντας τις και μεθόδους των βιβλιοθηκών της Xilinx. Οι buffers, που έχουν ήδη δημιουργηθεί με την κλήση της cma, εκχωρούνται στα αντικείμενα DMA, ενώ τα τρέχοντα βάρη γράφονται στη μνήμη και τελικά τα δεδομένα μεταφέρονται στην προγραμματιζόμενη λογική. Τα αποτελέσματα (partial gradients) επιστρέφονται και εγγράφονται στον αντίστοιχο buffer. Τέλος, οι buffers αποδεσμεύονται από τα αντικείμενα DMA ενώ αυτά καταστρέφονται.

- **cmf (contiguous memory free):** Αυτή η μέθοδος χρησιμοποιείται για την αποδέσμευση της μνήμης που είχε προηγουμένως δεσμευτεί και την καταστροφή των αντίστοιχων buffers.

Ο πηγαίος κώδικας του Python API καθώς και της νέας βιβλιοθήκης του Spark που υλοποιήθηκαν, βρίσκονται στο παράρτημα Δ′.2. Είναι φανερό, ότι το API του επιταχυντή είναι ανεξάρτητο από το Spark και μπορεί να χρησιμοποιηθεί από οποιαδήποτε Python εφαρμογή.

5 Πειραματική Αξιολόγηση

5.1 Εκτέλεση σε Επεξεργαστές

Στο πρώτο μέρος αυτού του κεφαλαίου, θα παρουσιάσουμε τα αποτελέσματα της αξιολόγησης μας χρησιμοποιώντας τις ενσωματωμένες βιβλιοθήκες του Spark. Μετρήσαμε τον χρόνο εκτέλεσης των εφαρμογών που περιγράψαμε στην ενότητα 3.3 καθώς και την ενεργειακή κατανάλωση των ενσωματωμένων συστημάτων της ενότητας 3.2. Είναι σημαντικό να σημειωθεί, ότι οι εν λόγω εφαρμογές είναι γραμμένες σε Scala η οποία σαν γλώσσα προγραμματισμού είναι πολύ πιο γρήγορη από την Python. Επιπλέον, μετρήσαμε τον χρόνο εκτέλεσης και την ενεργειακή απόδοση ενός διακομιστή που βασίζεται σε έναν τυπικό Intel Xeon επεξεργαστή υψηλής απόδοσης και τέλος τον χρόνο εκτέλεσης και την αποδοτικότητα ενός επεξεργαστή που χρησιμοποιείται κυρίως σε φορητούς υπολογιστές, που βασίζεται στην οικογένεια των Intel i5 επεξεργαστών. Έτσι, έχουμε τη δυνατότητα να διενεργήσουμε μια καλή και ολοκληρωμένη σύγκριση μεταξύ των συστημάτων που εξετάσαμε και τελικά να συνάγουμε συμπεράσματα για κάθε περίπτωση.

Προκειμένου η σύγκριση των συστημάτων να είναι όσο το δυνατό πιο δίκαιη, εκτελέσαμε όλες τις εφαρμογές του Spark σε local mode, όπου δεσμεύτηκαν 4 επεξεργαστικοί πυρήνες ενώ η διαθέσιμη μνήμη για κάθε executor καθώς και το driver program του Spark ορίστηκε στα 800MB. Εξαίρεση αποτελεί η περίπτωση του PYNQ-Z1. Η πλατφόρμα αυτή, διαθέτει μόνο 2 πυρήνες και 512MB μνήμης RAM. Συνεπώς για το Pynq χρησιμοποιήθηκαν και οι δύο διαθέσιμοι πυρήνες του ενώ η μνήμη για την εκτέλεση των εφαρμογών του Spark ορίστηκε στα 505MB. Θα ήταν σημαντικό να αναφέρουμε σε αυτό το σημείο, ότι μπορεί από την μία πλευρά αυτή η τελευταία σύγκριση του Pynq με τα υπόλοιπα συστήματα να μην είναι τόσο δίκαιη (δεδομένου

ότι έχουμε διαθέσει αρκετά λιγότερους υπολογιστικούς πόρους στο Spark) ωστόσο, ο κύριος σκοπός της συμπερίληψης του σε αυτή την αξιολόγηση αποσκοπεί στο να διαφανούν πληρέστερα τα οφέλη από την μετέπειτα «ενσωμάτωση» επιταχυντών υλικού στις εφαρμογές του Spark (στο κεφάλαιο 4). Τονίζεται λοιπόν ότι σε αυτή την αξιολόγηση, χρησιμοποιήθηκαν μόνο οι CPU πυρήνες του PYNQ-Z1.
Επιπλέον, αναφέρεται ότι η παρούσα αξιολόγηση απόδοσης δεν μετρά τις επιδόσεις των επεξεργαστών στους κόμβους Spark αλλά αξιολογεί την απόδοση ολόκληρου του Spark framework. Τα χαρακτηριστικά του λειτουργικού συστήματος και των επεξεργαστών κάθε πλατφόρμας παρουσιάζονται στον επόμενο πίνακα (5.1).

Features	**Server**	**Laptop**	**Raspberry Pi 3**	**DragonBoard 410c**	**PYNQ-Z1**
Vendor	Intel	Intel	Broadcom	Qualcomm	Xilinx
Device	E5-2650	i5-430M	BCM2837	Snapdragon 410	Zynq XC7Z020
Cores(threads)	8(16)	2(4)	4	4	2
Processor	E5-2650	i5-430M	A53	A53	A9
Architecture	64-bit CISC	64-bit CISC	64-bit RISC	64-bit RISC	32-bit RISC
Process	22nm	32nm	40nm	28nm	28nm
Clock Frequency	2.6GHz	2.26GHz	1.2GHz	1.2GHz	667MHz
Level 1 cache	512kB	128kB	32kB	32kB	32kB
Level 2 cache	2048kB	512kB	512kB	512kB	512kB
TDP	95W	35W	NA	3.7W	NA
Operating System	CentOS	Ubuntu	Debian	Debian	Ubuntu

Πίνακας 5.1: Κύρια χαρακτηριστικά των προς αξιολόγηση πλατφορμών.

Για την αξιολόγηση, δημιουργήθηκε ένα bash script, το οποίο χρησιμοποιήθηκε για την εκτέλεση κάθε μεμονωμένης εφαρμογής. Επειδή οι εφαρμογές εκτελέστηκαν σε πραγματικά συστήματα, ο μετρούμενος χρόνος ήταν λογικό να κυμαίνεται ανάμεσα σε διαφορετικές εκτελέσεις. Για το σκοπό αυτό, τροποποιήσαμε το script ώστε να εκτελεί πέντε φορές κάθε εφαρμογή, επιστρέφοντας ως τελικό χρόνο εκτέλεσης τον μέσο όρο των χρόνων από τις 5 εκτελέσεις. Ο χρόνος εκτέλεσης κάθε εφαρμογής μετρήθηκε με τη βοήθεια της ενσωματωμένης λειτουργίας *time* των Linux και αναφέρεται στον συνολικό χρόνο εκτέλεσης, συμπεριλαμβανομένου αυτού για την εκκίνηση των συστατικών στοιχείων του Spark. Επιπλέον εκτελώντας μερικές δοκιμές, διαπιστώσαμε ότι σε ορισμένες περιπτώσεις στα ενσωματωμένα συστήματα, ο περισσότερος χρόνος δαπανιούνταν στην εκκίνηση του Spark και όχι στο υπολογιστικό κομμάτι του εκάστοτε αλγορίθμου. Για το σκοπό αυτό, τροποποιήσαμε τις προεπιλεγμένες τιμές των παραμέτρων εισόδου των εφαρμογών (π.χ. σε επαναληπτικούς αλγόριθμους αυξήσαμε τον αριθμό των επαναλήψεων κ.λπ.) προκειμένου να καταστήσουμε το αλ-

γοριθμικό μέρος κάθε εφαρμογής περισσότερο χρονοβόρο σε σύγκριση με το κομμάτι της εκκίνησης του Spark. Έτσι, τα παραγόμενα αποτελέσματα θα είναι ακριβέστερα και θα αντιπροσωπεύουν καλύτερα τον κάθε αλγόριθμο. Τα ορίσματα εισόδου που χρησιμοποιήσαμε για κάθε εφαρμογή, εμφανίζονται στους πίνακες 5.2 και 5.3.

	Linear Regression	**Logistic Regression**	**KMeans**
Application File	LinearRegressionExample.scala	LogisticRegressionExample.scala	KMeansExample.scala
maxIter	100	100000000	-
regParam	0.001	0.001	-
elasticNetParam	0.9	1.0	-
tol	1.0E-20	1.0E-20	-
Input Data File	sample_linear_regression_data.txt	sample_libsvm_data.txt	sample_kmeans_data.txt

Πίνακας 5.2: Ορίσματα εισόδου των εφαρμογών της οικογένειας ML.

	CC	**Pagerank**	**Triangles**
Application File	Analytics.scala (cc)	Analytics.scala (pagerank)	Analytics.scala (triangles)
numEPart	20	20	20
Input Data File	pagerank_data.txt	pagerank_data.txt	pagerank_data.txt

Πίνακας 5.3: Ορίσματα εισόδου των εφαρμογών της οικογένειας επεξεργασίας γράφων.

Input Argument	**Meaning**
Application File	The source file of the evaluated application
maxIter	The maximum number of iterations the algorithm will run
regParam	The regularization parameter
elasticNetParam	The ElasticNet mixing parameter
tol	The convergence tolerance of iterations
numEPart	The number of edge partitions
Input Data File	The name of the dataset file given as input to the application

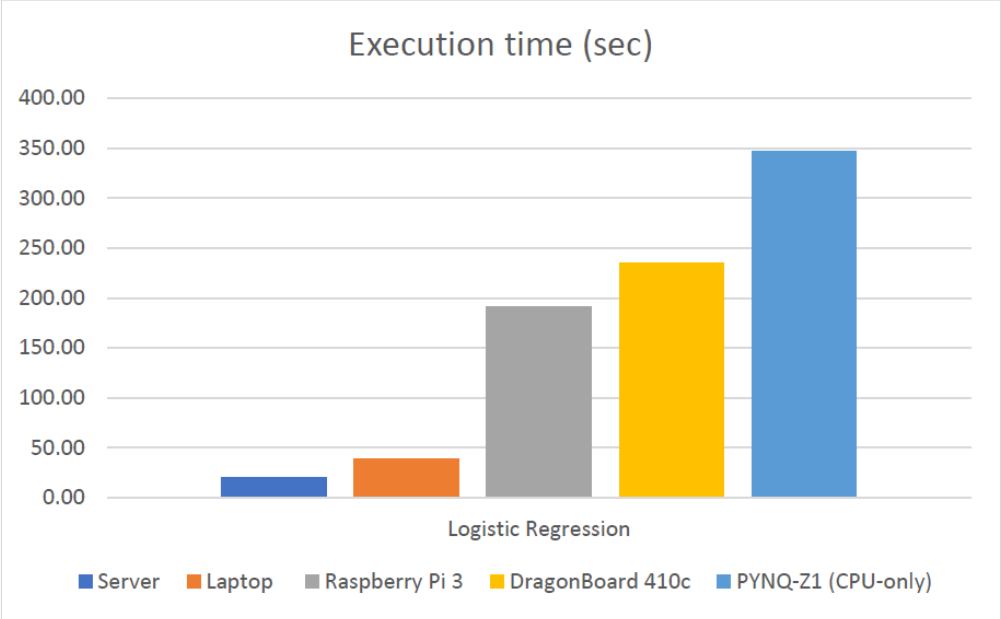
Πίνακας 5.4: Γλωσσάρι ορισμάτων εισόδου των εφαρμογών.

5.1.1 Αποτελέσματα Χρόνου Εκτέλεσης

Τα αποτελέσματα από την εκτέλεση κάθε εφαρμογής, παρουσιάζονται στη συνέχεια:

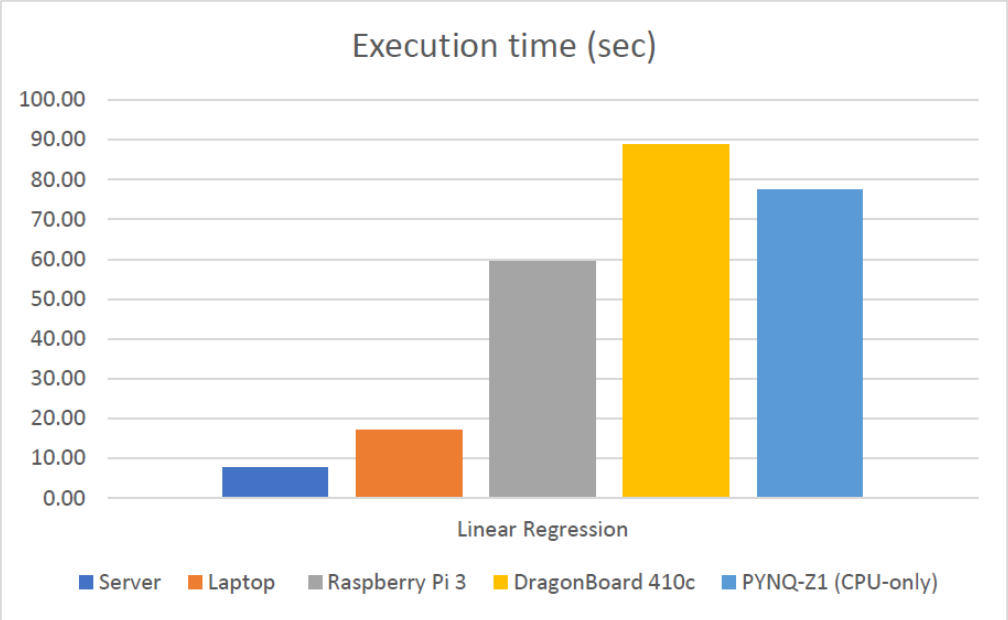
Features	Server	Laptop	Raspberry Pi 3	DragonBoard 410c	PYNQ-Z1
Logistic Regression	19.90	38.59	191.02	235.61	347.39
Linear Regression	7.72	17.15	59.50	88.90	77.46
KMeans	6.40	13.47	43.93	65.77	71.58
Pagerank	32.00	62.95	273.05	383.98	723.38
CC	6.44	15.20	55.12	78.25	83.19
Triangles	5.47	14.65	48.63	70.11	69.41

Πίνακας 5.5: Χρόνος εκτέλεσης (σε δευτερόλεπτα) για κάθε υπό αξιολόγηση πλατφόρομα και εφαρμογή

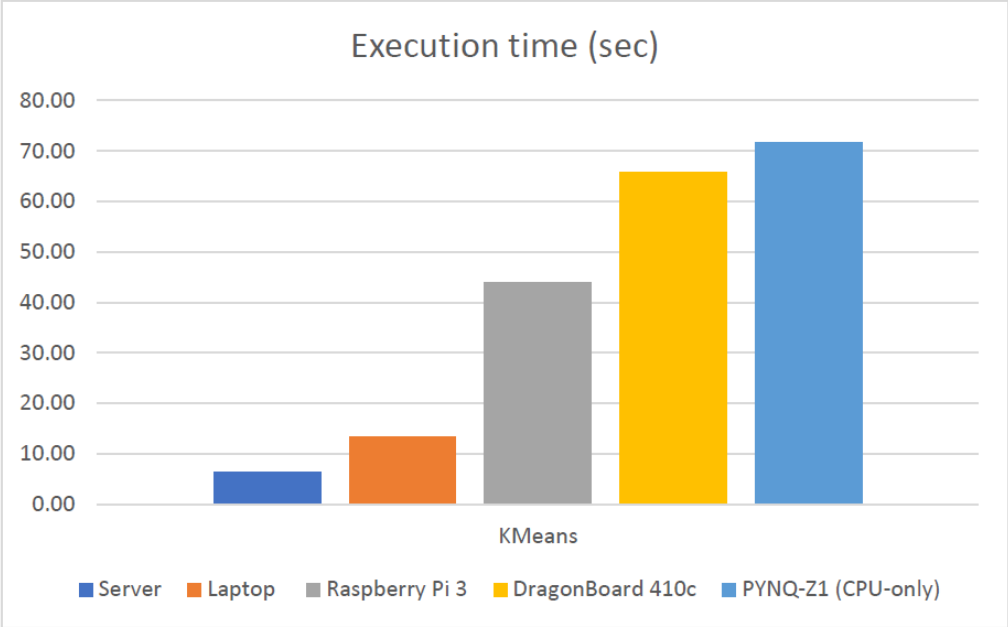


Σχήμα 5.1: Χρόνος εκτέλεσης του αλγορίθμου λογιστικής παλινδρόμησης

Το σχήμα 5.7 απεικονίζει τον χρόνο εκτέλεσης των εφαρμογών για κάθε σύστημα, κανονικοποιημένο στον χρόνο εκτέλεσης των εφαρμογών στο διακομιστή. Όπως φαίνεται, ο συνολικός χρόνος εκτέλεσης των εφαρμογών που εκτελέστηκαν στα συστήματα που βασίζονται σε SoC χαμηλής ισχύος, είναι 6,2 έως 13 φορές υψηλότερος από αυτόν του επεξεργαστή Xeon. Αναλυτικότερα, το Raspberry Pi 3 είναι 6,2 έως 9,6 φορές χειρότερο όσον αφορά το συνολικό χρόνο εκτέλεσης, ενώ το DragonBoard 410c είναι 10,3 έως 12,8 φορές πιο αργό στον ίδιο τομέα. Το PYNQ-Z1, αν και ομολογουμένως διαθέτει τους λιγότερους υπολογιστικούς πόρους, είναι μόλις 14 φορές

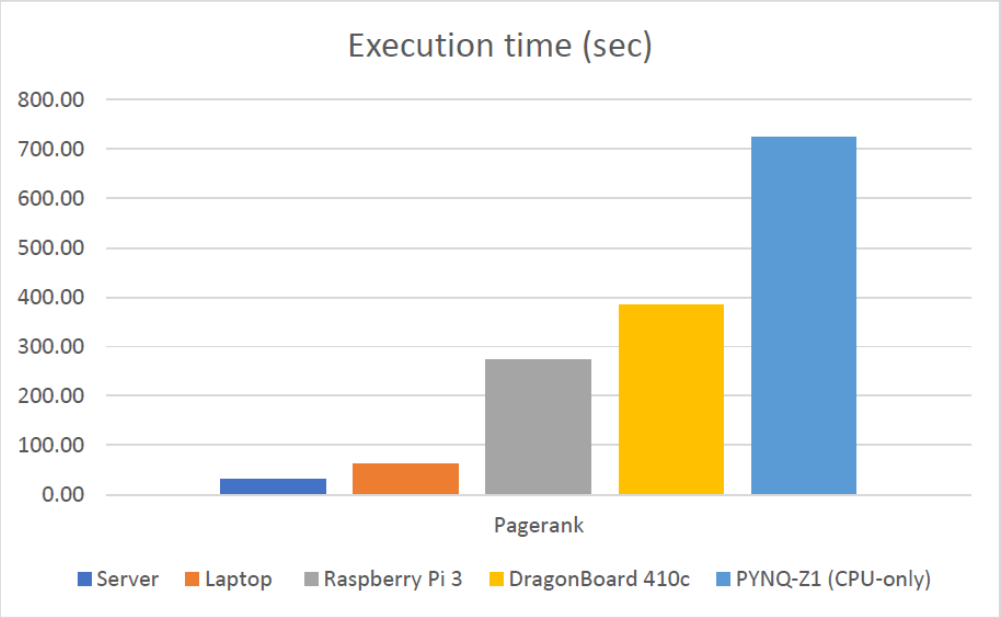


Σχήμα 5.2: Χρόνος εκτέλεσης του αλγορίθμου γραμμικής παλινδρόμησης

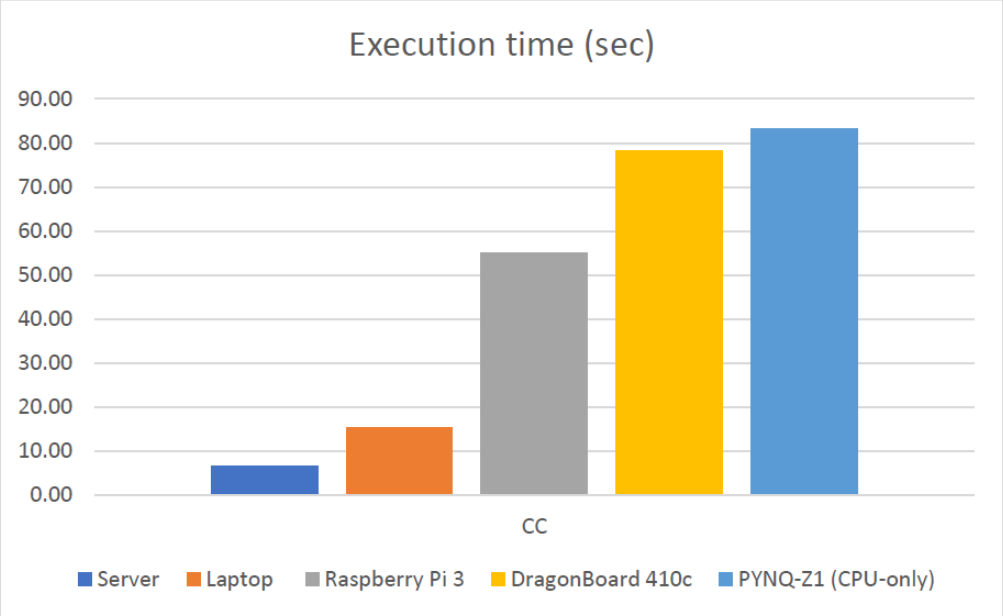


Σχήμα 5.3: Χρόνος εκτέλεσης του αλγορίθμου KMeans

πιο αργό από τον server κατά μέσο όρο. Στην πραγματικότητα, μπορούμε να δούμε ότι τέσσερις από τις έξι εφαρμογές έχουν σχεδόν τον ίδιο χρόνο εκτέλεσης στο PYNQ-Z1 και στο DragonBoard 410c.

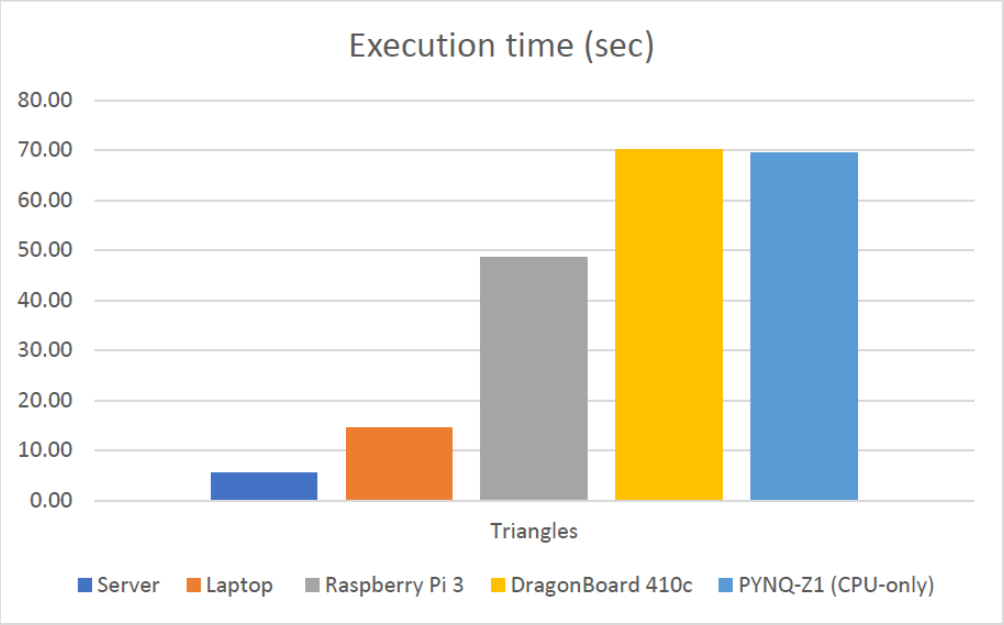


Σχήμα 5.4: Χρόνος εκτέλεσης του αλγορίθμου Pagerank

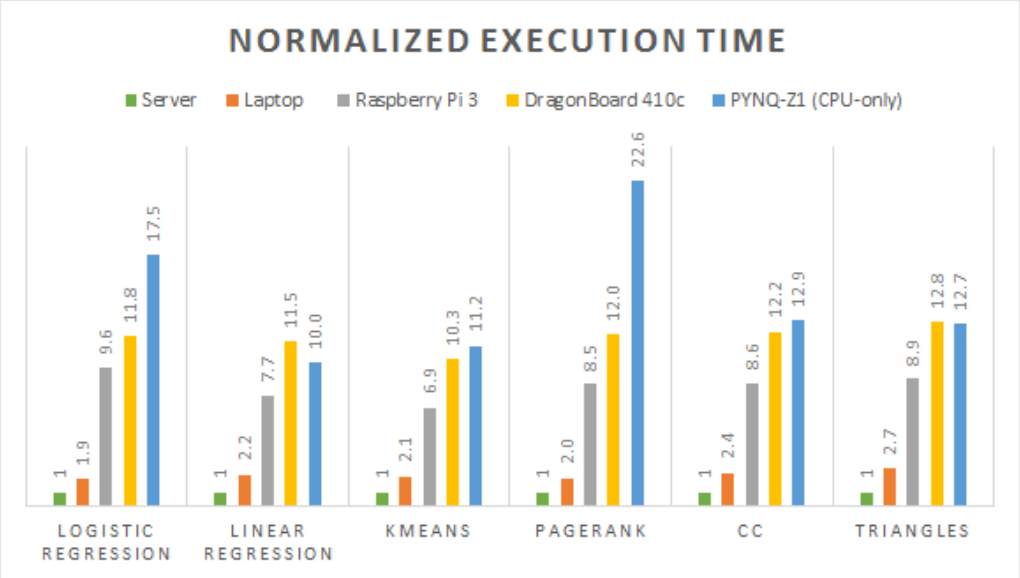


Σχήμα 5.5: Χρόνος εκτέλεσης του αλγορίθμου Connected Components

Είναι πολύ ενδιαφέρον να σημειωθεί ότι, ενώ τα Raspberry Pi 3 και το DragonBoard 410c χρησιμοποιούν τους ίδιους 64-bit A53 επεξεργαστές, οι οποίοι είναι μάλιστα χρονισμένοι στην ίδια ακριβώς συχνότητα των 1.2GHz, παρατηρούμε ότι ο χρόνος εκτέλεσης στην πλατφόρμα Snapdragon πλατφόρμα είναι γενικά υψηλότερος. Πιο αναλυτικά, ο μέσος χρόνος εκτέλεσης στην πλατφόρμα Raspberry η οποία βασίζε-



Σχήμα 5.6: Χρόνος εκτέλεσης του αλγορίθμου Triangles



Σχήμα 5.7: Σύγκριση χρόνου εκτέλεσης των εφαρμογών του Spark. Οι χρόνοι είναι κανονικοποιημένοι ως προς την πλατφόρμα Intel Xeon

ται στο Broadcom SoC είναι 8 φορές μεγαλύτερος από αυτόν της εκτέλεσης στον επεξεργαστή Xeon, ενώ ο χρόνος εκτέλεσης στην πλατφόρμας DragonBoard που βασίζεται στο Snapdragon SoC είναι 11 φορές μεγαλύτερος σε σύγκριση και πάλι με τον επεξεργαστή Xeon. Ο επικρατέστερος λόγος για τον οποίο το DragonBoard είναι χειρότερο από το Raspberry από άποψη απόδοσης, θα μπορούσε να είναι ότι η

Features	Server	Laptop	Raspberry Pi 3	DragonBoard 410c	PYNQ-Z1
Logistic Regression	1	1.9	9.6	11.8	17.5
Linear Regression	1	2.2	7.7	11.5	10.0
KMeans	1	2.1	6.9	10.3	11.2
Pagerank	1	2.0	8.5	12.0	22.6
CC	1	2.4	8.6	12.2	12.9
Triangles	1	2.7	8.9	12.8	12.7

Πίνακας 5.6: Χρόνος εκτέλεσης κανονικοποιημένος ως προς το χρόνο του διακομιστή για κάθε υπό αξιολόγηση πλατφόρομα και εφαρμογή

μνήμη RAM στην πλατφόρμα Snapdragon είναι χρονισμένη σε αρκετά χαμηλότερη συχνότητα (533MHz) από αυτή του Pi 3 (900MHz). Το Spark, όπως έχει ήδη τονιστεί, καταφέρνει να είναι τόσο γρήγορο σε σύγκριση με άλλα cluster computing frameworks, λόγω της ικανότητας του να εκτελεί υπολογισμούς πάνω σε δεδομένα που βρίσκονται ήδη στη μνήμη. Συνεπώς, η ταχύτητα και το μέγεθος της διαθέσιμης μνήμης ενός συστήματος, αποτελούν καθοριστικό παράγοντα για τη συνολική απόδοση του Spark. Ένας άλλος λόγος για τον οποίο βλέπουμε αυτή τη διαφορά στην απόδοση των δύο πλατφορμών, θα μπορούσε να είναι η διαφορετική τεχνολογία στην οποία έχει κατασκευαστεί κάθε SoC. Το Broadcom SoC κατασκευάζεται σε τεχνολογία 40nm, ενώ το Snapdragon SoC κατασκευάζεται σε τεχνολογία 28nm. Τέλος, είναι σημαντικό να επισημάνουμε ότι και το λειτουργικό σύστημα των δύω αυτών συστήματων δεν είναι το ίδιο και συνεπώς θα μπορούσε και αυτό να παίξει σημαντικό ρόλο στο χρόνο εκτέλεσης των εφαρμογών. Ειδικότερα, ενώ και τα δύο λειτουργικά συστήματα βασίζονται στη διανομή debian των Linux, το Raspbian είναι ένα 32-bit λειτουργικό σύστημα, ενώ του λειτουργικό σύστημα του Dragonboard 410c είναι 64-bit. Φυσικά, ο βαθμός στον οποίο κάθε λειτουργικό σύστημα είναι βελτιστοποιημένο για την αντίστοιχη πλατφόρμα παίζει σπουδαίο ρόλο στην συνολική απόδοση κάθε συστήματος.

5.1.2 Αποτελέσματα Ενεργειακής Απόδοσης

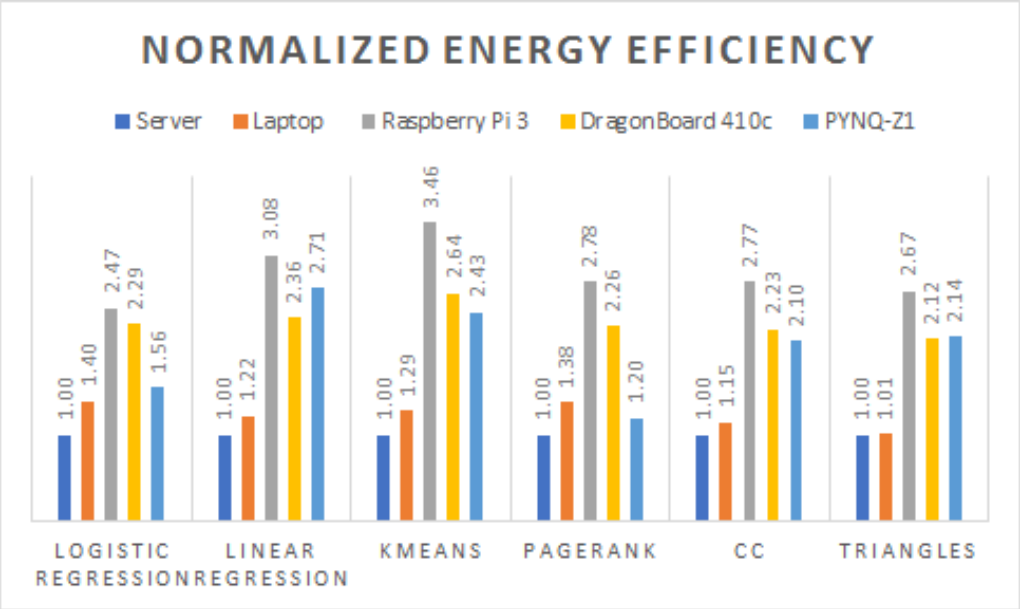
Όσον αφορά την ενεργειακή απόδοση, το βασικό πλεονέκτημα των SoCs εντοπίζεται ακριβώς στο ότι είναι βελτιστοποιημένα για χαμηλή κατανάλωση ενέργειας. Σε αυτή την ενότητα αξιολογούμε την ενεργειακή αποδοτικότητα ($power * execution_time$) των εμπλεκόμενων συστημάτων κατά την υποβολή εφαρμογών προς εκτέλεση από το Apache Spark. Εκτός από τον επεξεργαστή Xeon ο οποίος διαθέτει εργαλεία για

την λεπτομερή παρακολούθηση της κατανάλωσης ενέργειας και επιτρέπει με αυτόν τον τρόπο τη δυνατότητα λήψης μετρήσεων κατανάλωσης για συγκεκριμένες διεργασίες, οι υπόλοιπες πλατφόρμες δεν διαθέτουν κάποιο αντίστοιχο εργαλείο. Αυτό μας αφήνει μόνο με δύο επιλογές: θα μπορούσαμε είτε να τοποθετήσουμε ένα πολύμετρο σε σειρά με το τροφοδοτικό κάθε συστήματος για να μετρήσουμε το ρεύμα που καταναλώνει και στη συνέχεια να το πολλαπλασιάσουμε με την τάση τροφοδοσίας για να υπολογίσουμε την κατανάλωση σε Watt, είτε να μετρήσουμε την κατανάλωση ενέργειας βασιζόμενοι στο TDP (Thermal Design Power) κάθε επεξεργαστή. Ωστόσο, η πρώτη προσέγγιση απαιτεί μια πιο πολύπλοκη υλοποίηση και εφόσον δεν διαθέταμε τα απαραίτητα εργαλεία για να πάρουμε τις εν λόγω μετρήσεις, επιλέξαμε τη δεύτερη λύση. Επομένως, αυτή η αξιολόγηση για την κατανάλωση ενέργειας των SoCs, βασίζεται στο TDP των επεξεργαστών. Το TDP αναφέρεται στη μέση καταναλισκόμενη ισχύ των επεξεργαστών όταν εργάζονται με πλήρες φορτίο. Είναι σημαντικό να σημειωθεί, ότι η σύγκριση της ενεργειακής απόδοσης των εμπλεκόμενων συστημάτων είναι ενδεικτική και χρησιμοποιείται ως μία πρώτη προσέγγιση για τη δυνητική εξοικονόμηση ενέργειας που θα μπορούσε να επιτευχθεί χρησιμοποιώντας χαμηλής ισχύος επεξεργαστές.

Πριν προχωρήσουμε στα αποτελέσματα αυτής της αξιολόγησης, θα πρέπει να επισημάνουμε μερικά πράγματα σχετικά με τις πλατφόρμες Raspberry Pi3 και PYNQ-Z1. Όπως είδαμε στον πίνακα 5.1, δεν ήταν διαθέσιμη η τιμή TDP των επεξεργαστών τους. Ωστόσο, για το Raspberry Pi 3, καταφέραμε να βρούμε benchmarks τα οποία δείχνουν ότι ακόμα και σε λειτουργία πλήρους φόρτου, η πλατφόρμα δεν κατανάλωνε περισσότερο από 0,6A [29]. Υποθέτοντας ότι η τάση εισόδου της τροφοδοσίας είναι 5V, έχουμε μια κατανάλωση μόλις $0.6A * 5V = 3W$. Για να είμαστε αρκετά δίκαιοι, κάναμε μια απαισιόδοξη υπόθεση ότι το TDP του Broadcom SoC είναι περίπου 4W. Όσον αφορά την περίπτωση του PYNQ-Z1, μετρήσαμε την κατανάλωση ενός αντίστοιχου συστήματος που ονομάζεται ZedBoard το οποίο παρέχει εργαλεία για την παρακολούθηση και μέτρηση της κατανάλωσής του. Το ZedBoard, έχει ακριβώς το ίδιο Zynq SoC με το PYNQ-Z1. Η μετρούμενη κατανάλωση, ακόμη και όταν έγινε χρήση της επαναπρογραμματιζόμενης λογικής (FPGA) του Zynq δεν ξεπέρασε τα 3,2W, οπότε και πάλι υποθέσαμε ότι το TDP του Zynq είναι 3,5W.

Με βάση τα παραπάνω, είμαστε πλέον έτοιμοι να προχωρήσουμε στα αποτελέσματα

της παρούσας αξιολόγησης. Το σχήμα 5.8 απεικονίζει κανονικοποιημένη την ενεργειακή απόδοση κάθε εμπλεκόμενου συστήματος, στην ενεργειακή απόδοση του server.



Σχήμα 5.8: Σύγκριση της αποδοτικότητας ενέργειας, κανονικοποιημένης ως προς την πλατφόρμα Intel Zeon

Όπως φαίνεται στο παραπάνω σχήμα, η κατανάλωση ενέργειας των SoC χαμηλής ισχύος είναι 2 - 3,5 φορές καλύτερη από την κατανάλωση ενέργειας του Intel Xeon επεξεργαστή. Ενώ η πλατφόρμα που βασίζεται στον Xeon έχει την καλύτερη απόδοση όσον αφορά τον χρόνο εκτέλεσης των εφαρμογών, η ισχύς που χρειάζεται προκαλεί μια συνολικά υψηλή κατανάλωση ενέργειας. Από την άλλη πλευρά, τα SoC χαμηλής ισχύος παρέχουν πολύ καλύτερη ενεργειακή απόδοση το οποίο έχει αντίκτυπο στον χρόνο εκτέλεσης των εφαρμογών. Είναι εμφανές λοιπόν, ότι σε περιπτώσεις όπου η ενεργειακή απόδοση είναι το βασικό κριτήριο για την επιλογή ενός συστήματος, τα SoCs χαμηλής ισχύος αποτελούν αναμφισβήτητα την καλύτερη επιλογή.

Features	Server	Laptop	Raspberry Pi 3	DragonBoard 410c	PYNQ-Z1
Logistic Regression	1	1.4	2.47	2.29	1.56
Linear Regression	1	1.22	3.08	2.36	2.71
KMeans	1	1.29	3.46	2.64	2.43
Pagerank	1	1.38	2.78	2.26	1.2
CC	1	1.15	2.77	2.23	2.10
Triangles	1	1.01	2.67	2.12	2.14

Πίνακας 5.7: Ενεργειακή αποδοτικότητα κάθε πλατφόρμας, κανονικοποιημένη ως προς την αποδοτικότητα της πλατφόρμας Intel Xeon

5.2 Εκτέλεση σε Επιταχυντές Υλικού

Σε αυτή την ενότητα των αποτελεσμάτων, θα γίνει αξιολόγηση της απόδοσης του SPynq framework που περιγράψαμε στο κεφάλαιο 4, για ένα σενάριο χρήσης του αλγορίθμου λογιστικής παλινδρόμησης (υποενότητα 4.4.2). Για το προαναφερθέν σενάριο, δημιουργήσαμε ένα μοντέλο κατάταξης (classification model) το οποίο αποτελείται από 784 features (εικόνες 28∗28) και 10 labels (ένα για κάθε ψηφίο στο διάστημα 0-9) χρησιμοποιώντας 40 χιλιάδες διαθέσιμα δείγματα εκπαίδευσης, για ένα πρόβλημα αναγνώρισης χειρόγραφων ψηφίων (handwritten digits recognition problem).
Για να αξιολογήσουμε την απόδοση του SPynq cluster, δημιουργήσαμε ένα ακόμη Spark cluster αποτελούμενο από τέσσερις Worker κόμβους, οι οποίοι βρίσκονται στους Xeon πυρήνες ενός διακομιστή. Ο πίνακας 5.8 εμφανίζει τα χαρακτηριστικά των δύο συστημάτων.

Features	**Server**	**PYNQ-Z1**
Vendor	Intel	Xilinx
Device	E5-2658	Zynq XC7Z020
Cores(threads)	12(24)	2
Processor	E5-2658	A9
Architecture	64-bit	32-bit
INstruction Set	CISC	RISC
Process	22nm	28nm
Clock Frequency	2.2GHz	667MHz
Level 1 cache	380kB	32kB
Level 2 cache	3072kB	512kB
Level 3 cache	30MB	-
TDP	105W	3.5
Operating System	Ubuntu	Ubuntu

Πίνακας 5.8: Κύρια χαρακτηριστικά των αξιολογούμενων πλατφορμών Xeon και Zynq

Οι JVM (Java Virtual Machine) διεργασίες του Spark, δεσμεύουν το μεγαλύτερο μέρος της διαθέσιμης μνήμης RAM (512 MB) του PYNQ-Z1, θέτοντας ένα περιορισμό στην εφαρμογή μας, η οποία απαιτεί από την κύρια μνήμη να διατηρεί και να έχει επανειλημμένα πρόσβαση στο σύνολο των δεδομένων που έχουν διαβαστεί από το HDFS.
Αντίθετα, ο server έχει ένα Xeon CPU με 12 πυρήνες και συνολικά 24 νήματα (threads). Προκειμένου η σύγκριση με τους 4 κόμβους του Pynq cluster να είναι

δίκαιη, μόνο 4 από τα 24 διαθέσιμα νήματα του συστήματος Xeon χρησιμοποιούνται από το Spark cluster.

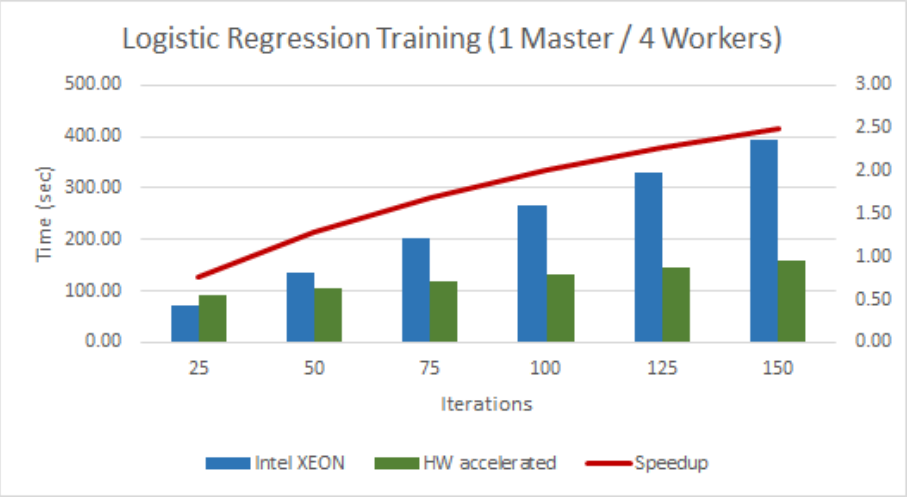
Επιπλέον, συγκρίναμε την εκτέλεση της εφαρμογής στο SPynq framework, που κάνει χρήση επιταχυντών, με την εκτέλεση της ίδιας εφαρμογής μόνο στους ARM πυρήνες. Με αυτόν τον τρόπο, είμαστε σε θέση να βγάλουμε συμπεράσματα και για την περίπτωση των ενσωματωμένων εφαρμογών στις οποίες μόνο ενσωματωμένοι επεξεργαστές χαμηλής ισχύος μπορούν να χρησιμοποιηθούν, λόγω των αυστηρών προδιαγραφών για χαμηλή κατανάλωση.

Η καθυστέρηση που εισάγεται από την επικοινωνία του επεξεργαστή με την επαναπρογραμματιζόμενη λογική, ειδικά σε περιπτώσεις που είναι συχνή και αμφίδρομη, μπορεί να προκαλέσει μία σημαντική επιβάρυνση στη συνολική απόδοση του συστήματος και να μειώσει κατά αυτόν τον τρόπο την επιτάχυνση που παρέχει ο επιταχυντής. Ωστόσο, σε εφαρμογές όπου ο επεξεργαστής αποστέλλει μαζικά μία μεγάλη ποσότητα δεδομένων (π.χ. μέσω του AXI streaming interface), ο απαιτούμενος χρόνος για την επικοινωνία των δύο οντοτήτων, επικαλύπτεται από τον χρόνο υπολογισμού των αποτελεσμάτων. Στην περίπτωση μας τώρα, όπου γίνεται χρήση του BGD αλγορίθμου λογιστικής παλινδρόμησης, ο επεξεργαστής πρέπει να στείλει ένα μεγάλο όγκο δεδομένων για να εκπαιδεύσει το ζητούμενο μοντέλο και ως εκ τούτου το κόστος της επικοινωνίας (μετρούμενο σε χρόνο) επικαλύπτεται από τον χρόνο των απαιτούμενων υπολογισμών. Ωστόσο, μια απλή οντότητα DMA (Direct Memory Access), δεν μπορεί να χειριστεί περισσότερα από 8 MB (στην περίπτωσή μας 2600 γραμμές δεδομένων) σε μία μόνο μεταφορά. Έτσι, χωρίζοντας το RDD σε κομμάτια των 4 με 5.2 χιλιάδων γραμμών (έχουμε 2 DMAs διαθέσιμους για τη μεταφορά δεδομένων), μπορούμε να εκμεταλλευτούμε στο μέγιστο τον επιταχυντή μας.

5.2.1 Αποτελέσματα Χρόνου Εκτέλεσης

Παρακάτω, παρουσιάζονται τα αποτελέσματα της αξιολόγησης, βάσει του χρόνου εκτέλεσης της εξεταζόμενης εφαρμογής.

Το σχήμα 5.9 απεικονίζει τον χρόνο εκτέλεσης της εφαρμογής της λογιστικής παλινδρόμησης για δύο εκτελέσεις και για διάφορους αριθμούς επαναλήψεων. Η πρώτη εκτέλεση αφορά έναν υψηλής απόδοσης επεξεργαστή x86 _64 αρχιτεκτονικής (Xeon E5 2658), ο οποίος λειτουργεί σε συχνότητα 2.2 GHz ενώ η δεύτερη το Pynq cluster



Σχήμα 5.9: Επιτάχυνση σε σχέση με το πλήθος επαναλήψεων, χρησιμοποιώντας το προτεινόμενο Python API (1)

που διαμορφώσαμε και υλοποιήσαμε, το οποίο κάνει χρήση της επαναπρογραμματιζόμενης λογικής. Ένα αρχείο 40000 γραμμών χωρισμένο σε κομμάτια των 5000 γραμμών χρησιμοποιείται για την εκπαίδευση του μοντέλου του αλγορίθμου και στις δύο περιπτώσεις εκτέλεσης. Όπως φαίνεται, ο συντελεστής επιτάχυνσης που πετυχαίνουμε από την εκτέλεση της εφαρμογής στο Pynq cluster, είναι ανάλογος του αριθμού των επαναλήψεων.

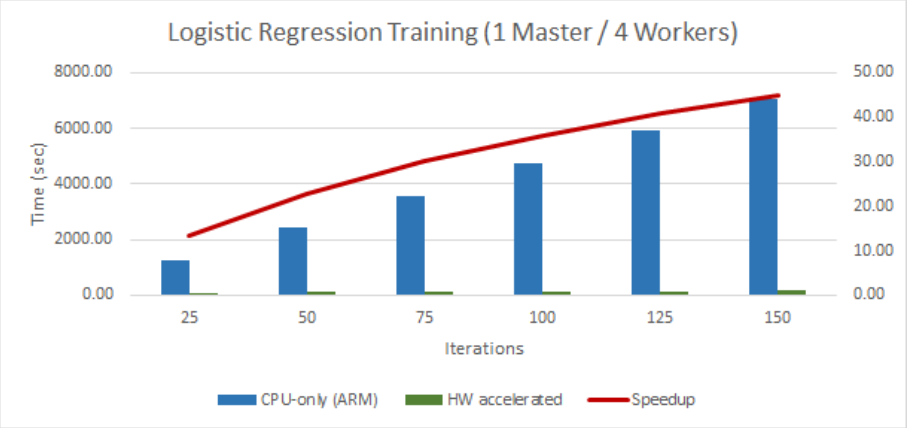
Πιο αναλυτικά, στους PYNQ-Z1 κόμβους η εξαγωγή των δεδομένων (η οποία περιλαμβάνει το διάβασμα των δεδομένων από τα αντίστοιχα αρχεία καθώς και όλους τους απαιτούμενους μετασχηματισμούς στα RDDs) διαρκεί περίπου 81 δευτερόλεπτα, ενώ κάθε επανάληψη του αλγορίθμου ολοκληρώνεται σε 0,53 δευτερόλεπτα εφόσον τα δεδομένα εκπαίδευσης του αλγορίθμου έχουν ήδη αποθηκευτεί σε προσωρινούς buffers. Από την άλλη πλευρά, ο Xeon διαβάζει και μετασχηματίζει τα δεδομένα μόνο σε 7.5 δευτερόλεπτα, αλλά κάθε επανάληψη του αλγορίθμου διαρκεί περίπου 2.6 δευτερόλεπτα.

Επομένως, η επιτάχυνση εξαρτάται από τον αριθμό των επαναλήψεων που εκτελούνται. Για τη συγκεκριμένη εφαρμογή, το μοντέλο μπορεί να επιτύχει ακρίβεια έως και 91,5% στις 100 επαναλήψεις, ενώ η αντίστοιχη επιτάχυνση ισούται με έναν συντελεστή 2x, σε σχέση με την εκτέλεση στον Xeon επεξεργαστή. Ωστόσο, υπάρχουν εφαρμογές στις οποίες απαιτείται πολύ μεγαλύτερος αριθμός επαναλήψεων. Σε αυτή την περίπτωση, μπορούμε να πετύχουμε ακόμα μεγαλύτερη επιτάχυνση.

Worker	Data Extraction	BGD Algorithm Computations (per iteration)
Xeon	7.5	2.6
ARM	78.4	46.6
Pynq (ARM+FPGA)	80.5 (ARM)	0.51 (FPGA)

Πίνακας 5.9: Χρόνος εκτέλεσης (σε δευτερόλεπτα) των συναρτήσεων που εκτελούνται στους workers

Ο πίνακας 5.9 δείχνει το χρόνο εκτέλεσης των δύο κύριων συναρτήσεων που εκτελούνται στους Worker κόμβους. Στις περιπτώσεις του Xeon και του ARM, τόσο η εξαγωγή δεδομένων όσο και ο BGD αλγόριθμος εκτελούνται στους επεξεργαστές κάθε συστήματος, ενώ στην περίπτωση του Pynq η εξαγωγή δεδομένων εκτελείται στους ARM επεξεργαστές ενώ η συνάρτηση BGD εκτελείται στην επαναπρογραμματιζόμενη λογική (FPGA). Επομένως, για εφαρμογές όπου απαιτείται μεγαλύτερος αριθμός επαναλήψεων, η ταχύτητα του συστήματος (έναντι του Xeon) συγκλίνει σε έναν συντελεστή 5x̃ καθώς ο χρόνος για την εξαγωγή των δεδομένων, σε σχέση με τον συνολικό χρόνο εκτέλεσης, καθίσταται αμελητέος.



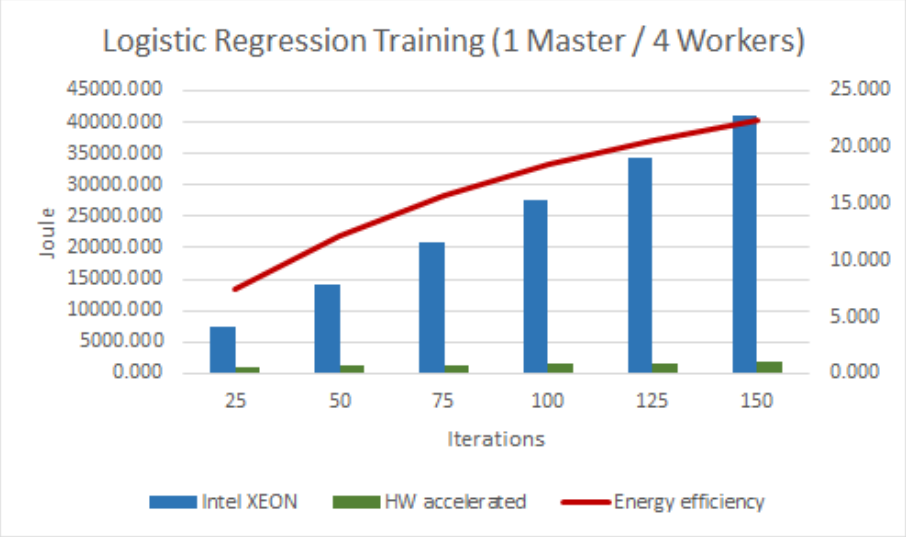
Σχήμα 5.10: Επιτάχυνση σε σχέση με το πλήθος επαναλήψεων, χρησιμοποιώντας το προτεινόμενο Python API (2)

Το σχήμα 5.10 δείχνει την ταχύτητα και το χρόνο εκτέλεσης της εφαρμογής για την περίπτωση του SPynq cluster όπου γίνεται χρήση επιταχυντών, σε σύγκριση και πάλι με την περίπτωση του SPynq cluster όπου χρησιμοποιούνται μόνο οι ARM επεξεργαστές. Από αυτή τη σύγκριση, φαίνεται ότι μπορούμε να επιτύχουμε έως και 36 φορές καλύτερη ταχύτητα εκτέλεσης με χρήση επιταχυντών υλικού. Η παρούσα σύγκριση, είναι χρήσιμη για την περίπτωση ενσωματωμένων εφαρμογών, όπου υπάρχουν αυστηροί περιορισμοί στην κατανάλωση ενέργειας και οι επεξεργαστές υψηλής

απόδοσης δεν μπορούν να χρησιμοποιηθούν ακριβώς γιατί τα χαρακτηριστικά τους δεν συνάδουν με αυτές τις απαιτήσεις.

5.2.2 Αποτελέσματα Ενεργειακής Απόδοσης

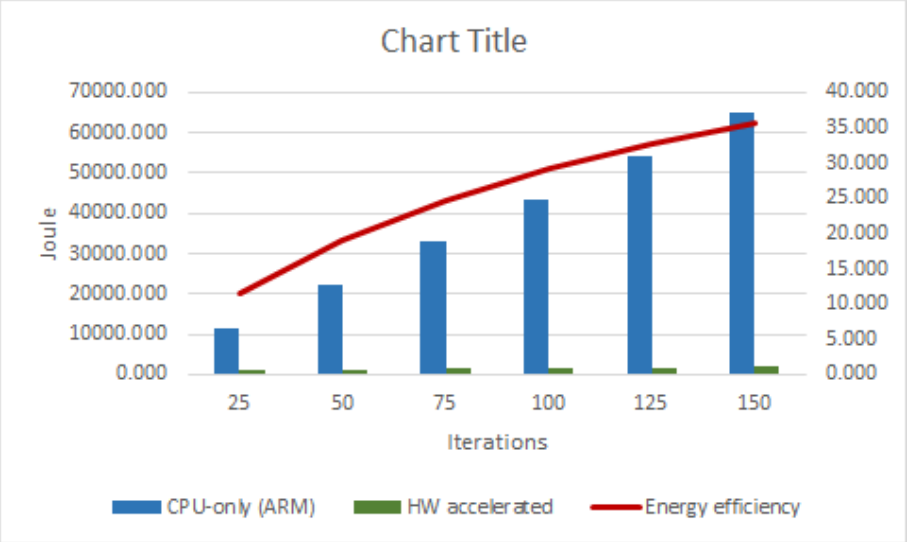
Προκειμένου να αξιολογήσουμε την εξοικονόμηση ενέργειας, μετρήσαμε τη μέση καταναλισκόμενη ισχύ, για τις περιπτώσεις εκτέλεσης του αλγορίθμου που περιγράψαμε νωρίτερα (εκτέλεση μόνο στους Xeon cores, εκτέλεση μόνο στους ARM core και εκτέλεση στους ARM cores κάνοντας χρήση επιταχυντών υλικού). Για να μετρηθεί η κατανάλωση ενέργειας του εξυπηρετητή Xeon, χρησιμοποιήθηκε το Processor Counter Monitor (PCM) API της Intel. Εκτός των άλλων, το PCM API επιτρέπει την καταγραφή της ενέργειας που καταναλώνεται από το CPU και την DRAM μνήμη, για την εκτέλεση μιας δεδομένης εφαρμογής. Μετρήσαμε επίσης την κατανάλωση ενέργειας του Zynq, χρησιμοποιώντας το evaluation board ZC702, το οποίο φιλοξενεί την ίδια συσκευή Zynq με αυτήν του PYNQ-Z1. Για την μέτρηση αυτή, χρησιμοποιήθηκαν οι ελεγκτές ισχύος που διαθέτει το σύστημα.



Σχήμα 5.11: Κατανάλωση ενέργειας στις πλατφόρμες Xeon και Zynq βάσει του πλήθους επαναλήψεων

Το σχήμα 5.11 δείχνει την κατανάλωση ενέργειας για την περίπτωση εκτέλεσης του αλγορίθμου στους επεξεργαστές Xeon καθώς και στο Zynq όταν γίνεται η χρήση των επιταχυντών υλικού. Αντίθετα, το σχήμα 5.12 απεικονίζει την κατανάλωση ενέργειας της πλατφόρμας Zynq για τις δύο περιπτώσεις εκτέλεσης (α) μόνο στους πυρήνες ARM και β)χρησιμοποιώντας επιταχυντές υλικού). Όσον αφορά το πρώτο σχήμα, η

μέση κατανάλωση ενέργειας του επεξεργαστή Xeon και των DRAM είναι 103 Watt, ενώ ένας κόμβος Zynq (συμπεριλαμβανομένων του MPSoC FPGA και της DRAM μνήμης) καταναλώνουν περίπου 2,6 Watt κατά τη διάρκεια της εξαγωγής δεδομένων (διάβασμα αρχείων εισόδου κλπ.) και 3,2 Watt κατά τη διάρκεια των υπολογισμών στον επιταχυντή υλικού . Συνεπώς, σε αυτή την περίπτωση μπορούμε να επιτύχουμε έως και 18 φορές καλύτερη ενεργειακή απόδοση (για 100 επαναλήψεις εκτέλεσης του αλγορίθμου) λόγω της χαμηλότερης κατανάλωσης ενέργειας και του χαμηλότερου χρόνου εκτέλεσης.



Σχήμα 5.12: Κατανάλωση ενέργειας στις πλατφόρμες μόνο-ARM και Zynq βάσει του πλήθους επαναλήψεων

Ακολούθως, στο σχήμα 5.12 φαίνεται ότι αν και η κατανάλωση ενέργειας του Zynq είναι ελαφρώς υψηλότερη όταν γίνεται χρήση της επαναπρογραμματιζόμενης λογικής (σε σχέση με την κατανάλωση από τη χρήση μόνο των επεξεργαστών A9, που ισούται κατά μέσο όρο με 2.3W ανά κόμβο), μπορούμε τελικά να επιτύχουμε έως και 29 φορές χαμηλότερη κατανάλωση ενέργειας λόγω του σημαντικά υψηλότερου χρόνου εκτέλεσης στην περίπτωση που χρησιμοποιούνται μόνο οι πυρήνες ARM.

6 Επίλογος

6.1 Συμπεράσματα

Σε αυτή την εργασία, παρουσιάσαμε όλα τα απαραίτητα βήματα για την κατασκευή και το στήσιμο του Spark σε ενσωματωμένα συστήματα χαμηλής κατανάλωσης, όπως το Raspberry Pi 3 και το Dragonboard 410, καθώς και όλες τις απαραίτητες τροποποιήσεις που χρειάστηκαν για να «τρέξει» το Spark. Ακόμη, δημιουργήσαμε ένα σύμπλεγμα υπολογιστών, απαρτιζόμενο από τέσσερεις PYNQ-Z1 κόμβους και έναν ακόμη βασισμένο στην Intel αρχιτεκτονική, στο οποίο εγκαταστήθηκε το Spark. Έπειτα, υλοποιήσαμε το SPynq framework προκειμένου να υπάρχει διαφανής χρήση επιταχυντών υλικού από τις εφαρμογές του Spark. Τέλος, διενεργήθηκαν δύο αξιολογήσεις. Στην πρώτη, εκτιμήσαμε τον χρόνο εκτέλεσης έξι ευρέως χρησιμοποιούμενων εφαρμογών του Spark, από τους τομείς της μηχανικής εκμάθησης και της επεξεργασίας γράφων. Στη συνέχεια, συγκρίναμε τα αποτελέσματα αυτά, με τα αποτελέσματα από την εκτέλεση των ίδιων εφαρμογών σε έναν τυπικό διακομιστή και έναν προσωπικό υπολογιστή και τέλος χρησιμοποιήσαμε την TDP μετρική για να προσεγγίσουμε την ενεργειακή απόδοση όλων των εμπλεκόμενων πλατφορμών. Όσον αφορά τη δεύτερη αξιολόγηση, αρχικά υλοποιήθηκε ένας επιταχυντής υλικού για ένα σενάριο χρήσης του αλγορίθμου λογιστικής παλινδρόμησης. Έπειτα, δημιουργήθηκε ένα νέο Python API καθώς και περαιτέρω νέες βιβλιοθήκες για το Spark που χρησιμοποιούν μεθόδους του προαναφερθέντος API. Τέλος, αξιολογήθηκε ο χρόνος εκτέλεσης του αλγορίθμου λογιστικής παλινδρόμησης για ένα πρόβλημα αναγνώρισης ψηφίων (από το 0 έως το 9) γραμμένων στο χέρι (handwritten digits recognition problem) και έπειτα προσεγγίστηκε η κατανάλωση ενέργειας του SPynq cluster από την παραπάνω εκτέλεση. Τέλος, τα αποτελέσματα συγκρίθηκαν με αυτά από την ίδια εκτέλεση σε

έναν τυπικό διακομιστή που χρησιμοποιείται στα κέντρα δεδομένων, ο οποίος βασίζεται σε έναν Intel Xeon επεξεργαστή.

Συνολικά, το στήσιμο του Spark σε ενσωματωμένα συστήματα ενέχει προκλήσεις. Τα βασικά προβλήματα που αντιμετωπίσαμε μπορούν να χωριστούν σε δύο κατηγορίες: σε θέματα συμβατότητας και σε προβλήματα που απορρέουν από τους διαθέσιμους πόρους υλικού. Όσον αφορά το πρώτο πρόβλημα, με την πάροδο του χρόνου βλέπουμε ότι τα θέματα συμβατότητας μειώνονται, δεδομένου ότι οι εφαρμογές καθώς και άλλα μεμονωμένα πακέτα υποστηρίζουν όλο και περισσότερες αρχιτεκτονικές συστημάτων. Από την άλλη πλευρά, ο πιο σημαντικός παράγοντας που καθιστά δυνατή ή αντίστοιχα αδύνατη την εκτέλεση εφαρμογών ανάλυσης μεγάλων δεδομένων σε ένα σύστημα, είναι η ποσότητα της διαθέσιμης μνήμης. Η έλλειψη μιας μεγάλης μνήμης RAM, θα μπορούσε να σημαίνει ότι τα δεδομένα δεν χωράνε στη μνήμη. Ωστόσο, το Spark είναι τόσο γρήγορο ακριβώς γιατί οι περισσότεροι υπολογισμοί γίνονται με δεδομένα που ήδη βρίσκονται στη μνήμη RAM. Γίνεται επομένως αντιληπτό, ότι η έλλειψη μιας μεγάλης και γρήγορης RAM έχει δραματικές επιπτώσεις στο χρόνο εκτέλεσης των εφαρμογών μέσω του Spark.

Ακόμη, δοκιμάσαμε και στήσαμε το Spark από πηγαίο κώδικα τόσο στο Raspberry Pi 3 όσο και στο DragonBoard 410c. Φυσικά, η παραπάνω διαδικασία διενεργήθηκε μόνο για ερευνητικούς σκοπούς, προκειμένου να δοκιμαστούν οι δυνατότητες των προαναφερθέντων ενσωματωμένων πλατφορμών. Ο χρόνος που απαιτείται για την κατασκευή ενός αναδιανεμητέου πακέτου του Spark σε έναν τυπικό διακομιστή και η μετέπειτα αντιγραφή αυτού στα ενσωματωμένα συστήματα, είναι σημαντικά μικρότερος από το χρόνο που δαπανήθηκε στην περίπτωσή μας. Σε περίπτωση που ένας χρήστης δεν χρειάζεται να ορίσει τις δικές του παραμέτρους κατά την κατασκευή του Spark, η καλύτερη επιλογή που έχει είναι να κατεβάσει ένα έτοιμο (pre-built) πακέτο του Spark. Με αυτόν τον τρόπο, όχι μόνο μπορεί να γλιτώσει πολύτιμο χρόνο, αλλά και να αποφύγει την ενδεχόμενη αντιμετώπιση άλλων προβλημάτων.

Τέλος, από την αξιολόγηση που πραγματοποιήσαμε, είδαμε ότι οι επεξεργαστές που βασίζονται σε SoC είναι 8 έως 11 φορές χειρότεροι όσον αφορά στο χρόνο εκτέλεσης των εφαρμογών που αξιολογήθηκαν, ενώ παράλληλα λόγω της χαμηλότερης κατανάλωσης ενέργειας έχουν τη δυνατότητα να προσφέρουν πολύ καλύτερη ενεργειακή απόδοση. Για εφαρμογές μηχανικής εκμάθησης βασισμένες στο Spark μπορούν να

παρέχουν έως και 3 φορές καλύτερη ενεργειακή απόδοση, ενώ στον τομέα επεξεργασίας γράφων μπορούν να παρέχουν έως και 3,5 φορές καλύτερη ενεργειακή απόδοση. Επομένως, σε περιπτώσεις όπου μας ενδιαφέρει περισσότερο η ενεργειακή απόδοση και όχι κυρίως ο χρόνος εκτέλεσης, οι εξυπηρετητές που βασίζονται σε SoCs θα μπορούσαν να αποτελέσουν μια πολλά υποσχόμενη εναλλακτική λύση, στην μείωση της δαπανώμενης ισχύος και του συνολικού κόστους ιδιοκτησίας (TCO) των κέντρων δεδομένων.

Όσον αφορά τη δεύτερη αξιολόγηση, είδαμε ότι τα παρόντα frameworks που χρησιμοποιούνται για την ανάλυση μεγάλων δεδομένων, όπως το Spark, δεν υποστηρίζουν την απρόσκοπτη χρήση επιταχυντών υλικού. Ωστόσο, με τη βοήθεια του SPynq framework που υλοποιήθηκε στην παρούσα εργασία, είδαμε ότι οι επιταχυντές υλικού μπορούν να βελτιώσουν σημαντικά την απόδοση και την ενεργειακή αποδοτικότητα τέτοιων εφαρμογών. Διαπιστώθηκε έτσι, ότι το προτεινόμενο σχήμα μπορεί να χρησιμοποιηθεί τόσο σε συστήματα υψηλής απόδοσης, για τη μείωση της κατανάλωσης ενέργειας (μέχρι 18 φορές) καθώς και του χρόνου εκτέλεσης (μέχρι 2 φορές), όσο και σε ενσωματωμένα συστήματα, όπου μπορεί να επιτύχει έως και 36 φορές καλύτερο χρόνο εκτέλεσης σε σύγκριση με τα αποτελέσματα από την εκτέλεση σε ενσωματωμένους επεξεργαστές χαμηλής ισχύος (όπως οι επεξεργαστές ARM) και 29 φορές χαμηλότερη κατανάλωση ενέργειας. Τέλος, δείξαμε ότι το προτεινόμενο σχήμα μπορεί να χρησιμοποιηθεί για την υποστήριξη οποιουδήποτε είδους επιταχυντών υλικού, προκειμένου να επιταχυνθεί ο χρόνος εκτέλεσης των υπολογιστικά απαιτητικών τμημάτων των εφαρμογών μηχανικής εκμάθησης καθώς και άλλων εφαρμογών ανάλυσης δεδομένων που βασίζονται στο Spark.

6.2 Μελλοντικά Σχέδια

Όσον αφορά τα μελλοντικά σχέδια, θα μπορούσαν να υπάρξουν διάφορες επεκτάσεις και βελτιώσεις.

Αρχικά, αν και από την αξιολόγηση του SPynq framework είδαμε ότι μπορούμε να πετύχουμε σηματνική βελτίωση στον χρόνο εκτέλεσης των εφαρμογών και παράλληλα μείωση της κατανάλωσης ενέργειας, θα πρέπει να εμπλουτίσουμε την βιβλιοθήκη mllib_accel που υλοποιήσαμε, ώστε να υποστηρίζει περισσότερους,νέους αλγορίθμους μηχανικής εκμάθησης όπως ο KMeans, ο Linear Regression κ.α. Για το σκοπό

αυτό, θα πρέπει να δημιουργηθουν νέοι επιταχυντές υλικού και νέα Python APIs μέσω των οποίων θα επιτυγχάνεται η επικοινωνία με αυτούς.

Επιπλέον, όπως είδαμε και στο Κεφάλαιο 4, αν και η ετερογένεια του υλικού των Worker κόμβων υποστηρίζεται από το Spark, στην πραγματικότητα δεν μπορούμε να επωφεληθούμε από αυτή. Πιο συγκεκριμένα, είδαμε ότι μπορεί κανείς εύκολα να χρησιμοποιήσει μεταβλητές περιβάλλοντος (όπως το SPARK_WORKER_TYPE), για να διαχωρίσει κόμβους που φιλοξενούν επαναπρογραμματιζόμενη λογική από κόμβους που δεν διαθέτουν τέτοια μονάδα, ωστόσο η παραπάνω δυνατότητα δεν προσφέρει στην παρούσα φάση κάποιο πλεονέκτημα. Ειδικά για τον συγκεκριμένο αλγόριθμο λογιστικής παλινδρόμησης που χρησιμοποιήσαμε, τα επιμέρους αποτελέσματα από κάθε Worker κόμβο μίας συγκεκριμένης επανάληψης του αλγορίθμου, πρέπει να επιστρέψουν στον Master κόμβο και να αθροιστούν μεταξύ τους προκειμένου να υπολογιστούν τα νέα βάρη. Αυτά τα νέα βάρη θα χρησιμοποιηθούν ως είσοδος στην επόμενη επανάληψη του αλγορίθμου. Με αυτόν τον τρόπο, γίνεται αντιληπτό ότι ο χρόνος κάθε επανάληψης του αλγορίθμου, καθορίζεται από τον χρόνο του πιο αργού, από άποψη απόδοσης, Worker κόμβου. Έτσι, ο μειωμένος χρόνος εκτέλεσης στους κόμβους που διαθέτουν επαναπρογραμματιζόμενη λογική , επικαλύπτεται από τον χρόνο εκτέλεσης του αλγορίθμου στους κόμβους που διαθέτουν μόνο κεντρική μονάδα επεξεργασίας (CPU). Για το σκοπό αυτό, θα πρέπει να δημιουργηθεί ένας νέος χρονοδρομολογητής εργασιών για το Spark, ο οποίος να γνωρίζει ακριβώς την πληροφορία για ύπαρξη ή μη, μονάδας FPGA σε κάθε κόμβο του υλοποιηθέντος συμπλέγματος. Δεδομένου αυτού, θα μπορεί να διαμοιράζει με έναν πιο «έξυπνο» τρόπο τις εργασίες που θα αναθέτει σε κάθε κόμβο, προκειμένου να βελτιστοποιείται ο συνολικός χρόνος εκτέλεσης κάθε εφαρμογής και να αξιοποιείται στο μέγιστο η ετερογένεια των Worker κόμβων. Για παράδειγμα, αν ο χρόνος εκτέλεσης στους κόμβους που δεν διαθέτουν επιταχυντή υλικού ήταν διπλάσιος από αυτόν των κόμβων που διαθέτουν, ο νέος task scheduler του Spark θα μπορούσε να στέλνει προς επεξεργασία τον διπλάσιο όγκο δεδομένων στους κόμβους με επιταχυντή υλικού πετυχαίνοντας έτσι τον ίδιο χρόνο εκτέλεσης, ανά επανάληψη, σε όλους τους Workers.

Τέλος, προκειμένου να εκμεταλλευτούμε στο μέγιστο τα οφέλη των επιταχυντών υλικού, το SPynq framework, θα μπορούσε να τροποποιηθεί ώστε να εφαρμοστεί στις εικονικές μηχανές της Amazon (F1 instances) που έχουμε ήδη αναφέρει ή ακόμα

και σε άλλες πλατφόρμες που διαθέτουν επαναπρογραμματιζόμενη λογική (FPGA) σε συνδυασμό με ισχυρούς επεξεργαστές και επικοινωνούν μεταξύ τους μέσω PCI-express. Με αυτόν τον τρόπο, θα μπορούσαμε να πετύχουμε ακόμα καλύτερους χρόνους εκτέλεσης για εφαρμογές που κάνουν χρήση επιταχυντών υλικού, ενώ παράλληλα να επωφεληθούμε και από την συνολικά μειωμένη κατανάλωση ενέργειας.

Παράρτημα

Α΄ Βιβλιοθήκες και Scripts

Σε αυτό το παράρτημα μπορείτε να βρείτε το bash script που χρησιμοποιήσαμε για την αξιολόγηση των ενσωματωμένων συστημάτων χαμηλής κατανάλωσης, καθώς επίσης και ολόκληρο το Python API που υλοποιήσαμε για τον αντίστοιχο επιταχυντή υλικού και την βιβλιοθήκη *mllib_accel* για την χρήση επιταχυντών υλικού από τις εφαρμογές του Spark.

Α΄.1 Bash script

Το bash script που χρησιμοποιήσαμε στο πρωτο μέρος της αξιολόγησης είναι το ακόλουθο. Το script αυτό λαμβάνει ένα όρισμα εισόδου το οποίο καθορίζει πόσσες φορές θα εκτελεστεί κάθε εφαρμογή.

```
#!/bin/bash

LINE="$1"
: > adv_results.txt
echo 'Logistic Regression' >> adv_results.txt
for ((i=1; i <= $1; i++)); do
  /usr/bin/time &>> logistic_tmp.txt -v ../bin/run-example ml.
    LogisticRegressionExample --regParam 0.001 --maxIter 100000000 --
    tol 1.0E-20 --elasticNetParam 0.9 ../data/mllib/sample_libsvm_data.
    txt
done

grep "Elapsed (wall clock) time (h:mm:ss or m:ss):" logistic_tmp.txt >>
    adv_results.txt
rm logistic_tmp.txt

```

```
13 echo 'Analytics - pagerank' >> adv_results.txt
14 for ((i=1; i <= $1; i++)); do
15   /usr/bin/time &>> an_pagerank_tmp.txt -v ../bin/run-example graphx.
       Analytics pagerank ../data/mllib/pagerank_data.txt --numEPart=50
16 done
17
18 grep "Elapsed (wall clock) time (h:mm:ss or m:ss):" an_pagerank_tmp.txt
       >> adv_results.txt
19 rm an_pagerank_tmp.txt
20
21 echo 'Analytics - cc' >> adv_results.txt
22 for ((i=1; i <= $1; i++)); do
23   /usr/bin/time &>> an_cc_tmp.txt -v ../bin/run-example graphx.
       Analytics cc ../data/mllib/pagerank_data.txt --numEPart=50
24 done
25
26 grep "Elapsed (wall clock) time (h:mm:ss or m:ss):" an_cc_tmp.txt >>
       adv_results.txt
27 rm an_cc_tmp.txt
28
29 echo 'Analytics - triangles' >> adv_results.txt
30 for ((i=1; i <= $1; i++)); do
31   /usr/bin/time &>> an_triangles_tmp.txt -v ../bin/run-example graphx.
       Analytics triangles ../data/mllib/pagerank_data.txt --numEPart=50
32 done
33
34 grep "Elapsed (wall clock) time (h:mm:ss or m:ss):" an_triangles_tmp.
       txt >> adv_results.txt
35 rm an_triangles_tmp.txt
36
37 echo 'KMeans' >> adv_results.txt
38 for ((i=1; i <= $1; i++)); do
39   /usr/bin/time &>> KMeans_tmp.txt -v ../bin/run-example ml.
       KMeansExample
40 done
41
42 grep "Elapsed (wall clock) time (h:mm:ss or m:ss):" KMeans_tmp.txt >>
       adv_results.txt
43 rm KMeans_tmp.txt
```

```
44
45 echo 'Linear Regression' >> adv_results.txt
46 for ((i=1; i <= $1; i++)); do
47   /usr/bin/time &>> linear_regression_tmp.txt -v ../bin/run-example ml.
       LinearRegressionExample --regParam 0.001 --tol 1.0E-20 --
       elasticNetParam 1.0 ../data/mllib/sample_linear_regression_data.txt
48 done
49
50 grep "Elapsed (wall clock) time (h:mm:ss or m:ss):"
       linear_regression_tmp.txt >> adv_results.txt
51 rm linear_regression_tmp.txt
52
53 awk -v var="$LINE" -F: '{ {if (NR % (var+1) == 1) print; else s+=60*$5+
       $6;}; if (NR % (var+1) == 0) {print s/var; s=0;}}' adv_results.txt
       > adv_results1.txt
54 mv adv_results1.txt adv_results.txt
```

Listing A′.1: Bash script for evaluating the ML and GraphX applications

A′.2 mllib_accel

Ακολούθως μπορείτε να βρείτε το Python API του επιταχυντή υλικού που χρησιμοποιήθηκε για το σενάριο χρήσης του αλγορίθμου λογιστικής παλινδρόμησης.

```
1  import cffi
2  import numpy as np
3  import os
4  import platform
5  import re
6
7  from itertools import tee
8  from math import ceil
9  from pynq import MMIO, Overlay, PL
10 from .drivers import DMA
11
12 chunkSizeMax = 5200
13 numClassesMax = 10
14 numFeaturesMax = 784
15
```

```
16 BS_SEARCH_PATH = os.path.dirname(os.path.realpath(__file__)) + "/
       overlays/"
17
18 ffi = cffi.FFI()
19
20 ffi.cdef("""
21 static uint32_t xlnkBufCnt = 0;
22 uint32_t cma_mmap(uint32_t phyAddr, uint32_t len);
23 uint32_t cma_munmap(void *buf, uint32_t len);
24 void *cma_alloc(uint32_t len, uint32_t cacheable);
25 uint32_t cma_get_phy_addr(void *buf);
26 void cma_free(void *buf);
27 uint32_t cma_pages_available();
28 """)
29
30 LIB_SEARCH_PATH = os.path.dirname(os.path.realpath(__file__)) + "/
       drivers/"
31 if platform.machine() == "x86_64":
32   # load 64bit ELF
33   libxlnk = ffi.dlopen(LIB_SEARCH_PATH + "libsds_lib64.so")
34 elif platform.machine() == "i686":
35   # load 32bit ELF
36   libxlnk = ffi.dlopen(LIB_SEARCH_PATH + "libsds_lib32.so")
37 elif platform.machine() == "armv7l":
38   # load 32bit ELF compiled for ARM
39   libxlnk = ffi.dlopen(LIB_SEARCH_PATH + "libsds_lib.so")
40 else:
41   print("Machine type not supported. Exiting!")
42   exit(1)
43
44 DMA_TO_DEV = 0     # DMA sends data to PL.
45 DMA_FROM_DEV = 1   # DMA receives data from PL.
46
47 def cma(LabeledPoints):
48
49   # ------------------------
50   #   Download Overlay.
51   # ------------------------
52
```

```
53  ol = Overlay(BS_SEARCH_PATH + "LogisticRegression.bit")
54  ol.download()
55
56  elements = []
57
58  LabeledPoints1, LabeledPoints2 = tee(LabeledPoints, 2)
59
60  numLabeledPoints = sum(1 for _ in LabeledPoints1)
61  numChunks = int(ceil(numLabeledPoints / chunkSizeMax))
62  chunkSize = int(ceil(numLabeledPoints / numChunks))
63  if bool(chunkSize & 1):
64    chunkSize += 1
65  paddingSize = (numChunks * chunkSize) - numLabeledPoints
66
67  c = 1
68
69  # ---------------------------
70  #   Allocate physically contiguous memory buffers.
71  #   Cast underlying buffers to a specific C-Type.
72  #   Get the physical address of the buffers.
73  #   Get the virtual address of the buffers.
74  # ---------------------------
75
76  data1_buf = libxlnk.cma_alloc(int(chunkSize / 2) * (numClassesMax +
     (1 + numFeaturesMax)) * 4, 1)
77  if data1_buf == ffi.NULL:
78    raise RuntimeError("Memory allocation failed.")
79  data1 = ffi.cast("float *", data1_buf)
80  data2_buf = libxlnk.cma_alloc(int(chunkSize / 2) * (numClassesMax +
     (1 + numFeaturesMax)) * 4, 1)
81  if data2_buf == ffi.NULL:
82    raise RuntimeError("Memory allocation failed.")
83  data2 = ffi.cast("float *", data2_buf)
84
85  buffers = []
86  buffers.append(int(libxlnk.cma_get_phy_addr(data1_buf)))
87  buffers.append(int(re.split("<| |>|0x", str(data1_buf))[5], 16))
88  buffers.append(int(libxlnk.cma_get_phy_addr(data2_buf)))
89  buffers.append(int(re.split("<| |>|0x", str(data2_buf))[5], 16))
```

```
90  buffers.append(chunkSize * (numClassesMax + (1 + numFeaturesMax)) *
     4)
91  buffers.append((chunkSize - paddingSize) if c == numChunks else
     chunkSize)
92  elements.append(buffers)
93
94  i = 0
95  for LabeledPoint in LabeledPoints2:
96    if i < int(chunkSize / 2):
97      data1[i * (numClassesMax + (1 + numFeaturesMax)) + int(
     LabeledPoint.label)] = 1.0
98      data1[i * (numClassesMax + (1 + numFeaturesMax)) + numClassesMax]
     = 1.0
99      f = ffi.from_buffer(LabeledPoint.features.astype(np.float32))
100     features = ffi.cast("float *", f)
101     offset = i * (numClassesMax + (1 + numFeaturesMax)) +
     numClassesMax + 1
102     data1[offset:offset + len(LabeledPoint.features)] = features[0:
     len(LabeledPoint.features)]
103   else:
104     data2[(i - int(chunkSize / 2)) * (numClassesMax + (1 +
     numFeaturesMax)) + int(LabeledPoint.label)] = 1.0
105     data2[(i - int(chunkSize / 2)) * (numClassesMax + (1 +
     numFeaturesMax)) + numClassesMax] = 1.0
106     f = ffi.from_buffer(LabeledPoint.features.astype(np.float32))
107     features = ffi.cast("float *", f)
108     offset = (i - int(chunkSize / 2)) * (numClassesMax + (1 +
     numFeaturesMax)) + numClassesMax + 1
109     data2[offset:offset + len(LabeledPoint.features)] = features[0:
     len(LabeledPoint.features)]
110
111   i += 1
112   if i == chunkSize:
113     c += 1
114     if c <= numChunks:
115
116     # ------------------------
117     #   Allocate physically contiguous memory buffers.
118     #   Cast underlying buffers to a specific C-Type.
```

```
119       #   Get the physical address of the buffers.
120       #   Get the virtual address of the buffers.
121       # ------------------------
122
123       data1_buf = libxlnk.cma_alloc(int(chunkSize / 2) * (numClassesMax
      + (1 + numFeaturesMax)) * 4, 1)
124       if data1_buf == ffi.NULL:
125         raise RuntimeError("Memory allocation failed.")
126       data1 = ffi.cast("float *", data1_buf)
127       data2_buf = libxlnk.cma_alloc(int(chunkSize / 2) * (numClassesMax
      + (1 + numFeaturesMax)) * 4, 1)
128       if data2_buf == ffi.NULL:
129         raise RuntimeError("Memory allocation failed.")
130       data2 = ffi.cast("float *", data2_buf)
131
132       buffers = []
133       buffers.append(int(libxlnk.cma_get_phy_addr(data1_buf)))
134       buffers.append(int(re.split("<| |>|0x", str(data1_buf))[5], 16))
135       buffers.append(int(libxlnk.cma_get_phy_addr(data2_buf)))
136       buffers.append(int(re.split("<| |>|0x", str(data2_buf))[5], 16))
137       buffers.append(chunkSize * (numClassesMax + (1 + numFeaturesMax))
      * 4)
138       buffers.append((chunkSize - paddingSize) if c == numChunks else
     chunkSize)
139       elements.append(buffers)
140
141       i = 0
142
143   return elements
144
145 def gradients_kernel_accel(buffers, weights):
146   chunkSize = int(buffers[4] / (numClassesMax + (1 + numFeaturesMax)) /
        4)
147   numClasses = len(weights)
148   numFeatures = len(weights[0]) - 1
149
150   # ------------------------
151   #   Physical address of the Accelerator Adapter IP.
152   # ------------------------
```

```
153
154 ADDR_Accelerator_Adapter_BASE = int(PL.ip_dict["
    SEG_LR_gradients_kernel_accel_0_if_Reg"][0], 16)
155 ADDR_Accelerator_Adapter_RANGE = int(PL.ip_dict["
    SEG_LR_gradients_kernel_accel_0_if_Reg"][1], 16)
156
157 # ----------------------------
158 #     Initialize new MMIO object.
159 # ----------------------------
160
161 bus = MMIO(ADDR_Accelerator_Adapter_BASE,
    ADDR_Accelerator_Adapter_RANGE)
162
163 # ----------------------------
164 #   Physical addresses of the DMA IPs.
165 # ----------------------------
166
167 ADDR_DMA0_BASE = int(PL.ip_dict["SEG_dm_0_Reg"][0], 16)
168 ADDR_DMA1_BASE = int(PL.ip_dict["SEG_dm_1_Reg"][0], 16)
169 ADDR_DMA2_BASE = int(PL.ip_dict["SEG_dm_2_Reg"][0], 16)
170 ADDR_DMA3_BASE = int(PL.ip_dict["SEG_dm_3_Reg"][0], 16)
171
172 # ----------------------------
173 #     Initialize new DMA objects.
174 # ----------------------------
175
176 dma0 = DMA(ADDR_DMA0_BASE, direction = DMA_TO_DEV)     # data1 DMA
177 dma1 = DMA(ADDR_DMA1_BASE, direction = DMA_TO_DEV)     # data2 DMA
178 dma2 = DMA(ADDR_DMA2_BASE, direction = DMA_TO_DEV)     # weights DMA
179 dma3 = DMA(ADDR_DMA3_BASE, direction = DMA_FROM_DEV)   # gradients DMA
180
181 # ----------------------------
182 #     Assign/Allocate physically contiguous memory buffers.
183 # ----------------------------
184
185 dma0._bufPtr = ffi.cast("uint32_t *", buffers[0])
186 dma0.buf = ffi.cast("void *", buffers[1])
187 dma0.bufLength = int(buffers[4] / 2)
188 dma1._bufPtr = ffi.cast("uint32_t *", buffers[2])
```

```
189 dma1.buf = ffi.cast("void *", buffers[3])
190 dma1.bufLength = int(buffers[4] / 2)
191 dma2.create_buf((numClassesMax * (1 + numFeaturesMax)) * 4)
192 dma3.create_buf((numClassesMax * (1 + numFeaturesMax)) * 4)
193
194 # ----------------------------
195 #    Get CFFI pointers to objects' internal buffers.
196 # ----------------------------
197
198 weights_buf = dma2.get_buf(data_type = "float")
199 gradients_buf = dma3.get_buf(data_type = "float")
200
201 w = ffi.from_buffer(np.pad(weights, ((0, 0), (0, numFeaturesMax -
      numFeatures)), "constant").flatten().astype(np.float32))
202 w_buf = ffi.cast("float *", w)
203 weights_buf[0:numClasses * (1 + numFeaturesMax)] = w_buf[0:numClasses
      * (1 + numFeaturesMax)]
204
205 # ----------------------------
206 #   Write data to MMIO.
207 # ----------------------------
208
209 CMD = 0x0028              # Command.
210 ISCALAR0_DATA = 0x0080  # Input Scalar-0 Write Data FIFO.
211
212 bus.write(ISCALAR0_DATA, chunkSize)
213 bus.write(CMD, 0x00010001)
214 bus.write(CMD, 0x00020000)
215 bus.write(CMD, 0x00000107)
216
217 # ----------------------------
218 #   Transfer data using DMAs (Non-blocking).
219 #   Block while DMAs are busy.
220 # ----------------------------
221
222 dma0.transfer(int(buffers[4] / 2), direction = DMA_TO_DEV)
223 dma1.transfer(int(buffers[4] / 2), direction = DMA_TO_DEV)
224 dma2.transfer((numClassesMax * (1 + numFeaturesMax)) * 4, direction =
      DMA_TO_DEV)
```

```
225
226   dma0.wait()
227   dma1.wait()
228   dma2.wait()
229
230   dma3.transfer((numClassesMax * (1 + numFeaturesMax)) * 4, direction =
        DMA_FROM_DEV)
231
232   dma3.wait()
233
234   g_buf = ffi.buffer(gradients_buf, (numClasses * (1 + numFeaturesMax))
        * 4)
235   g = np.frombuffer(g_buf, dtype = np.float32)
236   gradients = np.copy(np.reshape(g, (numClasses, 1 + numFeaturesMax))
      [:, :1 + numFeatures])
237
238   # ---------------------------
239   #   Destructors for DMA objects.
240   # ---------------------------
241
242   dma0.buf = None
243   dma0._bufPtr = None
244   dma0.bufLength = None
245   dma1.buf = None
246   dma1._bufPtr = None
247   dma1.bufLength = None
248
249   dma0.__del__()
250   dma1.__del__()
251   dma2.__del__()
252   dma3.__del__()
253
254   return gradients
255
256 def cmf(buffers):
257
258   # ---------------------------
259   #   Free previously allocated buffers.
260   # ---------------------------
```

```
261
262   data1_buf = ffi.cast("void *", buffers[1])
263   data2_buf = ffi.cast("void *", buffers[3])
264   libxlnk.cma_free(data1_buf)
265   libxlnk.cma_free(data2_buf)
266
267   return 0
```

Listing A′.2: Python API (LogisticRegression.py) of the logistic regression hardware accelerator

Ακολούθως μπορείτε να βρείτε το νέο *classification.py* αρχείο που υλοποιήθηκε.

```
1  import numpy as np
2
3  from math import exp
4  from pyspark import RDD
5  from time import time
6  from .accelerators.LogisticRegression import cma,
       gradients_kernel_accel, cmf
7
8  __all__ = ['LogisticRegression']
9
10 class LogisticRegression(object):
11   """
12   Multiclass Logistic Regression Model.
13
14     :param numClasses:      Number of possible outcomes.
15
16     :param numFeatures:     Dimension of the features.
17
18     :param weights:         Weights computed for every feature.
19     (The intercepts will be part of the weights.)
20   """
21
22   def __init__(self, numClasses, numFeatures, weights = None):
23     self.numClasses = numClasses
24     self.numFeatures = numFeatures
25     self.weights = weights
26
```

```
27  def train(self, trainRDD, alpha = 1.0, numIterations = 100, _accel_ =
      0):
28    """
29    Train a logistic regression model on the given data.
30
31      :param trainRDD:         The training data, an RDD of
    LabeledPoint.
32
33      :param alpha:            The learning rate (default: 1.0).
34
35      :param numIterations:    The number of iterations (default: 100).
36
37      :param _accel_:          0: SW-only, 1: HW accelerated (deafult:
    0).
38
39      :note:                   Labels used in logistic regression
    should be
40                   {0, 1, ..., k - 1} for k classes classification
    problem.
41    """
42
43    def gradients_kernel(data):
44      # Compute the gradients given the (label, features) pair of a
    single data point.
45      gradients = np.zeros((self.numClasses, 1 + self.numFeatures))
46
47      for k in range (0, self.numClasses):
48        # margin (rawPrediction)
49        margin = self.weights[k][0] + np.dot(data.features, self.
    weights[k][1:])
50
51        # score (probability)
52        if margin < 0:
53          score = 1.0 - 1.0 / (1.0 + exp(margin))
54        else:
55          score = 1.0 / (1.0 + exp(-margin))
56
57        multiplier = score - (1.0 if k == int(data.label) else 0.0)
58
```

```
59          gradients[k][0] = multiplier
60          gradients[k][1:] = multiplier * data.features
61
62        return gradients
63
64      # Reduction of multiclass classification to binary classification.
65      # Performs reduction using one against all strategy (OneVsRest).
66      # For a multiclass classification with k classes, train k models (
    one per class).
67      print("    * LogisticRegression Training *")
68
69      if _accel_:
70        trainRDD = trainRDD.mapPartitions(cma).cache()
71        numExamples = trainRDD.map(lambda buffers: buffers[5]).sum()
72      else:
73        trainRDD = trainRDD.cache()
74        numExamples = trainRDD.count()
75
76      print("      # numExamples:              {:d}".format(numExamples))
77      print("      # numClasses:               {:d}".format(self.
    numClasses))
78      print("      # numFeatures:              {:d}".format(self.
    numFeatures))
79      print("      # alpha:                    {:g}".format(alpha))
80      print("      # numIterations:            {:d}".format(numIterations)
    )
81
82      start = time()
83
84      # Run Batch Gradient Descent (BGD) in parallel.
85      # Averaging the subgradients over different partitions is
86      # performed using one standard spark map-reduce in each iteration.
87      if self.weights is None:
88        self.weights = np.zeros((self.numClasses, 1 + self.numFeatures))
89
90      # Momentum Optimization.
91      gamma = 0.9
92      velocity = np.zeros_like(self.weights)
93
```

```
94      for t in range(0, numIterations):
95        print("{:3d}% |{:35s}| {:d}/{:d}".format(int((t / numIterations)
        * 100), u"\u25A5" * int((t / numIterations) * 35), t, numIterations
        ), end = "\r")
96
97        if _accel_:
98          gradients = trainRDD.map(lambda buffers: gradients_kernel_accel
        (buffers, self.weights)).reduce(lambda a, b: np.add(a, b))
99        else:
100         gradients = trainRDD.map(gradients_kernel).reduce(lambda a, b:
        np.add(a, b))
101
102       velocity = np.add(gamma * velocity, (alpha / numExamples) *
        gradients)
103       self.weights = np.subtract(self.weights, velocity)
104
105     end = time()
106     print("{:3d}% |{:35s}| {:d}/{:d} Time: {:.3f} sec".format(100, u"\
        u25A5" * 35, numIterations, numIterations, end - start))
107
108     if _accel_:
109       trainRDD.map(cmf).collect()
110
111     return self
112
113   def test(self, testRDD):
114     """
115     Test a logistic regression model on the given data.
116
117       :param testRDD:    The testing data, an RDD of LabeledPoint.
118
119       :note:             Labels used in logistic regression should be
120                          {0, 1, ..., k - 1} for k classes classification
        problem.
121     """
122
123     # Each example is scored against all k models and the model with
        highest score
124     # is picked to label the example.
```

```
125     print("    * LogisticRegression Testing *")
126
127     true = testRDD.map(lambda data: 1 if data.label == self.predict(
    data.features) else 0).reduce(lambda a, b: a + b)
128     false = testRDD.count() - true
129
130     print("     # accuracy:                  {:.3f} ({:d}/{:d})".format(
    true / (true + false), true, true + false))
131     print("     # true:                      {:d}".format(true))
132     print("     # false:                     {:d}".format(false))
133
134   def predict(self, features):
135     """
136     Predict values for a single data point using the model trained.
137
138       :param features:    Features to be labeled.
139     """
140
141     # Compute and find the one with maximum margins.
142     maxMargin = -np.inf
143     bestClass = -1
144
145     for k in range (0, self.numClasses):
146       # margin (rawPrediction)
147       margin = self.weights[k][0] + np.dot(features, self.weights[k
    ][1:])
148
149       if margin > maxMargin:
150         maxMargin = margin
151         bestClass = k
152
153     return bestClass
154
155   def save(self, path):
156     """
157     Save this model to the given path.
158     """
159
160     np.savetxt(path, self.weights)
```

```
161
162   def load(self, path):
163     """
164     Load a model from the given path.
165     """
166
167     self.weights = np.loadtxt(path)
```

Listing Α΄.3: New Spark library (classification.py) that invokes the hardware accelerator Python API

Η εφαρμογή που χρησιμοποιήσαμε ως παράδειγμα για την εκτέλεση του αλγορίθμου λογιστικής παλινδρόμησης που κάνει χρήση του επιταχυντή υλικού, παρατίθεται στη συνέχεια:

```
1  from pyspark import SparkContext
2  from pyspark.mllib.regression import LabeledPoint
3  from pyspark.mllib_accel.classification import LogisticRegression
4  from sys import argv
5  from time import time
6
7    def parsePoint(line):
8      """
9      Parse a line of text into an MLlib LabeledPoint object.
10     """
11
12     data = [float(s) for s in line.split(',')]
13
14     return LabeledPoint(data[0], data[1:])
15
16 if __name__ == "__main__":
17
18   if len(argv) != 7:
19     print("Usage: LogisticRegressionApp <dataset> <fraction> <
      numPartitions> <alpha> <numIterations> <_accel_>")
20     exit(-1)
21
22   dataset = argv[1]
23   fraction = float(argv[2])
24   numPartitions = int(argv[3])
```

```
25  alpha = float(argv[4])
26  numIterations = int(argv[5])
27  _accel_ = int(argv[6])
28
29  train_file = "inputs/" + dataset + "_train.dat"
30  test_file = "inputs/" + dataset + "_test.dat"
31
32  with open(dataset, 'r') as f:
33    for line in f:
34      if line[0] != '#':
35        parameters = line.split(',')
36        numClasses = int(parameters[0])
37        numFeatures = int(parameters[1])
38  f.close()
39
40  sc = SparkContext(appName = "Python Logistic Regression on " +
      dataset)
41
42  print("* LogisticRegression Application *")
43  print(" # train file:                {:s}".format(train_file))
44  print(" # fraction:                  {:g}".format(fraction))
45  print(" # test file:                 {:s}".format(test_file))
46  print(" # numPartitions:             {:d}".format(numPartitions))
47
48  trainRDD = sc.textFile(train_file, numPartitions).sample(False,
      fraction).map(parsePoint)
49  testRDD = sc.textFile(test_file, numPartitions).map(parsePoint)
50
51  start = time()
52
53  # Train a logistic regression model given an RDD of (label, features)
      pairs.
54  # We run a fixed number of iterations of Gradient Descent using the
      specified alpha.
55  # We use the entire data set to update the gradient in each iteration
      (Batch GD).
56  LR = LogisticRegression(numClasses, numFeatures).train(trainRDD,
      alpha, numIterations, _accel_)
57
```

```
58  end = time()
59  if _accel_:
60    print("! Time running LogisticRegression train in hardware: {:.3f}
     sec".format(end - start))
61  else:
62    print("! Time running LogisticRegression train in software: {:.3f}
     sec".format(end - start))
63
64  LR.save("outputs/weights.out")
65
66  LR.test(testRDD)
67
68  sc.stop()
```

Listing A΄.4: Logistic Regression example application

Τέλος, παρατίθεται το *lr.sh* bash script που χρησιμοποιήθηκε για την εκτέλεση της προαναφερθείσας εφαρμογής:

```
1 #!/bin/bash
2
3 # $1 = _accel_ and $2 = dataset, or just $1 = dataset
4 if [ "$1" == "_accel_" ]
5 then
6   time spark-submit LogisticRegressionApp.py $2 1.0 4 0.75 100 1
7 else
8   time spark-submit LogisticRegressionApp.py $1 1.0 4 0.75 100 0
9 fi
```

Listing A΄.5: Bash script for executing the Logistic Regression example

Βιβλιογραφία

[1] **FPGA acceleration of Spark applications in a Pynq cluster**, Christofors Kachris, Elias Koromilas, Ioannis Stamelos, Dimitrios Soudris
FPL 2017

[2] **Big Data**,
https://en.wikipedia.org/wiki/Big_data, http://semeon.com/blog/big-data-ad-agencies-giving-reason-to-gut-decisions/

[3] **The Network Impact of Big Data**,
http://www.nojitter.com/post/240170228/the-network-impact-of-big-data

[4] **Cluster computing**, Lubna Luxmi Chowdhry, 20 Sep 2005
https://www.codeproject.com/Articles/11709/Cluster-Computing

[5] **Cluster**,
https://techterms.com/definition/cluster

[6] **Amdahl's Model for Scaling Efficiency**,
https://en.wikipedia.org/wiki/Amdahl%27s_law

[7] **Embedded Systems**,
https://en.wikipedia.org/wiki/Embedded_system,
http://internetofthingsagenda.techtarget.com/definition/embeddedsystem

[8] **Europe embedded system market size**,
https://www.gminsights.com/industry-analysis/embedded-system-market

[9] **AMD pins future growth to embedded marketplace**,
https://www.theregister.co.uk/2013/04/23/amd_gseries_embedded_chips

[10] **FPGAs**,
https://en.wikipedia.org/wiki/Field-programmable_gate_array

[11] **FPGAs Advantages**,
http://community.brocade.com/t5/Service-Providers/FPGA-or-ASIC-Pro-s-amp-Con-s-of-Each-Technology/ba-p/709

[12] **FPGAs in the Datacenter**,
https://www.forbes.com/sites/moorinsights/2016/11/14/xilinx-seeks-to-mainstream-fpgas-in-the-datacenter/

[13] **Xilinx All Programmable Devices**,
https://www.xilinx.com/support/documentation/white_papers/wp492-compute-intensive-sys.pdf

[14] **Amazon F1 Instances**,
https://aws.amazon.com/ec2/instance-types/f1/

[15] **Heterogeneous Computer Architectures**,
https://www.iti.uni-stuttgart.de/abteilungen/rechnerarchitektur/projekte/heterogeneous-computing.html

[16] **Heterogeneous Computing**,
https://en.wikipedia.org/wiki/Heterogeneous_ computing

[17] **Apache Spark - Databricks**,
https://databricks.com/spark/about

[18] **Apache Spark**,
https://spark.apache.org/

[19] **Hadoop**,
http://hadoop.apache.org/

[20] **Cassandra**,
http://cassandra.apache.org/

[21] **Safari Books - Apache Spark**,
https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/ch01.html

[22] **Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing**,Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
https://cs.stanford.edu/ matei/papers/2012/nsdi_ spark.pdf

[23] **Jacek Laskowski - Mastering Apache Spark 2**,
https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details

[24] **Architecture of Spark SQL**,
https://www.packtpub.com/mapt/book/big_ data_ and_ business_ intelligence/9781785884696/4/c of-spark-sql

[25] **Spark Cluster Managers**,
http://www.agildata.com/apache-spark-cluster-managers-yarn-mesos-or-standalone/

[26] **Cisco Visual Networking Index**, Global Mobile Data Traffic Forecast Update 20142019 White Paper

[27] **Power Challenges May End the Multicore Era**, Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger
http://www.cc.gatech.edu/ hadi/doc/paper/2013-cacm-dark_ silicon.pdf

[28] **Raspberry Pi - Wikipedia**,
https://en.wikipedia.org/wiki/Raspberry_Pi

[29] **MagPi - Pi 3 Specs, Benchmarks and More**,
https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/

[30] **DragonBoard 410c - Qualcomm**,
https://developer.qualcomm.com/hardware/dragonboard-410c

[31] **Pynq: PYTHON PRODUCTIVITY FOR ZYNQ**,
http://www.pynq.io/

[32] **Linear Regression - Wikipedia**,
https://en.wikipedia.org/wiki/Linear_regression

[33] **Logistic Regression - Wikipedia**,
https://en.wikipedia.org/wiki/Logistic_regression

[34] **K-means Clustering - Wikipedia**,
https://en.wikipedia.org/wiki/K-means_clustering

[35] **PageRank - Wikipedia**,
https://en.wikipedia.org/wiki/PageRank

[36] **Intel Acquires Altera**,
https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/

[37] **Zynq-7000 All Programmable SoC**,
https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[38] **Python in Big Data**,
https://www.continuum.io/why-python

[39] **PYNQ - GitHub**,
https://github.com/Xilinx/PYNQ

[40] **PYNQ - Quickstart Guide**,
http://pynq.readthedocs.io/en/latest/1_getting_started.html

[41] **Serialization - Wikipedia**,
https://en.wikipedia.org/wiki/Serialization

[42] **FPGA-Acceleration of Machine Learning in Cloud Computing, a case study using Logistic Regression**,Elias Koromilas
Master's Thesis, National Technical University of Athens, 2017

[43] **Performance and Energy evaluation of Spark applications on low-power SoCs**, Christofors Kachris, Ioannis Stamelos, Dimitrios Soudris
http://ieeexplore.ieee.org/document/7818362/

[44] **SPynq: Acceleration of Machine Learning Applications over Spark on Pynq**, Christofors Kachris, Elias Koromilas, Ioannis Stamelos, Dimitrios Soudris
SAMOS 2017

Αγγλικό Κείμενο - English Version

Acknowledgments

I would like to thank the rapporteur and supervisor Professor Soudris Dimitrios for the help and support he has provided me for the preparation of this thesis. I would also like to thank Dr Kachris Christoforos for his support and guidance, particularly in the technical part and in the implementation of this work. For their participation in the three-member committee, I would like to thank Dr Pekmetzi Kiamal and Dr Georgios Goumas. I would also like to thank my fellow students and especially my friends for being an integral part of my life all these years. Without them a lot would be different. Furthermore, I would like to wholeheartedly thank my family for their love and the uncompromising support and understanding that they provided me throughout my studies. Finally, I would like to dedicate this work to memory of my grandfathers.

Contents

List of Figures

List of Tables

Listings

Abstract

Emerging web applications like big data analytics have significantly increased the workload on the data centers during the last years. In 2015, the total network traffic of the data centers was around 4.7 Exabytes and it is estimated that by the end of 2018 it will cross the 8.5 Exabytes mark. The growing demands both in performance and energy efficiency, have led companies into charting new paths for developing energy-efficient platforms for heterogeneous datacenters, therefore they recently started deploying FPGA accelerators and further offloading part of the workload to embedded processors (i.e. ARM processors) at a datacenter scale.

For this reason we are going to first map Apache Spark, a widely used, fault-tolerant and general-purpose cluster computing framework on several embedded systems including Raspberry Pi 3, DragonBoard 410c and PYNQ-Z1. We present the whole procedure of mapping and deploying Spark on the embedded devices along with any necessary configurations.

Subsequently, we are going to create a heterogeneous cluster consisting of four PYNQ-Z1 nodes and a typical Intel based one. Next on, we will go through all the necessary steps and configurations for deploying Spark on the implemented cluster. Then, a proposed framework for the seamless utilization of hardware accelerators for Spark applications will be presented, as well as a set of libraries to hide the accelerator's low-level details, simplifying in this way the incorporation of hardware accelerators in Spark.

In the last part of the thesis, we are going to first explore the capabilities of the embedded platforms we used, by taking execution metrics using a set of typical machine learning and graph processing algorithms and further comparing the performance and energy efficiency of each system with a mainstream powerful server. Finally, the proposed framework is evaluated in a machine learning application for a use case scenario on logistic regression. The overall evaluation shows that in general the execution time on embedded systems is 6.2x to 13x higher compared to a

typical datacenter server but the embedded platforms are 2x - 3.5x better in terms of energy efficiency. On the other hand, the proposed framework for the utilization of hardware accelerators in Spark shows that PYNQ's heterogeneous accelerator-based ZYNQ MPSoC, can achieve up to 2x system speedup compared to a Xeon system and 18x better energy-efficiency. Especially for embedded applications, the proposed framework can achieve up to 36x speedup compared to the software only implementation on low-power embedded processors (ARM processors) and 29x lower energy consumption.

Keywords: Apache Spark, machine learning, FPGA accelerators, big data analytics, embedded systems, Raspberry Pi 3, DragonBoard 410c, PYNQ-Z1

Publications

Parts of this thesis have been published in the following conferences:

- **Performance and Energy evaluation of Spark applications on low-power SoCs**, Christoforos Kachris, Ioannis Stamelos, Dimitrios Soudris
  International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), 2016

- **Spark acceleration on FPGAs: A use case on machine learning in Pynq**, Elias Koromilas, Ioannis Stamelos, Christoforos Kachris, Dimitrios Soudris
  International Conference on Modern Circuits and Systems Technologies (MOCAST), 2017

- **FPGA acceleration of Spark applications in a Pynq cluster**, Christoforos Kachris, Elias Koromilas, Ioannis Stamelos, Dimitrios Soudris
  International Conference in Field-Programmable Logic and Applications (FPL), 2017

- **SPynq: Acceleration of Machine Learning Applications over Spark on Pynq**, Christoforos Kachris, Elias Koromilas, Ioannis Stamelos, Dimitrios Soudris
  International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), 2017

1 Introduction

1.1 Modern Systems and Applications

As the years go by, every science and specialty uses technology to move further on accomplishing new goals and targets. From farmers to economy analysts and computer programmers, they all use systems to help them do their jobs faster and more efficiently.

In that way, there is a growing demand on embedding sensors, actuators and other electronics in almost every system, so that the latter is able to collect and exchange data, thus being able to perform tasks and communicate with other devices over a network. And this is where Internet comes in. The Internet has changed the way we do business, and the way we communicate. Modern Systems use the Internet to communicate with other devices and distribute the processing load to other systems in the cloud. The Internet is also used as a big database that can provide information for almost any topic. If we take into consideration all of the above, emerging applications like big data analytics and IoTs require powerful systems that can process large amounts of data without consuming high power. These emerging applications, require fast time-to-market and reduced development times. To address the large processing requirements of emerging applications, novel architectures are required in the domain of high-performance and energy-efficient processors[1].

1.2 Big Data

As shown in the previous section, modern systems and applications either produce or need a lot of data as input, a fact that led us to define a new term, big data.

Big data is a term for data sets that are so large or complex that traditional data processing application software is inadequate to deal with them [2].

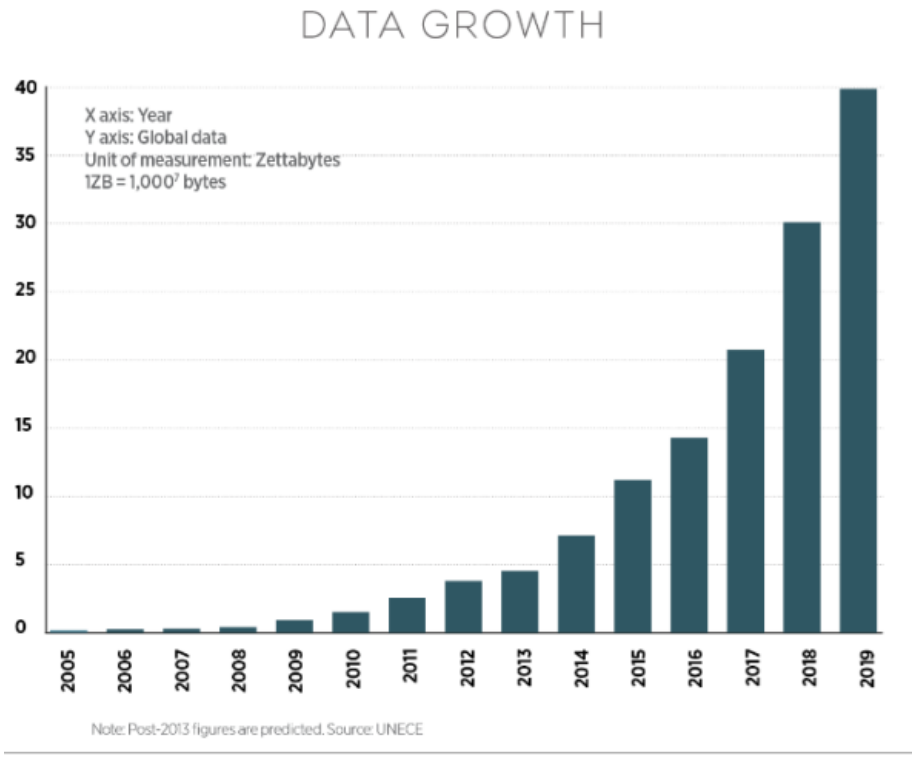


Figure 1.1: Big Data [3]

There are many challenges including capture, storage, analysis, data curation, search, sharing, transfer, visualization, querying, updating and information privacy. We also use the term ”big data” to refer to the data or the data sets extracted from predictive, behavior and other advanced data analytics methods. Data-sets, grow rapidly since there is a plethora of information-sensing Internet of Things devices (e.g mobile and wearable devices, software logs, cameras etc.). Through the analysis of these data, we can find new correlations in things, we can make predictions and spot trends and even prevent diseases and find solutions for many - until now - unsolved problems.

But the huge volume of these data, makes it difficult to handle and process them. Such actions may require massively parallel software running on up to thousands of servers.

1.3 Cluster Computing

As a solution to the previous problem of handling and processing this huge amount of data, there has been a shift to cluster computing. A cluster refers to a group of machines that work as a unified system through software and networking [4].

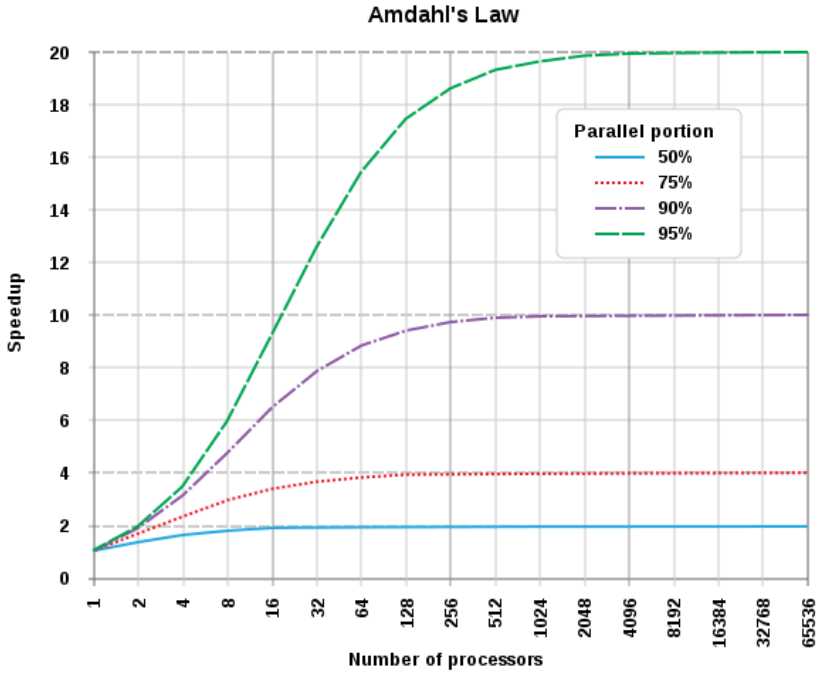


Figure 1.2: Amdahl's model for scaling efficiency [6]

Computer clusters can range from two machines to hundreds of connected computers. There are basically two main purposes of using clusters. The first one has to do with High Availability (HA) for greater reliability and the second one with High Performance Computing (HPC), for eras where greater computational power is needed. For example, small clusters can be used to improve the performance of web by handling multiple incoming requests in parallel. On the other hand, large clusters can be used to perform scientific calculations or to run a large number of complex algorithms [5]. As high-performance computing (HPC) clusters grow in size, there are a lot of new challenges, including finding solutions to the increasing systems as well as minimizing the power demands and the overall power consumption (including the consumption of the servers as well as the power consumption for cooling etc.).

1.4 Embedded Systems

The term *embedded system* refers to a computer system which has a specific function and is placed within a larger mechanical or electrical system, often with real-time computing constraints [7].

Nowadays, in a world of "smart" - so called - devices, embedded systems can be found everywhere. Industrial machines, agricultural and process industry devices, automobiles, medical equipment, cameras, household appliances, airplanes, vending machines and toys as well as mobile devices are all possible locations for an embedded system [7]. They are characterized by their small size, low power consumption and their low per-unit cost. They are also characterized by limited processing resources, a fact that makes programming and interacting with them a challenging procedure. Computer programmers and design engineers have to optimize both software and hardware in order to achieve at the same time low power consumption and high performance.

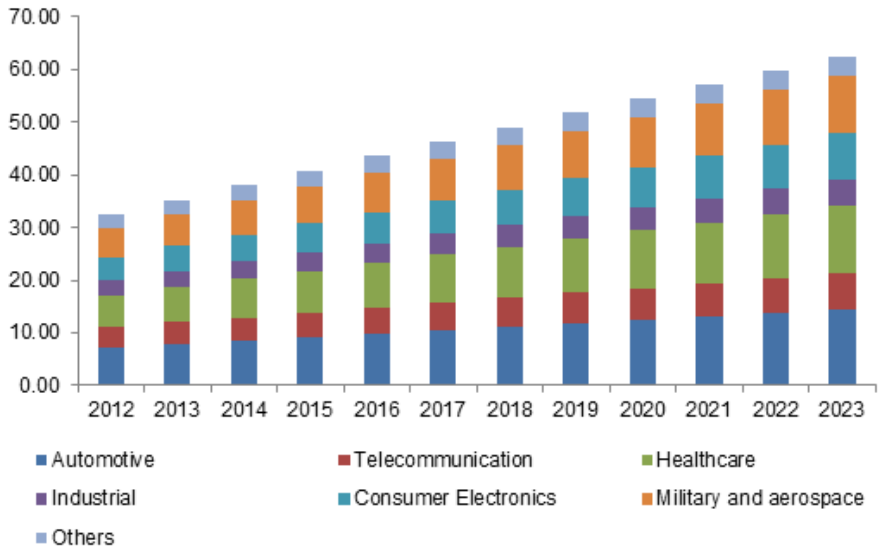


Figure 1.3: Europe embedded system market size, by application, 2012-2023 (USD Billion)[8]

Another good thing about embedded systems, is that they can scale up very much, thus creating a cluster which is consisted by thousands of systems. In that way the profits of the cluster computing can be combined with the benefits of embedded systems' low power consumption.

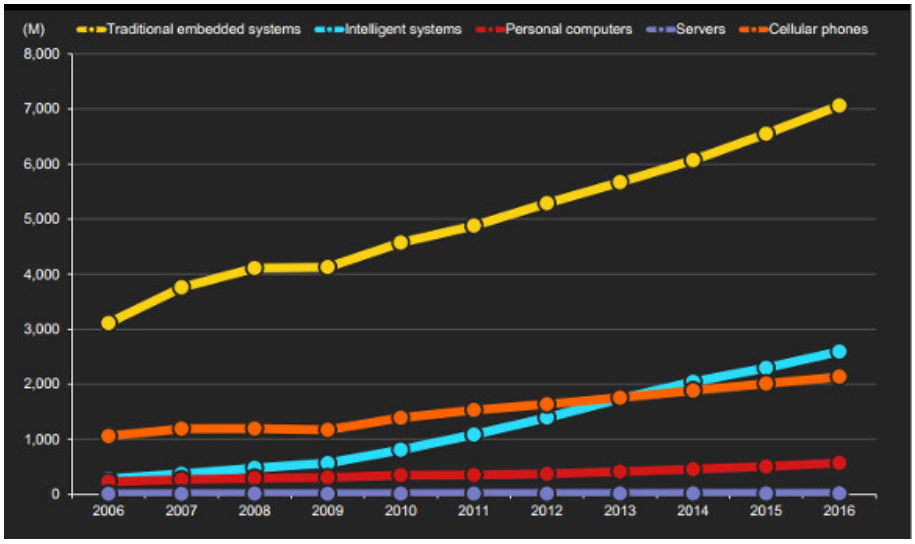


Figure 1.4: AMD pins future growth to embedded marketplace[9]

1.5 FPGAs in the Data Center

FPGA stands for field-programmable gate array and is basically an integrated circuit designed to be configured by a customer or a designer after manufacturing [10]. The configuration of an FPGA is usually specified using a hardware description language (HDL). From its name, it can be seen that an FPGA contains an array of programmable logic blocks and a hierarchy of reconfigurable interconnects that allow the blocks to be ”wired together”, like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. The greatest advantage of FPGAs is their flexibility. Through HDL, one can rapidly implement or reprogram the logic of an FPGA [11]. The HDL code is simply downloaded to the FPGA as a bit-stream code and then the FPGA's logic is created. Since the code written by FPGA programmers is translated to hardware logic, another advantage of the FPGAs is the very high performance thus maintaining a very low power consumption.

Today FPGAS are used for processing of automotive sensor data, network accelerators, embedded industrial applications and other tasks where high performance and energy efficiency are required [12], while they are going to be ideal for tomorrow's systems that require greater compute capabilities to support an expanding set of workloads and their underlying evolving algorithms. like Cloud Data Centers,

autonomous vehicles etc. For example, big data analytics, machine learning, vision processing, genomics, and advanced driver assistance systems (ADAS) sensor fusion workloads are all pushing compute boundaries beyond what existing systems (e.g., x86 based systems) can deliver in a cost effective and efficient manner [13]. That is why a lot of companies like Amazon have already started deploying FPGAs at a data-center scale and selling services based on FPGA computing[14].

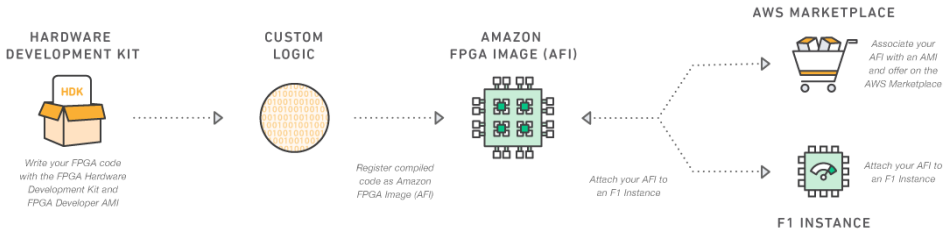


Figure 1.5: Amazon F1 instances - How it works[14]

1.6 Thesis Aim

The aim of this thesis is to explore the capabilities of embedded platforms and processors, by taking execution metrics using a set of typical machine learning and graph processing algorithms and further on comparing each system for its overall performance and the power it draws. Next on, there will be an attempt to combine the use of CPU cores along with the use of FPGAs and assess the benefits or the drawbacks of such an heterogeneous platform approach.

So in more detail, the aim is to collect feedback and provide some concrete proof or evidence on whether embedded devices and processors could establish a key position in the high performance computing or even at fields where the energy consumption is the main limiting factor. The results from this evaluation will provide information about performance, energy consumption and even scaling capabilities. Such factors are of critical importance for companies and in general for the market, since they could be used to define different profiles of possible setups (e.g low power consumption, greater scaling capabilities etc.). Each one separately, either the term ”one” stands for a person or a company, could then use the profiles he generated to finally decide whether a shift to this direction would be an ideal move and whether he would be benefited.

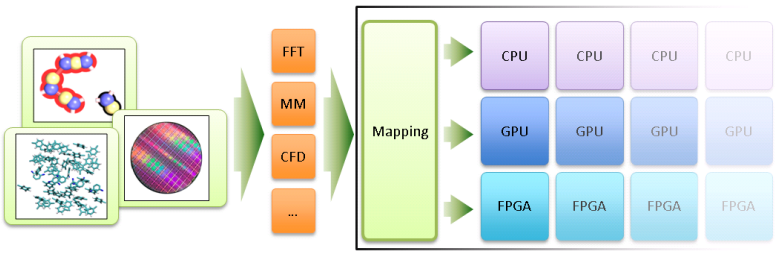


Figure 1.6: Example of heterogeneous platform architecture[15].

As far as heterogeneous computing is concerned, it goes without saying that it is a rather new field that houses a lot of challenges for the computing community. The presence of multiple processing elements raises all of the issues involved with homogeneous parallel processing systems, while the level of heterogeneity in the system can introduce non-uniformity in system development, programming practices, and overall system capability [16]. For that reason, there will be an attempt to create an API for the seamless utilization of hardware accelerators that can be used both for embedded systems and high-performance applications like cloud, edge and fog computing. In the end, performance metrics will be taken by comparing the execution time both when using only the CPU cores and when invoking the hardware accelerators.

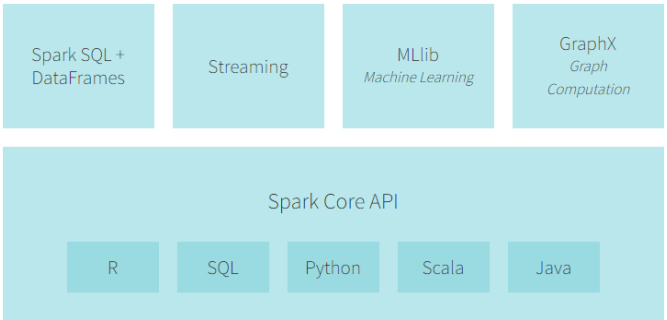
Finally, it is important to note that Apache Spark is going to be used for both the evaluation of the low power SoC-based processors (that are used in embedded systems) and the latter approach on heterogeneous platforms. Spark is one of the most widely used frameworks in cloud computing that comes with a bunch of built-in libraries and applications (including machine learning and graph processing ones) and on which the execution metrics will be taken. The proposed framework for the seamless utilization on the hardware accelerators is also based on the Spark framework so in the next chapter we are going to take a closer look at the functionality of Apache Spark as well as at the key features it provides.

2 Apache Spark

2.1 Overview

Apache Spark is a powerful 100% open source processing engine built around speed, ease of use, and sophisticated analytics. It was originally developed at UC Berkeley in 2009 [17]. In more detail, Spark is a fast fault-tolerant and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming[18]. It also integrates closely with other Big Data tools. In particular, Spark can run in Hadoop ([19]) clusters and access any Hadoop data source, including Cassandra [20] .

(α′) Apache Spark logo



(β′) Apache Spark Ecosystem[17]

Figure 2.1: Apache Spark

Since its release, Apache Spark has seen rapid adoption by enterprises across a wide range of industries. Internet powerhouses such as Netflix, Yahoo, and eBay have deployed Spark at massive scale, collectively processing multiple petabytes of

data on clusters of over 8,000 nodes. It has quickly become the largest open source community in big data, with over 1000 contributors from 250+ organizations. is the largest open source project in data processing [17].

2.2 Benefits

When it comes to the benefits of Apache Spark, they can be divided into three major categories: *speed*, *ease of use* and *a unified engine*.

2.2.1 Speed

Speed is a very important factor when it comes for big data analytics, where the processing of large datasets means the difference between exploring data interactively and waiting minutes or hours. Spark is engineered from the bottom-up for performance and is characterized as one of the fastest cluster computing systems for big data analytics. It is up to 100x faster than Hadoop for large scale data processing, extending the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing [21]. The main reason it is so fast, comes from its ability to run computations in memory, thus is still more efficient than MapReduce for complex applications running on disk due to further optimizations and it is noteworthy to point out that Apache Spark currently holds the world record for large-scale on-disk sorting.

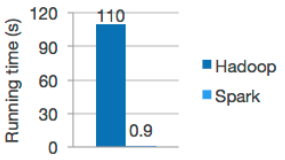


Figure 2.2: Logistic regression in Hadoop and Spark

2.2.2 A Unified Engine

Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms (i.e. machine learning algorithms etc.), interactive queries (e.g SQL queries), and streaming. By supporting these workloads in the same engine, Spark makes it easy and inexpensive to combine different processing types creating complex workloads, which is often necessary in production data analysis pipelines. In addition, it reduces the management burden of deploying, maintaining, testing and supporting separate tools thus increasing developer productivity.

At this point, it is made clear that the Spark project contains multiple closely integrated components. At its core, Spark is a "computational engine" that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a computing cluster. Because the core engine of Spark is both fast and general-purpose, it powers multiple higher-level components specialized for various workloads, such as SQL or machine learning. These components are designed to inter-operate closely, letting you combine them like libraries in a software project.

2.2.3 Ease of Use

Spark has easy-to-use APIs for operating on large datasets. This includes a collection of over 100 operators for transforming data and familiar data frame APIs for manipulating semi-structured data. Also the philosophy of the tight integration described in the previous subsection has several benefits. First, all libraries and higher-level components in the stack benefit from improvements at the lower layers. In example, if an optimization is added at Spark's core engine, SQL and machine learning libraries automatically speed up as well. Except from that, each time a new component is added to the Spark stack, everyone that uses Spark can immediately try this out. This is a very useful feature for people and organizations, since it makes it easy for them to try out a new type of data analysis on a framework they already know and work on thus saving them a lot of time that would be spent on downloading, deploying and learning a new software.

Finally, this tight integration provides the ability to build applications that seamlessly combine different processing models. For example, in Spark you can write one application that uses machine learning to classify data in real time as it is ingested from streaming sources. Simultaneously, analysts can query the resulting data, also in real time, via SQL (e.g., to join the data with unstructured logfiles). In addition, more sophisticated data engineers and data scientists can access the same data via the Python shell for ad hoc analysis. Others might access the data in standalone batch applications. All the while, the IT team has to maintain only one system [21].

2.3 Components

It is now time to perform a thorough analysis of Apache Spark's main components. They are basically consisted of the parts shown in figure 2.1β′ adding to them the stack of the cluster manager as shown in the following figure 2.3.

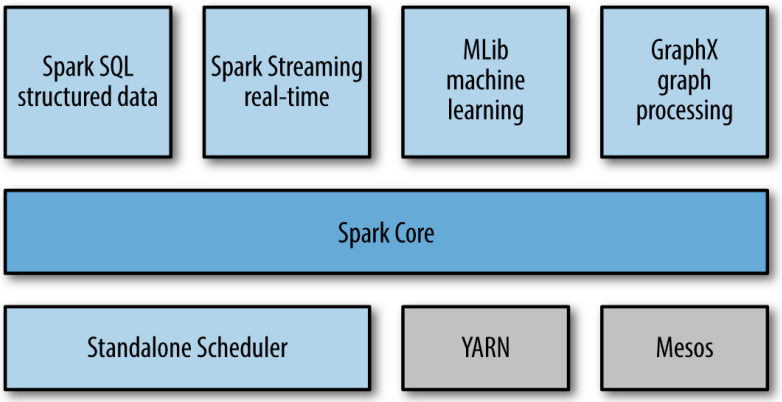


Figure 2.3: The Spark Stack[21].

2.3.1 Spark Core

Spark Core is the underlying general execution engine for the Spark platform that all other functionality is built on top of. It provides in-memory computing capabilities to deliver speed and a generalized execution model to support a wide variety of applications. It also provides Java, Scala, and Python APIs for ease of development.

Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more. Spark Core is also home to the API that defines resilient distributed datasets (RDDs), which are Spark's main programming abstraction and will be presented in more detail below.

Resilient Distributed Datasets - RDD

As mentioned before, the Resilient Distributed Dataset (aka RDD) is the primary data abstraction in Apache Spark and the core of Spark. A RDD is a fault-tolerant collection of elements that can be operated on in parallel. The concept of RDD as well as its name, first appeared in the paper *Resilient Distributed Datasets: A*

Fault-Tolerant Abstraction for In-Memory Cluster Computing by Matei Zaharia, et al. [22].
The features of RDDs (decomposing the name):

- **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures.+
- **Distributed** with data residing on multiple nodes in a cluster.
- **Dataset** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects (that represent records of the data you work with).

RDDs can be created mainly in two ways: by parallelizing an existing collection in the driver program, or by referencing a dataset from any storage source supported by Hadoop, including the local file system, HDFS, Cassandra, HBase, Amazon S3, etc. Spark supports text files, SequenceFiles, and any other Hadoop InputFormat [18]. An important parameter for parallel collections is the number of partitions to cut the dataset into. Spark will run one task for each partition of the cluster. By default, Spark has a variable set automatically, which basically refers to the default level of parallelism. In that way, Spark tries to set the number of partitions that will be created upon the characteristics of the corresponding cluster. However, the level of parallelism can be set manually by passing it as a second parameter to parallelize (e.g. sc.parallelize(data, 10)).
RDDs support two types of operations: *transformations*, which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation on the dataset. For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program. In order for Spark to be more efficient, all of the transformations upon an RDD are lazy, in that they do not compute their results until an action requires a result to be returned to the driver program. For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.

One of the most important capabilities in Spark is that the RDDs can also be persisted in memory and disk. By persisting the elements of a previously transformed RDD either in memory or in disk, Spark is able to access them much faster (often by more than 10x when peristed in memory) since new actions upon them don't require re-computations. Caching is a key tool for iterative algorithms and fast interactive use.
On the other hand, Spark also provides a mechanism for manually removing data from the cluster using the "*unpersist()*" method. Apart from that, Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion.

2.3.2 Built-in Libraries

Machine Learning (MLlib)

Machine learning has quickly emerged as a critical piece in mining Big Data for actionable insights.
Spark comes with a library containing common machine learning (ML) functionality, called MLlib. Built on top of Spark, MLlib is designed to scale out across a cluster. It delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) including classification, clustering, regression and collaborative filtering, and great speed (figure 2.2). It also provides some lower-level ML primitives, including a generic gradient descent optimization algorithm. What is more, the library can be used either in Java, Python or Scala as part of Spark applications. This enables the users to include it in complete workflows.

Spark SQL

Many data scientists, analysts, and general business intelligence users rely on interactive SQL queries for exploring data. Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine. It enables unmodified Hadoop Hive queries, that supports many sources of data including Hive tables, Parquet and JSON, to run up to 100x faster on existing deployments and data. Beyond providing a SQL interface to Spark, Spark SQL allows developers to intermix SQL

queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics. This tight integration with the rich computing environment provided by Spark makes Spark SQL unlike any other open source data warehouse tool. Spark SQL was added to Spark in version 1.0.

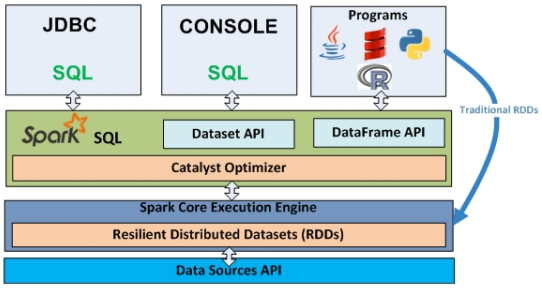


Figure 2.4: Spark SQL Architecture[24].

Just to quote here that Shark was an older SQL-on-Spark project out of the University of California, Berkeley, that modified Apache Hive to run on Spark. It has now been replaced by Spark SQL to provide better integration with the Spark engine and language APIs.

Spark Streaming

Spark Streaming is a Spark component that enables processing of live streams of data. It is a very important package, since many applications need the ability to process and analyze not only batch data, but also streams of new data in real-time. The data streams could either include log-files of a production web server, or queues of messages containing status updates posted by users of a web service. Running on top of Spark, Spark Streaming enables powerful interactive and analytical applications across both streaming and historical data. It provides an API for manipulating data streams that closely matches the Spark Core's RDD API, making it easy for programmers to learn the project and move between applications that manipulate data stored in memory, on disk, or arriving in real time. It readily integrates with a wide variety of popular data sources, including HDFS, Flume, Kafka, and Twitter. Finally, it is noteworthy to point out that Spark Streaming was designed to provide the same degree of fault tolerance, throughput, and scalability as Spark Core.

Figure 2.5: Spark Streaming Architecture.

Graph Processing (GraphX)

GraphX is a graph computation engine for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations. It is built on top of Spark that enables users to interactively build, transform and reason about graph structured data at scale. Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. GraphX also provides various operators for manipulating graphs (e.g., subgraph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting).

2.3.3 Cluster Managers

A Spark application can be either executed locally or in distributed mode on a cluster.

If spark is ran in ***local mode***, a non-distributed single-JVM is created. In this JVM, Spark spawns all the needed execution components including the driver program, the executor, the master and the LocalSchedulerBackend (as shown in figure 2.6). The local mode is very convenient for testing, debugging or demonstration purposes as it requires no earlier setup to launch Spark applications. It is also very convenient for testing on embedded systems where the limitations in processing resources, and especially the capacity of RAM, are critical, a problem that is going to be presented in more detail in chapter 3.

But Spark is designed to efficiently scale up from one to many thousands of compute nodes. To achieve this while maximizing flexibility, Spark can be ran in ***cluster mode*** using one of the next three cluster managers: Hadoop YARN, Apache Mesos, and the Standalone Scheduler. Spark is agnostic to the underlying cluster manager, all of the supported cluster managers can be launched on-site or in the

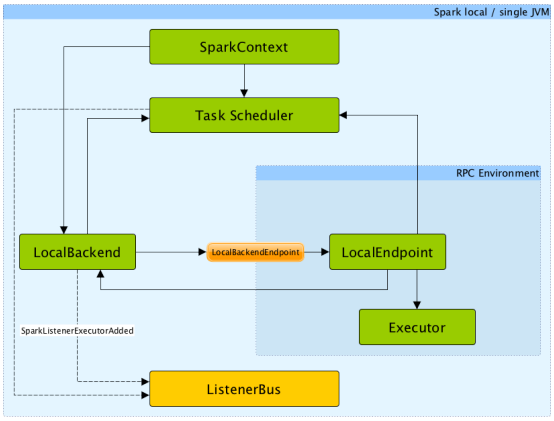


Figure 2.6: Architecture of Spark Local Mode [23].

cloud. All have options for controlling the deployment's resource usage and other capabilities, and all come with monitoring tools. The cluster manager is responsible for the scheduling and allocation of resources across the host machines forming the cluster.

Spark uses a master/slave architecture while the so called driver program (aka the SparkContext object), is the central coordinator and the main process of a submitted application. Spark driver requests executors from a cluster manager. Each executor comes with a number of CPUs and an amount of memory. In other words, the resources requested from the cluster manager are basically the previously mentioned ones (CPU cores and memory). It is a cluster manager's responsibility to spawn Spark executors in the cluster. The driver process is responsible for converting a user application into smaller execution units called tasks. These tasks are then executed by executors which are worker processes that run the individual tasks[25].

Another noteworthy feature when running Spark in cluster mode, is that an application can be deployed in two modes: ***client*** and ***cluster***. In ***cluster deploy mode*** the driver program is launched inside of the cluster in one of its workers and the client process exits as soon as it fulfills its responsibility of submitting the application without waiting for the application to finish. In ***client deploy mode***, the submitter launches the driver outside of the cluster as an independent process which is typically the same as the client process used to initiate the job.

An overview of Spark's cluster mode is depicted in the following figure, while table

2.1 summarizes the terms being used to refer to cluster concepts.

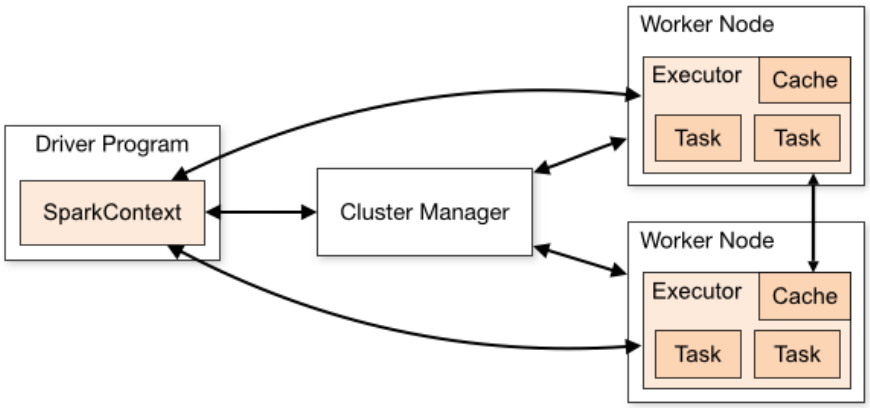


Figure 2.7: Cluster Mode Overview [18].

Standalone Scheduler

Spark Standalone cluster (aka standalone cluster or Spark deploy cluster) is a simple cluster manager that is included in the Spark distribution. Since it is Spark's built-in clustered environment, it is the easiest way to run your Spark applications in a clustered environment in many cases. It comes with a plethora of features, including high availability for the master, resilience to worker failures and it can also manage the resources provided per application. One of its most important features is that it can run alongside of an existing Hadoop cluster and access data from a Hadoop distributed file system (aka HDFS). The Standalone cluster manager also comes with a bunch of scripts (i.e for launching and stopping a Standalone Master or several Standalone Workers etc.), making it easy when it comes to deploying it locally or even on Amazon EC2. What makes it more special, is that it supports a huge variety of operating systems ranging from Linux to Windows and Mac OSX. [23]

Spark standalone uses a simple FIFO scheduler for applications. By default, each application uses all the available nodes in the cluster. The number of nodes can be limited per application, per user, or globally. Other resources, such as memory, cpus, etc. can be controlled via the application's SparkConf object [25].

In this thesis, everything will be setup and ran using the Standalone cluster manager.

Term	Meaning
Application	User program built on Spark. Consists of a driver program and executors on the cluster.
Application jar	A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.
Driver program	The process running the main() function of the application and creating the SparkContext
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Worker node	Any node that can run application code in the cluster
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Task	A unit of work that will be sent to one executor
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect); you'll see this term used in the driver's logs.
Stage	Each job gets divided into smaller sets of tasks called stages that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

Table 2.1: Glossary for terms used to refer to cluster concepts [18].

Yarn

Hadoop YARN is a general-purpose distributed computing framework for job scheduling and cluster resource management and was initially produced to improve the management of MapReduce jobs. It though quickly turned itself into also supporting non-MapReduce applications, like Spark on YARN. Like the Standalone Scheduler, it has HA for the masters and slaves (aka workers), support for Docker containers in non-secure mode, Linux and Windows container executors in secure mode, and a pluggable scheduler. It supports almost the same operating systems as the Standalone Scheduler does, except for MAC OSX [23].

Mesos

Apache Mesos, a distributed systems kernel, has HA for masters and slaves, can manage resources per application, and has support for Docker containers. It abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual). It can run Spark jobs, Hadoop MapReduce, or any other service application. It has API's for Java, Python, and C++. It can run on Linux or Mac OSX. [25]

2.3.4 Monitoring

There are several ways to monitor Spark applications: web UIs, metrics, and external instrumentation. Each Apache Spark application has a Web UI for monitoring purposes, that is launched through the application's SparkContext. The Web UI shows information about the scheduler stages and tasks, a summary of RDD sizes and memory usage, as well as information about the running executors, and the storage that is being used. A web UI is also provided through the Standalone cluster manager, which contains information about cluster and job statistics as well as detailed log output for each job. Past submitted applications can also be reconstructed in a graphical view using Spark's History server. Examples of Spark's UI and the History server are going to be presented later on, in the evaluation of our proposed framework.

2.4 Configuration

Spark provides a concrete configuration scheme. It includes properties that control most applications parameters, environment variables that can be used to set per-machine settings (i.e IP address, python version for PySpark etc.) and even properties for logging.

2.4.1 Spark Properties

Spark properties control most application settings and are configured separately for each application. These properties can be set directly on a SparkConf object which is then passed to a SparkContext one, or by adding configuration options in the ***conf/spark-defaults.conf*** configuration file. Any values specified as flags or

in the properties file will be passed on to the application and merged with those specified through SparkConf. Properties set directly on the SparkConf take highest precedence. These properties include information for the url of the master node, the number of executors per application etc.

2.4.2 Environment Variables

Certain Spark settings can be configured through environment variables, which are read from the ***conf/spark-env.sh*** script in the directory where Spark is installed. In Standalone and Mesos modes, this file can give machine specific information such as hostnames. It is also sourced when running local Spark applications or submission scripts.
It is the most important configuration file for this thesis, along with the *conf/spark-defaults.conf* since it will be used for enabling Spark to run on embedded systems.

2.4.3 Logging

Spark uses log4j for logging. The configurations are set adding a log4j.properties file in Spark's conf directory.

3 Spark on Embedded Systems

3.1 Purpose

Emerging web applications like big data analytics have increased significantly the workload on the data centers during the last years. In 2015, the total network traffic of the data centers was around 4.7 Exabytes and it is estimated that by the end of 2018 it will cross the 8.5 Exabytes mark, following a cumulative annual-growth rate (CAGR) of 33% (Figure 3.1, [26]). In response to this scaling in network traffic, data-center operators have resorted to utilizing more powerful servers.

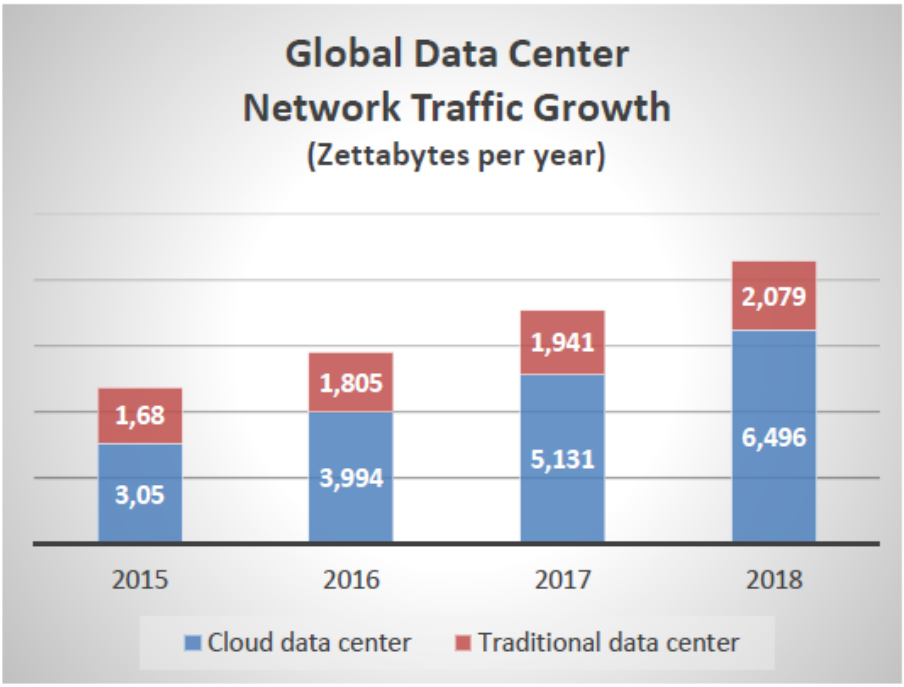


Figure 3.1: Global Data Center Network Traffic Growth Projection, Source: Cisco Global Cloud Index 2016.

Relying on Moore's law for the extra edge, CPU technologies have scaled in re-

cent years through packing an increasing number of transistors on chip, leading to higher performance. However, on-chip clock frequencies were unable to follow this upward trend due to strict power-budget constraints. Thus, a few years ago a paradigm shift to multi-core processors was adopted as an alternative solution to the problem. With multi-core processors the servers' performance could be increased, without having to increase the clock frequency. Unfortunately, this solution did not scale well in the long term. The performance gains achieved by adding more cores inside a CPU come at the cost of various, rapidly scaling complexities like inter-core communication, memory coherency and most importantly power consumption [27].

One way to address this problem is through the utilization of microservers. Microservers have recently gained attention as low-cost, low power, reduced footprint servers that are mainly based on low-power energy-efficient SoC-based processors such as the ones used in embedded systems. Microservers are mainly targeting lightweight applications or parallel applications that benefit most from individual servers with sufficient I/O between nodes rather than high performance processors.

All these facts led us to perform a thorough performance evaluation, in terms of execution time and power consumption, of several embedded platforms. Comparing the results with these of a powerful mainstream server could give us answers to whether embedded systems could play a key role in the datacenter and whether there are any benefits from running big data applications on them. To perform this evaluation, we first had to supply a series of embedded systems and then find some representative applications for taking the metrics. Apache Spark was a perfect fit for our purpose. It is 100% open-source, it is a framework for big data analytics and also provides built-in algorithms and applications.

Below we are going to take a closer look on the embedded platforms and the specific applications we used for the evaluation. Then, we will go through the challenges we faced trying to map Spark on these platforms. There will be a thorough description of all the necessary steps we had to follow for the mapping, including the required changes in Spark configuration.

3.2 Embedded Platforms

Today, there is a plethora of low-power embedded platforms available for sale. For the purpose of this evaluation we chose to supply platforms, the low-power processors of which are based on the ARM architecture.

3.2.1 Raspberry Pi 3 - Model B

The first board we used, is the famous Raspberry Pi 3 board (figure 3.2). It is a rather small single-board computer and was first developed in the United Kingdom by the Raspberry Pi Foundation to promote the teaching of basic computer science in schools and in developing countries. The original model became far more popular than anticipated, selling outside of its target market for uses such as robotics. [28]

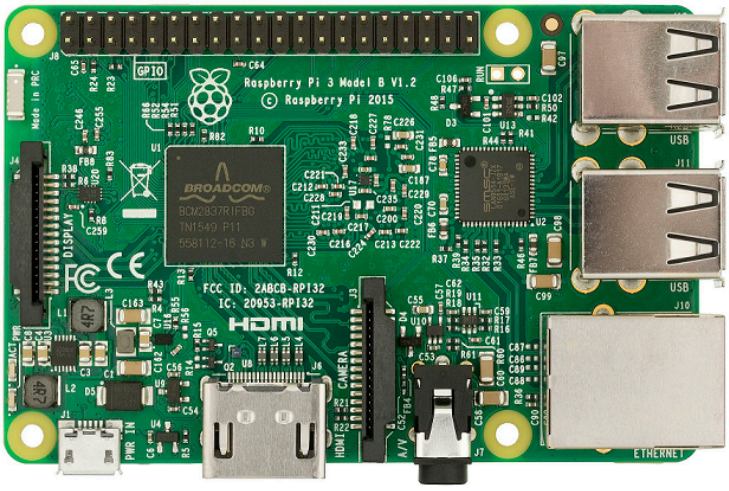


Figure 3.2: Raspberry Pi 3 Model B.

This board, features the Broadcom BCM2837 system on a chip (SoC), which includes an ARM compatible central processing unit (CPU) and an on-chip graphics processing unit (GPU, a VideoCore IV). It provides several ports for interfacing with other devices including 4 USB 2 slots, a full size HDMI, a composite video output, and a 3.5mm phono jack for audio. It also has an 8P8C Ethernet port as well as on board Wi-Fi 802.11n and Bluetooth. It has a 64-bit quad-core ARM Cortex-A53 CPU clocked at 1.2 GHz and also comes with 1 GB LPDDR2 RAM of on board memory operating at 900 MHz [29]. Secure Digital (SD) cards are

used to store the operating system and program memory. The Foundation provides Raspbian, a Debian-based Linux distribution which is the one we used for the evaluation.

It is noteworthy to point out that although the Pi 3 hosts a 64-bit CPU, there was no availability of a 64-bit operating system by the time the evaluation was made. This means that the processors were running in 32-bit mode.

3.2.2 Dragonboard 410c

Another board we used, is the DragonBoard 410c (figure 3.3) . The DragonBoard 410c is the first development board based on a Qualcomm Snapdragon 400 series processor. It features advanced processing power, Wi-Fi, Bluetooth connectivity, and GPS, all packed into a board the size of a credit card. Based on the 64-bit capable Snapdragon 410E processor, the DragonBoard 410c is designed to support rapid software development, education and prototyping. All this makes it ideal for enabling embedded computing and Internet of Things (IoT) products, including the next generation of robotics, cameras, medical devices, vending machines, smart buildings, digital signage, casino gaming consoles etc. [30]

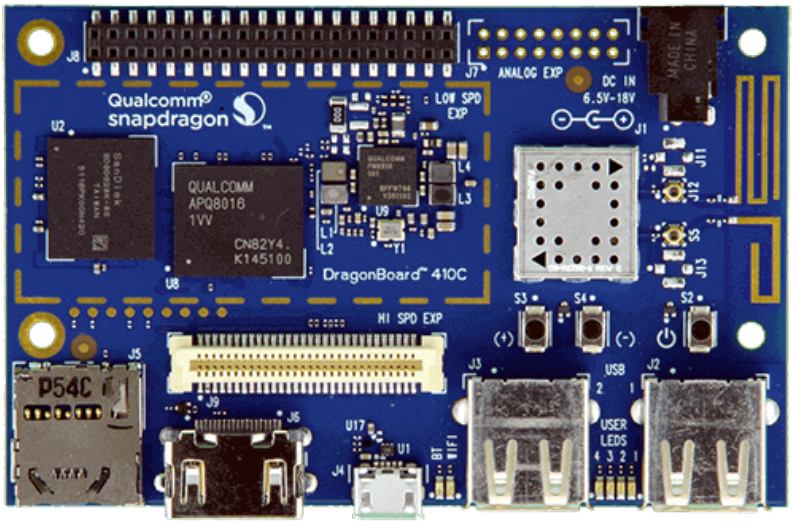


Figure 3.3: Dragonboard 410c.

In more detail, this board has a 64-bit quad-core ARM Cortex-A53 CPU clocked at 1.2 GHz, 1GB of LPDDR3 RAM which operates at 533MHz and 8GB of eMMC 4.5 storage space. As far as the GPU is concerned, it sports a Qualcomm Adreno

306 GPU with support for advanced APIs, including OpenGL ES 3.0, OpenCL, DirectX, and content security.

It is important to note here, that although the Pi 3 and the DragonBoard 410c have the same processor, the BCM2837 SoC is manifactured at a 40nm process while the Snapdragon 410 SoC is manifactured at 28nm process. Also, both boards have 1GB of RAM but as we can see the memory module of the DragonBoard 410c board, operates at a much lower frequency.

3.2.3 Pynq-Z1

The PYNQ-Z1 board is designed to be used with PYNQ, a rather new Xilinx open-source framework that enables embedded programmers to exploit the capabilities of Xilinx Zynq All Programmable SoCs (APSoCs) without having to design programmable logic circuits. Instead, the APSoC is programmed using Python and the code is developed and tested directly on the PYNQ-Z1. The programmable logic circuits are imported as hardware libraries and programmed through their APIs in exactly the same way that the software libraries are imported and programmed. [31]

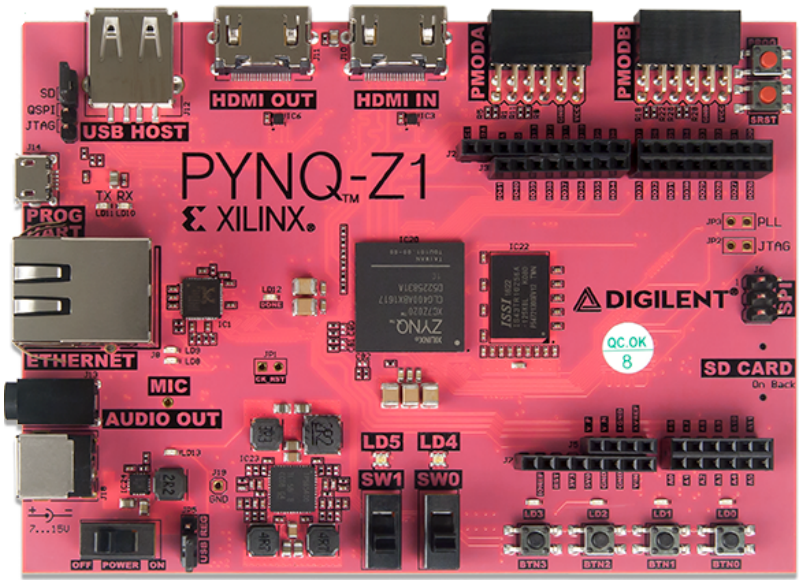


Figure 3.4: PYNQ-Z1.

It hosts a ZYNQ XC7Z020-1CLG400C SoC which features a 32bit 667MHz dual-core Cortex-A9 processor and a DDR3 memory controller with 8 DMA channels and 4 high performance AXI3 slave ports and an Artix-7 family programmable

logic. It also has 512MB of DDR3 memory and 16MB of Quad-SPI Flash with factory programmed globally unique identifier.
Undeniably, the specifications of this board are inferior to the ones of the Pi 3 and the DrangonBoard 410c. Although we don't expect this board to overcome the performance of the previous ones, the main reason why we chose to include it into our evaluation, is that in chapter 4 we are going to develop a framework for accelerating algorithms and applications that run on Spark. In other words, this evaluation could help us better understand the gains in performance and power consumption of using hardware accelerators.

3.3 Spark Applications

To evaluate the performance and energy-efficiency of the low-power SoCs, we used a set of Spark applications. In more detail, we have evaluated three representative applications of the machine learning domain and three of the graph computation domain. Below we are going to take a quick look at each application, by briefly describing their use.

3.3.1 Machine Learning Applications

- **Linear Regression:** Linear regression is used for modeling the relationship between a scalar dependent variable y and one or more explanatory variables (or independent variables) denoted X. The case of one explanatory variable is called simple linear regression [32].

- **Logistic Regression:** Logistic regression or logit regression, is a regression model where the dependent variable (DV) is categorical. It measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic function. [33]

- **K-Means:** K-means is a clustering algorithm. It is a method of vector quantization that is widely used for cluster analysis in data mining. It aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. [34]

3.3.2 GraphX Applications

- **Connected Components - CC:** CC computes the connected components of verices in a graph. In more detail, the connected components algorithm labels each connected component of the graph with the ID of its lowest-numbered vertex. For example, in a social network, connected components can approximate clusters. [18]

- **PageRank:** PageRank (PR) is an algorithm used to rank websites in search engines. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites. [35]

- **Triangles:** This algorithm counts the triangles in a graph. A vertex is part of a triangle when it has two adjacent vertices with an edge between them. The algorithm determines the number of triangles passing through each vertex, providing a measure of clustering.

3.4 Mapping Spark

Now that we have gone through the platforms and the applications we used for the evaluation, we are ready to present the procedure we followed to map Apache Spark on these embedded devices. Spark was first deployed on the Pi3 and DragonBoard 410c boards that sport almost the same hardware resources.

To begin with, Apache Spark can be downloaded either as a pre-built or a source code package. Although there is not a great difference between the two types of packages, we chose to go with the source code option. In that way we were able to set the Scala version upon which Spark would be built, the minimum compatible version of a Hadoop distributed filesystem as well as the support and the inclusion of other packages (i.e. support for Mesos, packaging with or without Hadoop dependencies for YARN etc.). Apart from the options it provided, this gave us the opportunity to try out and see if it was possible for Spark to be built on these embedded devices.

We tried to build Spark on both Pi 3 and DragonBoard 410c. By the time we started to work on this evaluation, Apache Spark 1.6 was available. To build Spark a prerequisite is that a Java development kit is already present and set correctly on the system. In that way, the first thing needed to be done after downloading and extracting the Spark source code was to download a jdk version for the ARM systems. Seeing that with Java 7 we had to add additional options like "-XX:MaxPermSize=512M" for avoiding errors and warnings in the build, we decided to choose Java 8 where we didn't need to add any additional options. Since Pi's Raspbian OS is a 32bit Debian distribution, a 32-bit jdk package was downloaded while for the Dragonboard 410c, that comes with a 64-bit Debian OS, a 64-bit package was downloaded. After downloading and extracting the packages, $JAVA\_HOME$ and $PATH$ variables should be set accordingly under the *.bashrc* file (located in the home directory of the logged in user), in order for Java to be visible to Spark. The commands added to the *.bashrc* file are shown below:

```
export JAVA_HOME="<path/to/jdk/folder>"
export PATH="$PATH:$JAVA_HOME/bin"
```

Listing 3.1: Java configuration

After sourcing .bashrc (e.g $sudo source .bashrc), we continued with the steps required to build Spark. Spark comes packaged with a self-contained Maven installation to ease its building and deployment. It includes a script which automatically downloads and setups all necessary build requirements (Maven, Scala, and Zinc) locally [18]. The information about the project and all the configurations and packages needed to build the project are contained in an XML file the *pom.xml*. The default version of scala was 2.10 so in order to change it to 2.11 we first executed the following command:

```
./dev/change-scala-version.sh 2.10
```

Finally, we started building Apache Spark by executing the following command.

```
build/mvn -Pyarn -Phadoop-2.6 -Dhadoop.version=2.6.0 -Dscala-2.11 -
    DskipTests clean package
```

In that way, we set the minimum Hadoop version to 2.6, the version of Scala for the compilation to 2.11, and we also added support for YARN.

3.4.1 Challenges of Building Spark on Embedded Systems

Although compiling the Spark source code with Java 8 does not require any additional options to be set, we came up with JVM (Java Virtual Machine) errors regarding the heap size. So, Java could not allocate the default heap size and exceptions were caused. To overcome this problem, before re-submitting the command for building Spark we explicitly set the maximum heap size that could be allocated to 512MB and the maximum size of the memory allocation pool to 800MB, by executing the following command in the terminal:

```
export MAVEN_OPTS="-Xmx=800M -XX: MaxHeapSize=512M"
```

Another problem we faced while trying to build Spark on the embedded systems, was that Snappy, a compression/decompression library that is used for building Spark, did not support the ARM architecture. To overcome this problem we had to modify the pom.xml file which contains all the dependencies for building Spark and is located under its home directory, and specify to download and use the latest version of Snappy, which by the time was the 1.1.2.4 and in which there was added support for ARM architectures. The modification made is presented below.

```
### old XML tag value ###
<snappy.version>1.1.2.3</snappy.version>
### new XML tag value ###
<snappy.version>1.1.2.4</snappy.version>
```

After following the above steps, the build was successful.

3.4.2 Challenges of Running Spark on Embedded Systems

A user can easily test if everything works properly, by executing any of the examples included in Spark, or by typing: *./bin/spark − shell.sh* in the terminal (from the Spark home folder) to open an interactive Spark shell, where Scala and Java code can be evaluated. By running spark-shell, we realized that there was again a problem with Java. Like before, the JVM threads were trying to allocate more than the available space. The problem was that by default, Spark sets the maximum pool of available memory to 1GB. But both boards (Pi 3 and DragonBoard 410c) have only 1GB (actual size is less than 1GB) of RAM and still

an amount of memory is allocated by the operating system. As shown in chapter 2, Spark has a concrete scheme for configuring the resources it allocates and uses when an application is submitted or a cluster is started. In more detail, in the $/conf/spark-env.sh$ script there are options for setting the driver, executor, master, worker and daemon memory as shown below:

Option	Comments
SPARK_EXECUTOR_MEMORY	Memory per Executor (Default: 1G)
SPARK_DRIVER_MEMORY	Memory for Driver (Default: 1G)
SPARK_WORKER_MEMORY	To set how much total memory workers have to give executors
SPARK_DAEMON_MEMORY	To allocate to the master, worker and history server themselves (Default: 1G)

Table 3.1: Spark memory configuration options.

From the above options we set the driver memory to 800MB for evaluating the applications in Spark's local mode and the problem was resolved.

```
export SPARK_DRIVER_MEMORY=800MB
```

It is noted at this point that the same configuration could be added in $spark-defaults.conf$ file. As it is shown below:

```
spark.driver.memory=800MB
```

The main difference between these two cases, is that in the first one, the contents of the spark-env.sh script are sourced at the beginning of Spark's execution, meaning that its entries are transformed into environment variables, while in the latter case, the contents of the spark-defaults.conf file are passed as configurations to any submitted application and merged with any further configurations that are set in the SparkConf() object.

But for evaluating the applications in Spark's cluster mode, we should also set the memory for the master and worker nodes respectively. Both the Pi 3 and DragonBoard 410c have a quad core CPU, which means that they can host up to 4 workers each. Also, it is important to note that each master and worker node runs in its own JVM process. Spark, limits the minimum amount of memory given to each process (i.e the master, the driver, the executor etc.) to 475MB. In this way, even in the simplest case of creating a cluster consisted of a master and one

worker node we need to have available $475MB + 475MB = 950MB$ of memory. But because of the fact that only 800MB-850MB of RAM is truly available on the embedded devices, we were not able to setup a Spark cluster and evaluate the applications in this mode.

As far as the PYNQ-Z1 platform is now concerned, its resources in CPU and RAM are far more limited than these of the Pi 3 and the DragonBorad 410c. For this reason, it did not make sense to try and build Spark from source, therefore we deployed an already built version of it with the exact same built-in components (i.e Scala 2.11, Hadoop 2.6 etc.).

4 Spark on a Pynq Cluster

4.1 Introduction

The growing demands both in performance and energy efficiency have led companies into charting new paths by deploying FPGA accelerators on datacenter scale. Many colossal brands like Amazon and Intel have already made moves into the FPGA market. For example, Intel recently acquired Altera [36], a very known FPGA-maker company, while Amazon already provides the so called F1 instances which include a programmable logic [14].

For this reason, we wanted to build an heterogeneous cluster that could be capable of accelerating the computational intensive parts of algorithms. So, in this chapter we are going to present the whole setup process of creating and configuring this cluster consisted mainly of PYNQ-Z1 nodes, have also embedded a programmable logic, and a master node which hosts a processor based on the Intel architecture. Spark will be deployed on all of the cluster's nodes and will be configured for running over it. Furthermore, an efficient framework for the seamless utilization of hardware accelerators in Spark will be presented along with the available application programming interfaces (APIs) it provides. Evaluation metrics will be performed to assess the whole project.

4.2 PYNQ-Z1

One of the most important questions to answer, is why we chose the PYNQ-Z1 to build our cluster. The answer is that its features and hardware specifications meet the requirements of such a project. In more detail, the characteristics upon which we made this decision are the following:

- **Zynq APSoC:** To begin with,the PYNQ board is based on the Zynq All Programmable SoC and specifically on the Zynq-7000 family. All Zynq-7000 devices are equipped with a dual-core ARM Cortex-A9 processor, that is integrated along with an Artix-7 or Kintex-7 based programmable logic, for excellent performance-per-watt and maximum design flexibility. In this way, a user is able to combine the software programmability of the ARM processors with the hardware programmability of the programmable logic. Figure 4.1 depicts the block diagram of Zynq-7000 series.

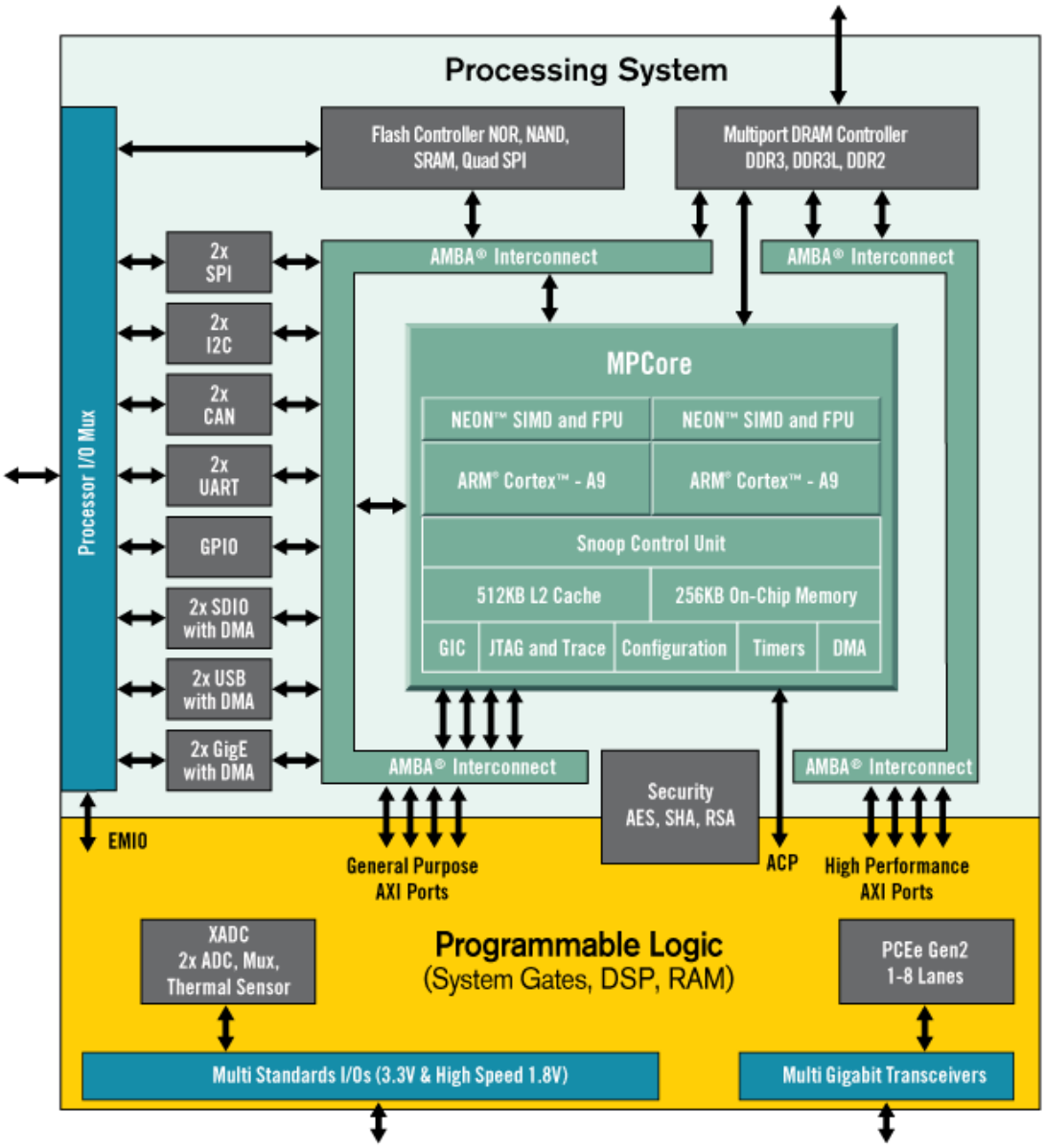


Figure 4.1: Zynq-7000 series block diagram.

On the other hand, when it comes to accelerating an algorithm, it seems that only parts of it can be truly accelerated. These parts are the computational intensive ones. Therefore, it seems that the ideal platform for accelerating an algorithm has to support both the software and hardware programmability we previously referred to. It is obvious, that the Zynq platform is a perfect fit for this purpose and of course it is perceived that this was one of the main reasons for using the PYNQ platform.

- **Low power:** As we have already seen in chapter 3, the PYNQ-Z1 board belongs to the family of the low-power SoC embedded platforms. It's energy consumption hardly trespasses the 3.2W limit, while it hosts a full Ubuntu distribution.

  The main part of this thesis is to explore new alternatives for advancing the performance of datacenters while reducing the overall energy consumption. So, up to this point, there are two reasons that make the PYNQ-Z1 board ideal for our purpose.

- **Small size:** All of PYNQ's components, including the external I/O interfaces and the APSoC are packed on a rather small sized printed board circuit (PCB) almost three times the size of a credit card. In this way, not only do we save space but it can also cope with today's great needs for scaling.

- **Python libraries:** PYNQ-Z1 is a platform based on a rather new open-source Xilinx project called PYNQ. The main advantage of this project is that it enables users to easily design embedded systems with Xilinx Zynq APSoCs. The accelerators are imported as hardware libraries (the so called overlays), while the whole application including the hardware accelerator's interface are written in Python.

  Apart from that, Python is increasingly used in the big data analytics field because of being a powerful, flexible, open source language that is easy to learn, easy to use, and has powerful libraries for data manipulation and analysis. Python has a unique combination of being both a capable general-purpose programming language as well as being easy to use for analytical and

quantitative computing [38]. Therefore, we believe in the future of serving hardware accelerators through Python driver libraries.

4.3 The Spark on Pynq (SPynq) Cluster

In this section we are going to present the whole procedure of building the SPynq Cluster. For the purposes of the cluster we were supplied a typical personal computer based on an Intel i5 processor and four PYNQ-Z1 boards along with a small switch to create the cluster network.

To set the PYNQ-Z1 boards, we consulted the quickstart guide. We then downloaded Xilinx's PYNQ image by navigating to the GitHub project. After writing this image to four sd cards respectively, we were able to boot up the PYNQ-Z1 nodes. Below we are going to take a closer look at the configurations we made:

4.3.1 Network Configurations

PYNQs by default, are given a static IPv4 resolving to *192.168.2.99*. To avoid any conflicts between the boards at a network scale, we had to change the networking configuration of each node and provide different IPs for them. The IPv4 of the Intel based node was also modified to a static one. In more detail, under the */etc/network/interfaces.d* path we modified the *static* file. The modifications we made are presented below:

```
### Old configuration for the static IP ###

address 192.168.2.99

### New configuration for the static IP ###

address 192.168.1.<231-234 depending on the pynq node and 203 for the
    master>
gateway 192.168.1.1
```

Listing 4.1: Ip configuration

But the PYNQ-Z1 nodes also provide information for a fixed IP address we also had to modify the the *dhclient.conf*, which is located under the */etc/dhcp* directory.

Below is presented the final dhcp configuration along with the old value for the fixed-address:

```
### Old fixed-address value ###

alias {   interface "eth0";
  fixed-address 192.168.99;
  option subnet-mask 255.255.255.0; }

### New DHCP configuration ###

option rfc3442-classless-static-routes code 121 = array of unsigned
    integer 8;

send host-name = gethostname();

request subnet-mask, broadcast-address, time-offset, routers,
domain-name, domain-name-servers, domain-search, host-name,
dhcp6.name-servers, dhcp6.domain-search,
netbios-name-servers, netbios-scope, interface-mtu,
rfc3442-classless-static-routes, ntp-servers,
dhcp6.fqdn, dhcp6.sntp-servers;

timeout 30;
retry 10;
reboot 3;
select-timeout 0;

alias {   interface "eth0";
  fixed-address 192.168.231;
  option subnet-mask 255.255.255.0; }
```

Listing 4.2: DHCP configuration

Furthermore, as an optional step, we wanted to create some DNS entries to refer to the nodes using their hostnames and not their IP addresses. We will skip the procedure of creating the DNS entries since it has nothing to do with the implementation of the cluster, but we will dwell on how he changed the hostname of each PYNQ board. Xilinx provides a script that is named *hostname.sh* for this exact

purpose and is located under the */home/xilinx/scripts* directory. By executing the following command on a terminal, we were able to change the PYNQs' hostnames. It is important to note that for the changes to take effect, a reboot is needed.

```
$ sudo ./hostname.sh <new hostname>
```

Listing 4.3: The command for altering PYNQs' hostnames

Finally, we present the network configuration of our cluster:

Hostname	IPv4 address	Platform
vinemaster	192.168.1.203	Intel i5 based
pynq1	192.168.1.231	PYNQ-Z1
pynq2	192.168.1.232	PYNQ-Z1
pynq3	192.168.1.233	PYNQ-Z1
pynq4	192.168.1.234	PYNQ-Z1

Table 4.1: Cluster's network configuration.

4.3.2 Security configurations

Since we wanted to create a cluster that would be accessible over the Internet, we had to make sure that some security standards are met. For this reason, we updated the Firewall policy of each individual node upon which the cluster is consisted, using the *iptables* command. Furthermore, we modified the *sshd_config* file which is placed under the */etc/ssh* directory, to permit the authentication via the SSH protocol only for users that provide a private key. The file was modified so to finally include the options below:

```
####### sshd_config ########

PermitRootLogin without-password
PubkeyAuthentication yes
PasswordAuthentication no
```

Listing 4.4: Modifications done in the *sshd_config* file

It is important to note here that we permitted empty key passphrases when logging in with the use of a public key. The main reason for doing so, will be explained later, in the Spark configuration part.
Finally, we generated a public/private rsa key pair with an empty passphrase using the following command:

```
#### linux command that generates a private/public rsa key pair ####

$ ssh-keygen
```

Listing 4.5: Generating a private/public rsa key pair

Then we kept the private part of the key at the Intel based node, which as we are going to see later, will be the cluster's master node (as far as the Spark is concerned), and finally we copied the public part of it to every PYNQ-Z1 node under the */root/.ssh* directory, appending it to the *authorized_keys* file.
It is noteworthy to point out that all the configurations relate to the root user, and of course, this was done intentionally. The main reason behind this, is that Xilinx's built-in libraries for using the programmable logic, require elevated privileges. In this way, everything has to be executed from the root user. Even the Spark worker and master threads that we will see later in this chapter, have to be invoked from the root user.

4.3.3 Spark Configurations

Before starting with the deployment of Spark, we downloaded and installed the JDK framework on all nodes. After that, we downloaded a pre-built version of Apache Spark, which we then copied and extracted under the /home/xilinx folder of each node. Furthermore, to export all the necessary environment variables, we first added the corresponding *$JAVA_HOME*, *$SPARK_HOME* and *$PATH* variables under the */etc/environment* file, and then rebooted the platforms for the changes to take effect. The reason why we added the environment variables into the */etc/environment* file and not into a user's *.bashrc*, is that we wanted them to be visible for any user of the system. After adding all of the above, the */etc/environment* file of the PYNQ-Z1 boards, looked like the following (we didn't modify the whole file as other entries could affect the systems proper functionality):

```
###### Contents of /etc/environment file ######

#Java
JAVA_HOME="/usr/lib/jvm/jdk1.8.0_111"

#Spark
```

```
SPARK_HOME="/home/xilinx/spark-2.1.1"

#PATH
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr
    /games:/usr/local/games:/home/xilinx/spark-2.1.1/bin:/home/xilinx/
    spark-2.1.1/sbin:/usr/lib/jvm/jdk1.8.0_111/bin"
```

Listing 4.6: Setting environment variables on the PYNQ-Z1 nodes

Below we quote the same configuration file of the Intel i5 based node:

```
###### Contents of /etc/environment file ######

#Spark
export SPARK_HOME=/home/xilinx/spark-2.1.1

#Java
export JAVA_HOME=/usr/lib/jvm/jdk1.8.0

#PATH
export PATH=$PATH:/home/xilinx/spark-2.1.1/bin:/home/xilinx/spark
    -2.1.1/sbin:/usr/lib/jvm/jdk1.8.0/bin
```

Listing 4.7: Setting environment variables on the Intel i5 based node

As one might observe, the directory under which Spark is located, is the same for all the involved nodes. When it comes for running Spark applications in Standalone cluster mode, Spark Master after establishing a connection with the slaves, searches for the corresponding Spark package on each worker node to launch the worker threads, on the same path as its own. Therefore, deploying the Spark on a different path would cause problems in launching the worker threads.

So we are now ready to move on to the specific configurations of Spark for the intended cluster scheme. Figure 4.2 depicts its final intended scheme. We will setup the Intel based node to host the Spark master node, while a Spark Worker will be launched on every PYNQ-Z1 node. So typically, any applications will be submitted on the master node, while the PYNQ nodes will undertake the execution of the jobs and tasks that will be assigned to them.

We had to both configure the master to connect with the slaves and vice versa. Below we will describe each case separately:

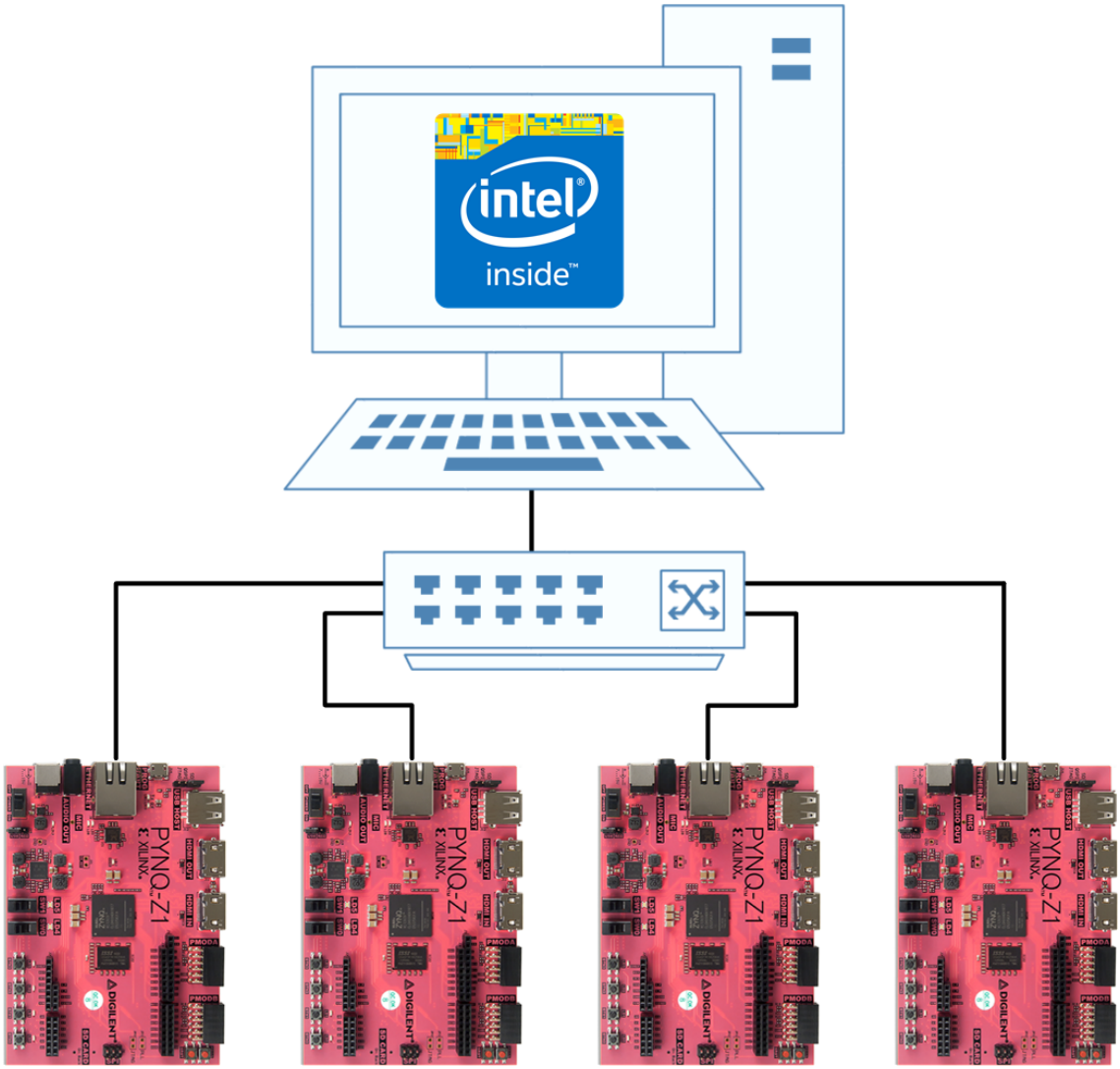


Figure 4.2: Proposed Cluster Scheme

Configuring the Spark Master node

On the Master node we configured the four following files: *slaves*, *spark-env.sh*, *spark-defaults.conf* and *log4j.properties*.

- **slaves:** Spark 2.0 and any later versions of it, comes with a *slaves* file under its *conf* directory. This file must contain the hostnames of all the machines where we intend to start Spark workers, one per line. Once we have set up this file, we can launch or stop the cluster using any of the Spark's available scripts, located under the SPARK_HOME/sbin directory. Table 4.2 shows all the available Spark scripts.

Script name	Description
sbin/start-master.sh	Starts a master instance on the machine the script is executed on.
sbin/start-slaves.sh	Starts a slave instance on each machine specified in the conf/slaves file.
sbin/start-slave.sh	Starts a slave instance on the machine the script is executed on.
sbin/start-all.sh	Starts both a master and a number of slaves as described above.
sbin/stop-master.sh	Stops the master that was started via the bin/start-master.sh script.
sbin/stop-slaves.sh	Stops all slave instances on the machines specified in the conf/slaves file.
sbin/stop-all.sh	Stops both the master and the slaves as described above.

Table 4.2: Spark available scripts for launching/stoping a cluster

So in our case, on the Intel based node, we modified Spark's slaves file to contain the hostnames of all the PYNQ-Z1 nodes. The contents of the file are presented below:

```
##### Contents of conf/slaves file #####

pynq1
pynq2
pynq3
pynq4
```

Listing 4.8: Setting the Worker nodes of the Spark Cluster

- **spark-env.sh:** After the above, we modified the *spark-env.sh* file to finally contain the next entries:

```
##### Master - conf/spark-env.sh #####

export SPARK_DRIVER_MEMORY=505m
export SPARK_DAEMON_MEMORY=505m
export SPARK_MASTER_HOST=192.168.1.203
export SPARK_LOCAL_IP=192.168.1.203
```

Listing 4.9: The spark-env.sh file of the Master Node

At first, we set the memory for the Driver program and the Spark daemon to 505MB. The amount of the memory given, was set after many tests and is so limited due to the corresponding limitation in the memory of the Worker nodes. The PYNQ-Z1 nodes have 512MB of RAM, meaning that the maximum memory we could allocate for the Spark workers, caps at this size. The amount of the Spark driver memory should also not exceed this value. Upon submitting an application, the application's given memory is set to the driver's memory . So in case that the memory of the driver is set to a greater value than the workers', the cluster manager could not accept the requested resources, since there are no slaves offering them, therefore no tasks would be assigned to the workers.

Finally, we had to set the Master Host and the Master's IP to the IP we previously assigned on the Intel based node 4.3.1.

- **spark-defaults.conf:** In this file, we have first set the Spark master URL. It was not necessary to add it in the configuration file, although in case of not including it, we would have to define it in every application submission. Then, we enabled the logging of the execution events and we finally set the path to where the events would be stored. The same path served as the logging directory for the history server as it can be seen in the listing below.

  It is worth to point that using the history server, one can navigate through past submitted applications and reconstruct its UI.

```
1 ##### Master conf/spark-defaults.conf #####
2
3 spark.master                     spark://192.168.1.203:7077
4 spark.eventLog.enabled           true
5 spark.eventLog.dir               /home/xilinx/spark-2.1.1/work
6 spark.driver.memory              505m
7 spark.history.fs.logDirectory    /home/xilinx/spark-2.1.1/work
```

Listing 4.10: The spark-defaults.conf file of the Master Node

- **log4j.properties:** This step was optional and intended to hide any unnecessary informational logs at the point of an application submission. The contents of the log4.properties file are presented below:

```
################SPARK log4j ocnfiguration ####################

# Set everything to be logged to the console, file
log4j.rootCategory=INFO, console, file
log4j.appender.console.threshold=ERROR
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:
    ss} %p %c{1}: %m%n

log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=/home/xilinx/spark-2.1.1/logs/logging.log
log4j.appender.file.MaxFileSize=5MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss}
     %p %c{1}: %m%n

# Set the default spark-shell log level to WARN. When running the
    spark-shell, the
# log level for this class is used to overwrite the root logger's
    log level, so that
# the user can have different defaults for the shell and regular
    Spark apps.
log4j.logger.org.apache.spark.repl.Main=WARN

# Settings to quiet third party logs that are too verbose
log4j.logger.org.spark_project.jetty=WARN
log4j.logger.org.spark_project.jetty.util.component.
    AbstractLifeCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=INFO
log4j.logger.org.apache.spark.repl.
    SparkILoop$SparkILoopInterpreter=INFO
log4j.logger.org.apache.parquet=ERROR
log4j.logger.parquet=ERROR

# SPARK-9183: Settings to avoid annoying messages when looking up
    nonexistent UDFs in SparkSQL with Hive support
```

```
log4j.logger.org.apache.hadoop.hive.metastore.RetryingHMSHandler=
    FATAL
log4j.logger.org.apache.hadoop.hive.ql.exec.FunctionRegistry=ERROR
```

Listing 4.11: The log4.properties file of the Master Node

Configuring the Spark Worker nodes:

On the workers' side, we only had to configure the *spark-env.sh* and *spark-defaults.conf* files respectively. Optionally we can use the same log4j.properties file on the PYNQ-Z1 nodes too. Below, we quote the contents of the corresponding files, pointing out their differences with that of the master.

- **spark-env.sh:** Compared to the master's spark-env.sh configuration file, in this case we have additionally included all the necessary environment variables related to the workers' and executors' options. More specifically we have set the worker instances that will be launched on each PYNQ-Z1 node to one and their available memory to 505MB. Apart from that, we have defined that every worker node can host only one executor instance which will also have available 505MB of RAM.

  Both the worker and the executor are given one CPU. In fact, both the CPUs of the Zynq APSoC could be used in every worker, however doing so would produce problems to the integration of Spark with the hardware accelerators. Since the tasks of Spark's executors run in parallel, this would cause a race condition between the two processors for who would be the one to access the programmable logic. Of course, this problem is closely tied with the specific features of our implemented hardware accelerator and the corresponding software library for its mapping on the software. Therefore this doesn't necessarily mean that at a given moment the programmable logic can be accessed by only one of the two processors, but it is a feature that requires more research in order to be integrated to Spark applications.

```
###### Worker - conf/spark-env.sh ######

export SPARK_DRIVER_MEMORY=505m
export SPARK_MASTER_HOST=192.168.1.203
```

```
5  export SPARK_LOCAL_IP=192.168.1.203
6
7  ##### Worker and Executor configurations ####
8
9  export SPARK_EXECUTOR_INSTANCES=1
10 export SPARK_EXECUTOR_CORES=1
11 export SPARK_EXECUTOR_MEMORY=505m
12 export SPARK_WORKER_CORES=1
13 export SPARK_WORKER_INSTANCES=1
```

Listing 4.12: The spark-env.sh file of the Worker Nodes

- **spark-defaults.conf:** There were not added any additional settings in this file compared to the same one on the master node. In fact this file is not needed in the workers' case since any applications will be submitted on the master node.

At this point we have completed the setup of Spark for the Pynq cluster. Figure 4.3, shows the web UI of the running Spark Cluster as well as the launched PYNQ workers and their available resources, while figure 4.4 depicts the web interface of the history server.

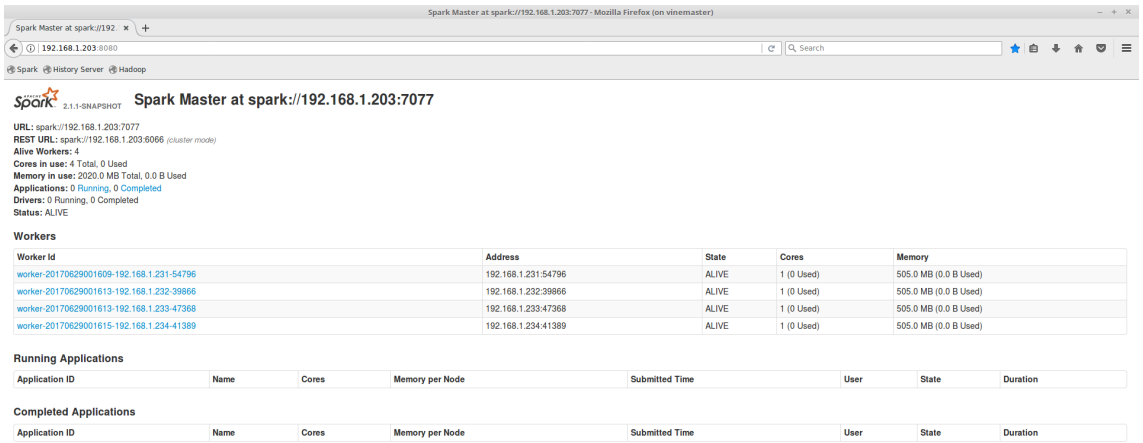


Figure 4.3: PYNQ Cluster - Spark Web UI

Before moving on to the integration of hardware accelerators to Spark's applications, it is important to note that the Hadoop framework was also deployed on the Intel based node. The main purpose of deploying Hadoop on the Spark

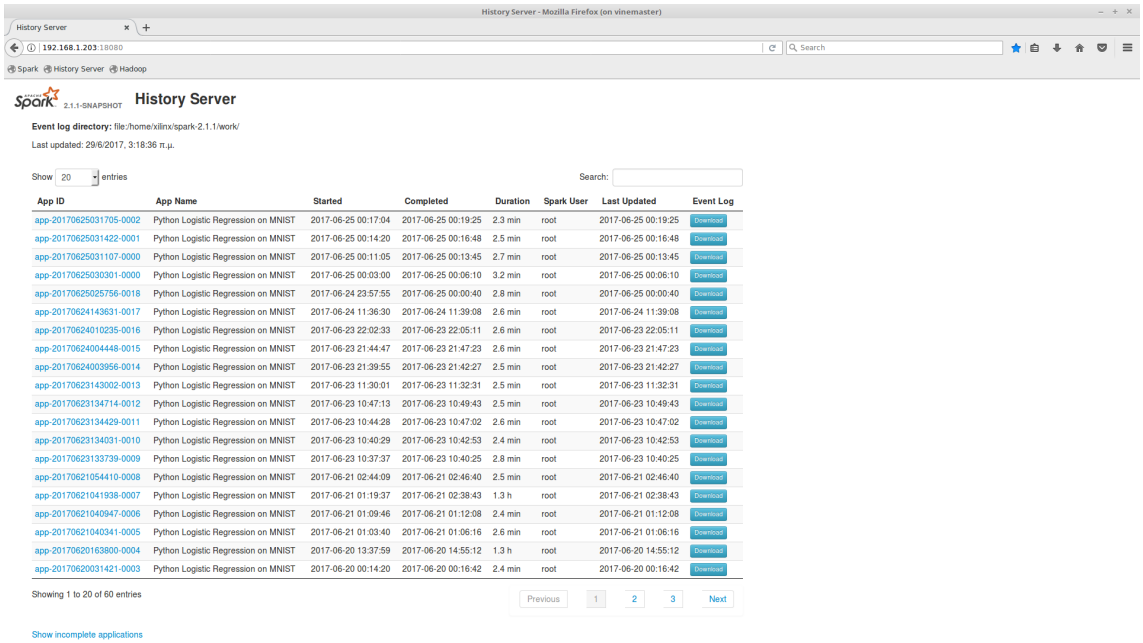


Figure 4.4: Spark History Server Web UI

Master node, was to create a a Hadoop distributed filesystem. Before creating the HDFS, we had to place all the input files given to a Spark submitted application, to the same path with that of the master (because the applications are submitted from the master node), therefore spending time in copying the files to each node. Given the HDFS, the only thing we had to do was to first place the files in the distributed filesystem and then point the application to read any data from it. Below we are going to take a closer look at all the necessary configurations for the Hadoop setup in this pseudo-distributed mode. The setup steps we followed, were based on Hadoop's documentation.

4.3.4 Hadoop Configurations

At first, he had to set the *HADOOP_HOME* variable under the */etc/environment* file and then add to the *PATH* its *bin* and *sbin* sub-folders, in the same way we did for exporting SPARK_HOME and its sub-folders. Then, we added the following lines of code to the *etc/hadoop/core-site.xml* file which is located in the Hadoop directory.

```
1 <configuration>
2   <property>
```

```
    <name>fs.defaultFS</name>
    <value>hdfs://192.168.1.203:9000</value>
  </property>
</configuration>
```

Listing 4.13: Hadoop core-site.xml configuration

We also explicitly set the *JAVA_HOME* path adding it to the *etc/hadoop/hadoop-env.sh* file, and finally included the configuration for our distributed file-system into the *etc/hadoop/hadoop-site.xml* file, as follows:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>/home/xilinx/hadoop-2.7.3/hdfs/namenode</value>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>/home/xilinx/hadoop-2.7.3/hdfs/datanode</value>
  </property>
</configuration>
```

Listing 4.14: Hadoop hadoop-site.xml configuration

Figure 4.5 depicts the Hadoop web UI along with the files we uploaded to the HDFS.

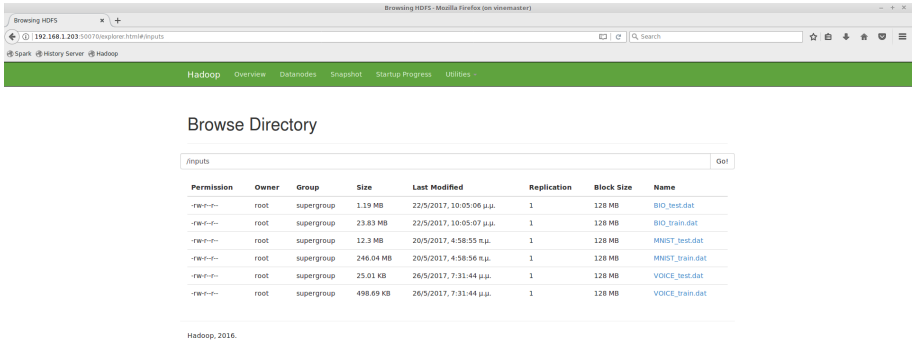


Figure 4.5: Hadoop Web UI

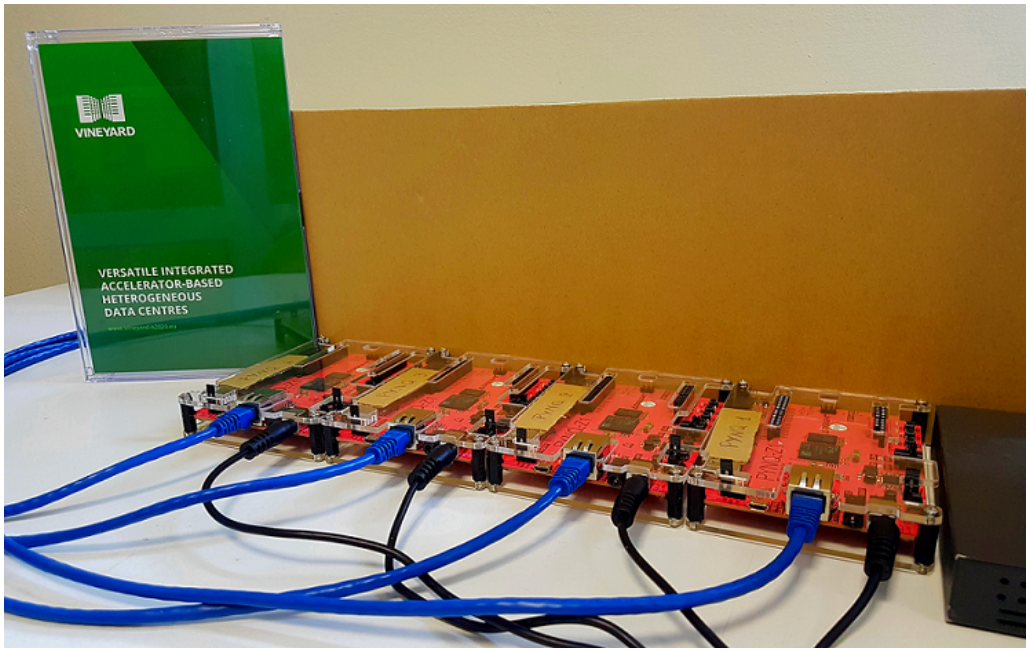


Figure 4.6: Photo of the actual PYNQ-Z1 cluster

4.4 Acceleration of Spark Applications

In this section, we are going to describe our proposed framework for the seamless utilization of hardware accelerators for Spark applications in heterogeneous FPGA-based MPSoCs as well as the development of an efficient set of libraries that hide the accelerator's details, to simplify the incorporation of hardware accelerators in Spark. Furthermore, we are going to integrate these new libraries to our already built SPynq cluster and finally we are going to evaluate the gains of using hardware accelerators for a use-case scenario on machine learning (logistic regression).

4.4.1 Related Work

In the last few years, there are several efforts for the efficient deployment of hardware accelerators for cloud computing, as well as for Apache Spark applications. In the paper entitled *A survey on reconfigurable accelerators for cloud computing* [41], a detailed survey on hardware accelerators for cloud computing applications has been presented. The survey shows both the programming framework that has been developed for the efficient utilization of hardware accelerators as well as the accelerators that have been developed for several applications like machine learning, graph computation applications and databases.

IBM also announced in 2016, the availability of SuperVessel cloud, a development framework for the OpenPOWER Foundation. SuperVessel has been developed by IBM Systems Labs and IBM Research based in Beijing. The goal of the SuperVessel cloud is to deliver a virtual environment for the development, testing and piloting of applications. The SuperVessel cloud framework takes advantage of IBM POWER 8 processors. Developers have access to Xilinx FPGA accelerators which use IBMs Coherent Accelerator Processor Interface (CAPI). Using CAPI an FPGA is able to appear to the POWER 8 processor as if it were part of the processor. Xilinx has also announced in late 2016 a new framework called Reconfigurable Acceleration Stack. The FPGA boards can be hosted in typical servers and are utilized based on application specific libraries and framework integration for the five key workloads. These include machine learning inference, SQL query and data analytics, video transcoding, storage compression, and network acceleration [42]. According to Xilinx, the acceleration stack based on the FPGAs can deliver up to 20x acceleration over traditional CPUs with a flexible, reprogrammable platform for rapidly evolving workloads and algorithms.

In the paper entitled *FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack* [43], a novel approach is presented for integrating virtualized FPGA based hardware resources into cloud computing systems with minimal overhead. The proposed framework allows cloud users to load and utilize hardware accelerators across multiple FPGAs using the same methods as the utilization of Virtual Machines. The reconfigurable resources of the FPGA are offered to the users as a generic cloud resources through OpenStack.

Finally, a relative framework called Blaze [44], was presented by *Jason Cong et al.* for the efficient utilization of hardware accelerators under the Spark framework. Their proposed scheme is based on a cluster-wise accelerator programming model and runtime system, that is portable across accelerator platforms. Blaze is mapped to the Spark cluster programming framework. The accelerators are abstracted as subroutines for Spark tasks. These subroutines can be executed on local accelerators when they are available. Otherwise the subroutines will be executed on the CPU to guarantee application correctness. The proposed scheme has been mapped to a cluster of 8 Xilinx Zynq boards, each hosting two ARM processors and a recon-

figurable logic block. The performance evaluation shows that the proposed system can achieve up to 1.44x speedup for the Logistic regression and almost the same throughout for the K-Means and 2.32x and 1.55x better energy efficiency respectively. It has been also mapped to typical FPGA devices connected to the host through the PCI interface. In this case, the performance evaluation shows that the proposed system can achieve up to 3.05x speedup for the Logistic regression and 1.47x speedup for the K-Means and reduces the overall energy consumption by 2.63x and 1.78x respectively.

4.4.2 The Spark on Pynq (SPynq) Framework

Now that we have mapped Spark on top of the PYNQ-Z1 cluster, we are ready to go through all the necessary steps for adapting it to communicate with the hardware accelerators, located in the programmable logic (PL) of the Zynq system.
Figure 4.7, depicts the software stack of our implemented setup. The Hadoop DFS is in the first level, while on top of it the Apache Spark framework is built, along with its APIs for machine learning, graph computing and other applications.

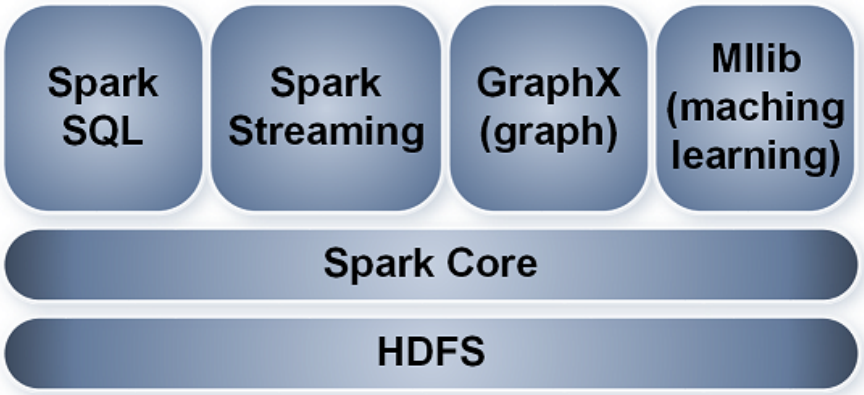


Figure 4.7: The software stack of our implemented setup

In the typical case of running a machine learning application, the application invokes the Spark MLlib, which utilizes the Breeze library. Breeze library invokes the Netlib Java framework that is a wrapper for lowlevel linear algebra tools implemented in C or Fortran. Netlib Java is executed through the Java Virtual Machine (JVM) and the actual linear algebra tools (BLAS - Basic Linear Algebra Subprograms) are executed through the Java Native Interface (JNI). All these layers

add significant overhead to the Spark applications. So the main idea of Spynq framework, is to create new packages that deliver hardware acceleration to Spark applications. In that way, the only modification needed for any Spark application, is the replacement of the old MLlib function with the new one that invokes the hardware accelerator. Figure 4.8 depicts our proposed scheme for accelerating Spark applications, where we have implemented a new MLlib package called MLlib_accel for accelerating machine learning algorithms.

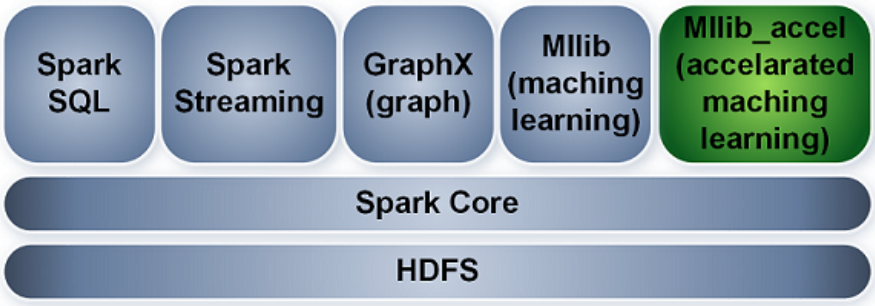


Figure 4.8: The software stack of our proposed setup for accelerating Spark applications

On the PYNQ side now, as already mentioned in section 4.2, the PYNQ project comes with a bunch of Python libraries for communicating with the programmable logic. These libraries include methods for deploying the hardware accelerators on the PL as well as whole structures and methods for handling the components of each accelerator. In example, Python libraries are provided for creating and destroying DMA (Direct Memory Access) objects as well as methods for allocating contiguous memory buffers that serve as input or output buffers for the hardware accelerator. Behind this Python API, a C API is used which is invoked for the actual communication with the hardware accelerator, therefore it serves as its driver. In other words, PYNQ provides an easy and efficient way to handle FGPA accelerators without requiring from the user deep hardware engineering knowledge and expertise. So, for every new implemented hardware accelerator, an also new Python library needs to be created that will host the lower level function calls for the communication with the PL. It is important to note that this library is independent of any given framework (e.g Apache Spark, Hadoop etc.), therefore it could be integrated into a multitude of applications. Figure 4.9, shows the

intervening stages when communicating with the hardware accelerator.

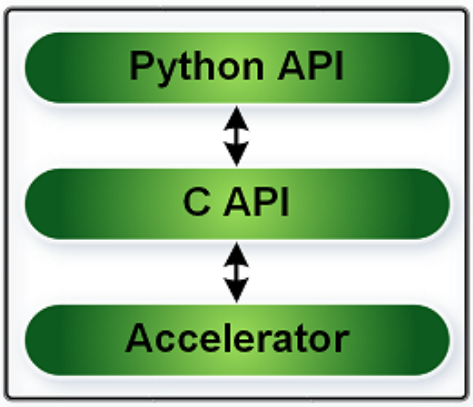


Figure 4.9: Flow diagram depicting the intervening stages for the communication with the hardware accelerator

Upon creating the Python API for the accelerator, the corresponding library for accelerating Spark's applications could be implemented. The whole stack is shown below.

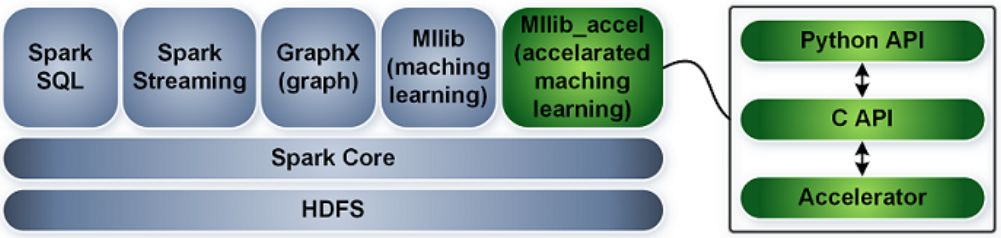


Figure 4.10: Final Spark software stack including "accelerated" libraries

Further Configurations

Before being able to evaluate our proposed framework, a few more configurations have to be made. These configurations concern both the Spark framework and the existing PYNQ's libraries.

As far as Spark is concerned, since the new libraries that invoke the hardware accelerators are written in Python, PySpark is going to be used for any submitted application. By default, Python 2 is used for PySpark. On the other hand, the libraries provided by the PYNQ project are written in Python 3, so we have to explicitly set PySpark to use Python 3. This is done by adding to every node's spark-env.sh file the following code lines:

```
export PYSPARK_PYTHON=python3
export PYTHONHASHSEED=0

# SPARK_WORKER_TYPE = 0 for CPU only, SPARK_WORKER_TYPE = 1 for CPU + FPGA
export SPARK_WORKER_TPYE={0,1}
```

Listing 4.15: Configuring PySpark to use Python 3

We then had to make sure that every node of the cluster had the same Python 3 minor version (e.g Python 3.**4**), which is necessary for Spark to operate in cluster mode.

Furthermore, we set the PYTHONHASHSEED variable. This variable stands for the hash randomization of Python and if set to an integer value, this value is used as a fixed seed for generating the hash(). Its value had to be set to zero, in order for the cluster to work properly.

Moving on, we set a new variable, the SPARK_WORKER_TYPE. This variable is used to define the system's hardware resources. For platforms that host a programmable logic (like PYNQ-Z1) it should be set to "1", while for platforms that do not, its value should be set to zero. In that way, upon launching a heterogeneous, at worker scale, Spark cluster, we are able to both invoke the hw accelerator for the nodes supporting it (i.e the PYNQ-Z1 nodes) and use the default Spark's libraries for the nodes that don't (i.e the Intel based node).

Moreover, any new implemented libraries for accelerating Spark's applications had to be transfered under its */python/pyspark* directory which is already included in the *PYTHONPATH* variable. But Spark, also holds a pyspark.zip file under its */python/lib* directory that is used upon invoking a library from the PySPark package. So for the new libraries to be accessible, the *pyspark.zip* file had to be either renamed or deleted.

Apart from that, we had to also transfer PYNQ's libraries for communicating with the hardware accelerator, to the master node. Upon submitting a PySpark application, the whole application context is constructed on the node that submitted it, and then the part of the application code that contains transformations or actions on a RDD, is executed on the workers. In that way, even the hardware accelerator's libraries should be present on the master node. But as we have already mentioned,

The Python API uses a C API to communicate with the hardware accelerator. PYNQ, provides this API by including some libraries in a shared object format (.so file) (i.e libdma.so). The problem here, is that these libraries are compiled for the 32-bit ARM architecture. Since we wanted the Spark applications to be submitted from the master node, we had to also recompile any shared object file that is being used by the accelerator's Python API for Intel's x86 and x86_64 architectures. For this reason, we recompiled the *libdma.so* and *libsds_lib.so* files that are being used in PYNQ's *dma.py* library. After recompiling them for each system architecture we had to modify the dma.py code to choose the right *.so* file, depending on the system it is ran. The final content for that part of the *dma.py* file is presented below.

```
LIB_SEARCH_PATH = os.path.dirname(os.path.realpath(__file__))

if((platform.machine()=='x86_64')):
  # load 64bit ELF
  libdma = ffi.dlopen(LIB_SEARCH_PATH + "/libdma64.so")
  libxlnk = memapi.dlopen(LIB_SEARCH_PATH + "/libsds_lib64.so")
elif (platform.machine()=='armv7l'):
  #load 32bit ELF compiled for ARM
  libdma = ffi.dlopen(LIB_SEARCH_PATH + "/libdma.so")
  libxlnk = memapi.dlopen(LIB_SEARCH_PATH + "/libsds_lib.so")
elif (platform.machine()=="i686"):
  libdma = ffi.dlopen(LIB_SEARCH_PATH + "/libdma32.so")
  libxlnk = memapi.dlopen(LIB_SEARCH_PATH + "/libsds_lib32.so")
else:
  print("Machine type not supported. Exiting!\n")
exit(1)
```

Listing 4.16: Modification in dma.py (1) - Support for Intel x86 and x86_64 architectures

Table 4.3, shows all of the available .so files after the recompilation process we followed, along with the architecture for which they are built for. The libsds_lib.so file contains the C API for handling memory requests(i.e allocating or freeing memory buffers) while the libdma.so file contains the C API for the communication with any DMAs in the hardware accelerator.

.so File Name	System Architecture
libdma.so	ARM 32-bit
libdma32.so	Intel x86
libdma64.so	Intel x86_64
libsds_lib.so	ARM 32-bit
libsds_lib32.so	Intel x86
libsds_lib64.so	Intel x86_64

Table 4.3: Available .so files after the recompilation process

Furthermore, we had yet another challenge with the dma.py file, which is used for the communication with the hardware DMAs (if any). In more detail, when a DMA object is created, a buffer can be allocated for it. Then by invoking the corresponding call (get_buf()), a *CFFI* pointer to object's internal buffer is returned. This could only be of either *integer* or *long long* data type. But most applications in Spark, work with floating point data. In that way, we had to modify the "*get_buf()*" method to also return a CFFI pointer of float data type. The final form of the method is presented below.

```
def get_buf(self, width=32, data_type = 'int'):
  """Get a CFFI pointer to object's internal buffer.

  This can be accessed like a regular array in python. The width can be
    either 32 or 64.

  Parameters
  ----------
  width : int
  The data width in the buffer.
  data_type : string
  The type of the returned pointer

  Returns
  -------
  cffi.FFI.CData
  An CFFI object which can be accessed similar to arrays in C.

  """
  if self.buf is not None:
```

```
20      if width == 32:
21        if data_type == 'int':
22        return ffi.cast("int *",self.buf)
23      elif data_type == 'float':
24        return ffi.cast("float *",self.buf)
25      else:
26        raise RuntimeError ("Not supported type")
27    elif width == 64:
28      return ffi.cast("long long *",self.buf)
29    else:
30      raise RuntimeError("Buffer not created.")
```

Listing 4.17: Modification in dma.py (2) - Add functionality to retun a pointer of float data type

Finally, Spark uses serializers for sending or receiving data from workers. To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object [45]. By default Spark uses Pickle serializer for PySpark, while it includes a few more serializers (i.e Marshall, Cloudpickle etc.). Of course, a user can write his own serializer and port it to Spark. The challenge here, is again related to the DMA library. The DMA objects that are being created from PYNQ's library have a complex structure that Spark cannot serialize or deserialize. In fact, the Spark can't serialize or desirialize the CFFI objects that are being used for the DMA's buffers. At this point, we had to choose between two possible solutions: either create a new serializer or have the buffers of the DMAs created outside the DMA objects.

The first solution, seemed to have the most disadvantages. Apart from spending more time in writing a new serializer for the DMA objects, a user should always explicitly set our serializer by creating a SparkConf object and passing it to it. Apart from that, the serialization and deserialization process of the DMA objects could be time consuming, while all of these serialized data would be transfered over the network. It is now made clear, that this solution would not help the acceleration process since it would add a huge overhead to the execution time of an application.

On the other hand, the solution of creating the buffers outside the DMA objects seemed promising. The main idea is simple enough. At first, the buffers for the

data of the DMA objects are created and then, upon creating the DMA objects the buffers are assigned to them. Before returning any data to the master node, the buffers are dis-assigned from the DMA objects and instead of returning the whole object structure only the virtual and physical addresses of the buffers are returned along with their size into a new RDD. In that way, the data of the buffers are persisted in memory on the worker nodes, therefore implementing a similar to Spark's method for persisting the RDDs on the workers. A more detailed example will be given at the following section, where a use case scenario on Logistic Regression is presented.

Now that we have finished with the steps necessary to accelerate a Spark application, we can take a look to the whole architecture of our cluster (figure 4.11).

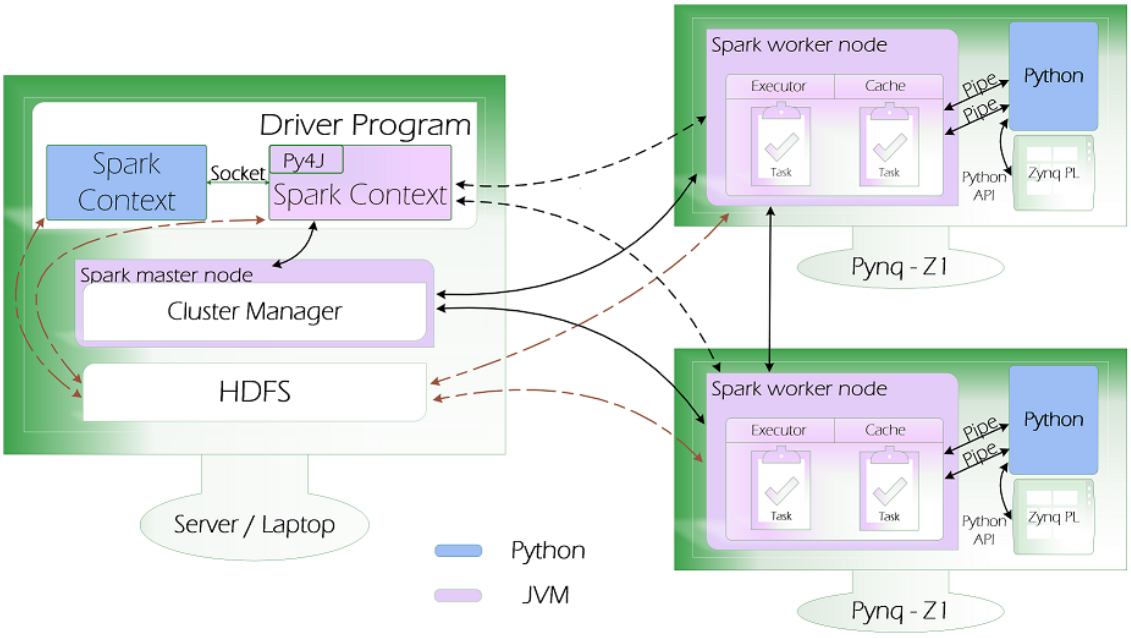


Figure 4.11: Final architecture of the implemented cluster

As we have already mentioned, Spark Master is deployed on the Intel based Node, that also hosts the Hadoop filesystem, while Spark Workers are hosted on the PYNQ-Z1 boards. Since our applications are written in Python PySpark is used for their deployment. PySpark is built on top of Spark's Java API, so data is processed in Python and cached / shuffled in the JVM. Upon submitting a Python application, in the driver program a Python SparkContext is created and the the driver uses Py4J to launch a JVM and create a JavaSparkContext. Py4J is

only used on the driver for local communication between the Python and Java SparkContext objects.

RDD transformations in Python are mapped to transformations on PythonRDD objects in Java. On remote worker machines, PythonRDD objects launch Python subprocesses and communicate with them using pipes, sending the user's code and the data to be processed. So in our particular case, when the data are received to the Python subprocesses the hardware accelerator is invoked through the Python API. Figure 4.12 shows in more detail the communication and the data flow for a given task on the worker side.

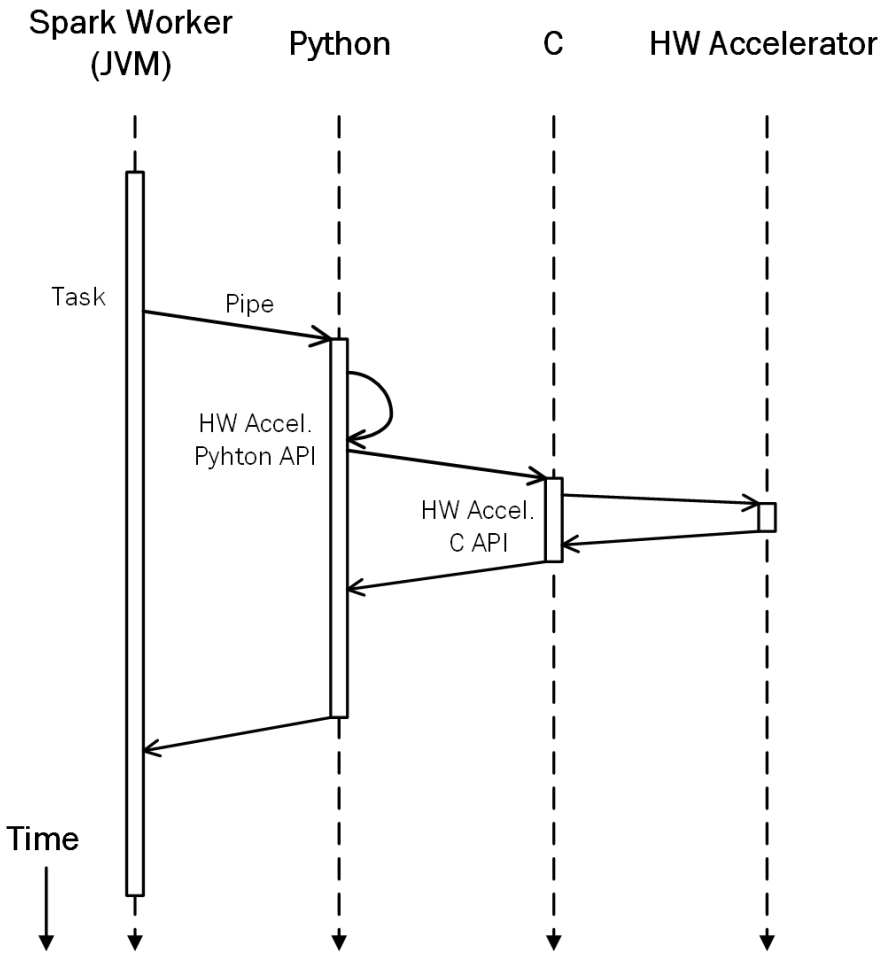


Figure 4.12: The data-path on the worker side.

The worker is active for the whole time and waits to receive some piece of work. When a task arrives, an executor is launched (into the worker) and then the data is sent to Python subprocesses. Upon using the hardware accelerators, the Python API is invoked which then uses the C API to communicate with the programmable

logic. The data is returned to the Python subprocess through the same path and are sent back to the worker.

4.4.3 A Use Case Scenario on Logistic Regression

To evaluate the proposed framework, a hardware accelerator was developed for Logistic Regression (LR) training with Batch Gradient Descent (BGD) and more specifically for the gradients kernel. The accelerator was implemented as part of the thesis entitled "*FPGA-Acceleration of Machine Learning in Cloud Computing, a case study using Logistic Regression*" written by Elias Koromilas [46]. For any further implementation or other information on the accelerator please refer to the corresponding thesis and its author.

Figure 4.13 depicts the block diagram of the logistic regression overlay. The driver is used to send the parameters through the AXI interface to the hardware accelerator. In this example, four different channels are used for the communication between the ARM and the accelerator; two channels are used for sending the data and one channel is used for sending the weights. One more channel is used to receive the results of the accelerator (gradients). We would like to note here that AXI4-Stream Accelerator Adapter IP is also used (it does not appear in the figure), in order to make the size of the data chunks variable.

Finally, to speedup the execution time, the programmable logic hosts two copies of the kernels that can run in parallel. Each kernel consists of four blocks that are used to calculate the gradients and are pipelined to increase the overall throughput. In Spark *gradients_kernel* can be parallelized using Map-Reduce, so partial gradients are computed in each Worker, using different chunks of the training set, and then the Master aggregates them and updates the *weights*.

The Spark code for the utilization of the hardware accelerator through our accelerated machine learning library is shown in listing 4.18. When the Spark user wants to utilize the hardware accelerator, the only change that needs to be done is the replacement of the Spark *mllib* library with the *mllib_accel*. Therefore, the user can speedup the execution time of the Spark application with a simple replacement of the libraries that he wish to accelerate.

```
1 from pyspark import SparkContext
```

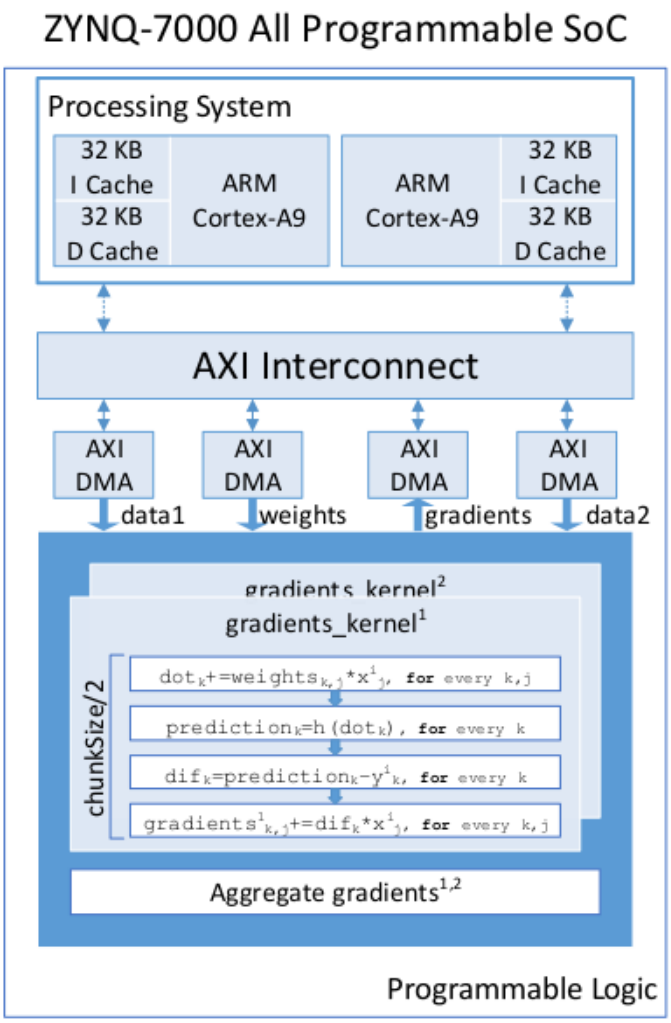


Figure 4.13: Acceleration of Logistic Regression in Spark using PYNQ-Z1's FPGA [46]

```
from pyspark.mllib.regression import LabeledPoint
from mllib_accel.classification import LogisticRegression

...

sc = SparkContext(appName = "Python Logistic Regression")

trainRDD = sc.textFile(train_file, numPartitions).map(parsePoint)
testRDD = sc.textFile(test_file, numPartitions).map(parsePoint)

LR = LogisticRegression(numClasses, numFeatures).train(trainRDD, alpha,
    numIterations, _accel_)
LR.test(testRDD)
```

```
14
15 sc.stop()
```

Listing 4.18: Spark code for the utilization of the hardware accelerator

More specifically, in Python, a LogisticRegression object is created and various methods are supported (train, test, predict etc.). Each required action is passed in a map statement which is followed by a corresponding reduce or collect action. For example, in method train of the LogisticRegression object, *gradients_kernel* is mapped to all available workers and then on the workers' side, the accelerator's specific library is called to take advantage of the programmable logic.
After profiling the application, we concluded that most of the time (99,2%) is spent on writing the train data to the allocated DMAs' buffers. Since they remain the same over the whole execution of the logistic regression training, we managed to implement a novel scheme that allows the persistent storing of the data throughout the invocation of the accelerator. We implemented a new function that returns the allocated buffers for the corresponding data needed for the accelerator. In that function, also the train data are written into the buffers and remain there for the rest of the execution of the application. At each iteration of the algorithm, DMA objects are created and the previously created buffers are assigned to them. Before the return command of each iteration,the buffers are dis-assigned from the DMAs and the DMA objects are destructed.
Based on the above, we have created a Python API which basically consists of three calls:

- **cma (contiguous memory allocate):** This call is used for the creation of the buffers and the further allocation of contiguous memory. Also at this point overlay is downloaded and train data are written to the corresponding buffers. Using cma, a new RDD, which contains only information about these buffers (memory addresses, sizes, etc.), is created and persisted.

- **gradients_kernel_accel**: In this call, the DMA objects are created using Xilinx's built-in modules and classes. The previously allocated buffers are assigned to the DMAs, while current weights are written to memory and finally the data are transferred to the programmable logic. Gradients are

computed in return, buffers are dis-assigned from DMAs and the last ones are destructed.

- **cmf (contiguous memory free):** This call is explicitly used to free all previously allocated buffers (RDD unpersist).

The source code of the Python API as well as of the new Spark library, can be found in appendix A′.2. As it can be seen, the accelerator's API is Spark independent and can be used in any Python application.

5 Results

5.1 Results of Execution on Processor

In this first section, we are going to present the results of our evaluation using Spark's built-in libraries. We have measured the execution time and the energy efficiency of the applications we described in section 3.3 on the embedded platforms of section 3.2. It is important to note that these applications are written in Scala which is much faster than Python. We also measured the execution time and the energy efficiency of a mainstream server, that comes with a typical high-performance Intel Xeon processor and finally the execution time and efficiency of a commodity processor for laptops, based on the Intel i5 processors family. In this way, we are able to provide a good comparison on how each platform stands against each other and to finally draw conclusions for each case.

To make a fair comparison between the evaluated systems, we have executed all of the applications in Spark's local mode, where 4 cores were used and the available memory for the executor and the driver memory of Spark was set to 800MB respectively. An exception to this, is the PYNQ-Z1 platform which only has 2 cores and 512MB of RAM available. In this case, both cores were used while the memory for the execution of Spark's applications was set to 505MB. This final comparison may be unfair, but as we have already mentioned, the main purpose of including Pynq in this evaluation is to later (in chapter 4) project the gains of integrating the use of hardware accelerators with Spark. So in this evaluation, only the CPU cores of the PYNQ-Z1 were used.

Furthermore, the current performance evaluation does not measure the performance of the processors on the Spark nodes but evaluates the performance of the

whole Spark framework. The features of the operating system and the processors of each platform are presented in the next Table (5.1).

Features	Server	Laptop	Raspberry Pi 3	DragonBoard 410c	PYNQ-Z1
Vendor	Intel	Intel	Broadcom	Qualcomm	Xilinx
Device	E5-2650	i5-430M	BCM2837	Snapdragon 410	Zynq XC7Z020
Cores(threads)	8(16)	2(4)	4	4	2
Processor	E5-2650	i5-430M	A53	A53	A9
Architecture	64-bit CISC	64-bit CISC	64-bit RISC	64-bit RISC	32-bit RISC
Process	22nm	32nm	40nm	28nm	28nm
Clock Frequency	2.6GHz	2.26GHz	1.2GHz	1.2GHz	667MHz
Level 1 cache	512kB	128kB	32kB	32kB	32kB
Level 2 cache	2048kB	512kB	512kB	512kB	512kB
TDP	95W	35W	NA	3.7W	NA
Operating System	CentOS	Ubuntu	Debian	Debian	Ubuntu

Table 5.1: Main features of the evaluated platforms.

For the overall evaluation, a bash script was created that invoked the execution of each individual application. Because of the applications being executed in a real system, the time measured varied from execution to execution. Therefore, we modified the script to execute five times each application, thus returning the mean execution time for each case. The time for each application was measured by invoking Linux's built-in *time* function and refers to its total execution time including this for launching the Spark interface. Furthermore, upon running some tests, we realized that in the embedded platforms, in some cases the most time was spent in launching Spark rather than in computing the algorithm's outcome. For this reason, we modified the values of the applications' default input arguments (i.e. in iterative algorithms we increased the number of the iterations etc.) to "force" the algorithmic part in spending more time compared to the time needed for launching Spark's components. The arguments we used for each application are shown in the Tables 5.2 and 5.3.

	Linear Regression	Logistic Regression	KMeans
Application File	LinearRegressionExample.scala	LogisticRegressionExample.scala	KMeansExample.scala
maxIter	100	100000000	-
regParam	0.001	0.001	-
elasticNetParam	0.9	1.0	-
tol	1.0E-20	1.0E-20	-
Input Data File	sample_linear_regression_data.txt	sample_libsvm_data.txt	sample_kmeans_data.txt

Table 5.2: Input arguments for ML family applications.

	CC	**Pagerank**	**Triangles**
Application File	Analytics.scala (cc)	Analytics.scala (pagerank)	Analytics.scala (triangles)
numEPart	20	20	20
Input Data File	pagerank_data.txt	pagerank_data.txt	pagerank_data.txt

Table 5.3: Input arguments for graph family applications.

Input Argument	**Meaning**
Application File	The source file of the evaluated application
maxIter	The maximum number of iterations the algorithm will run
regParam	The regularization parameter
elasticNetParam	The ElasticNet mixing parameter
tol	The convergence tolerance of iterations
numEPart	The number of edge partitions
Input Data File	The name of the dataset file given as input to the application

Table 5.4: Glossary for applications' input arguments.

5.1.1 Performance Results

The results for each application are presented below:

Features	**Server**	**Laptop**	**Raspberry Pi 3**	**DragonBoard 410c**	**PYNQ-Z1**
Logistic Regression	19.90	38.59	191.02	235.61	347.39
Linear Regression	7.72	17.15	59.50	88.90	77.46
KMeans	6.40	13.47	43.93	65.77	71.58
Pagerank	32.00	62.95	273.05	383.98	723.38
CC	6.44	15.20	55.12	78.25	83.19
Triangles	5.47	14.65	48.63	70.11	69.41

Table 5.5: Execution time in seconds for each evaluated platform and each application

Figure 5.7 depicts the execution time of the applications for each platform, normalized to the server's execution time. As it is shown there, the total execution time of the applications running on the low-power SoC of the embedded platforms is 6.2x to 13x higher than that on the Xeon processor. In more detail, the Raspberry Pi 3 is 6.2x to 9.6x times worst in terms of total execution time, while the DragonBoard 410c is 10.3x to 12.8x worst in the same field. PYNQ-Z1, although being the one having the most limited hardware resources, stands well against the other

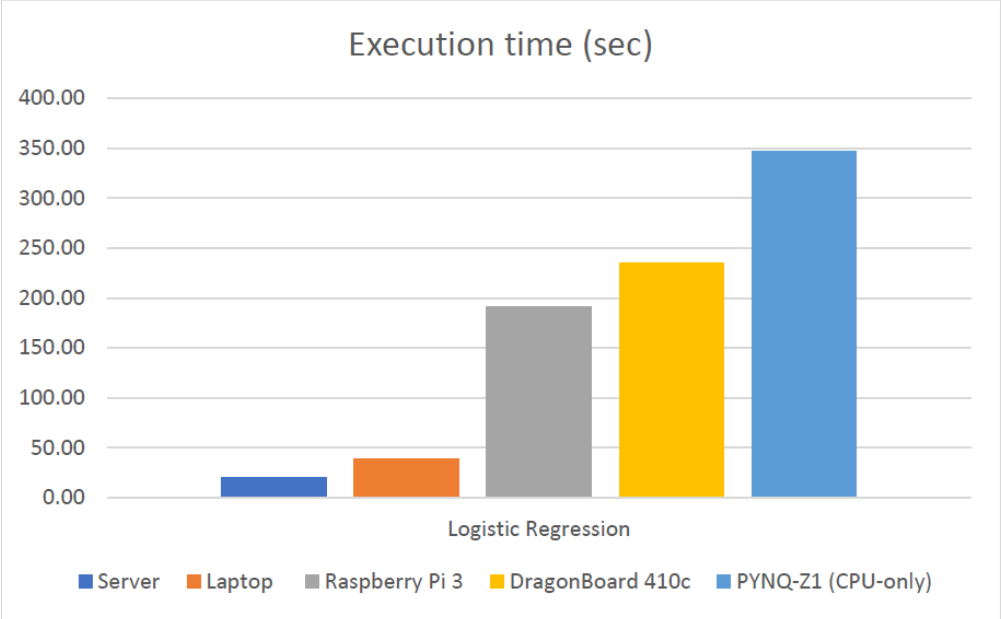


Figure 5.1: Execution time for Logistic Regression

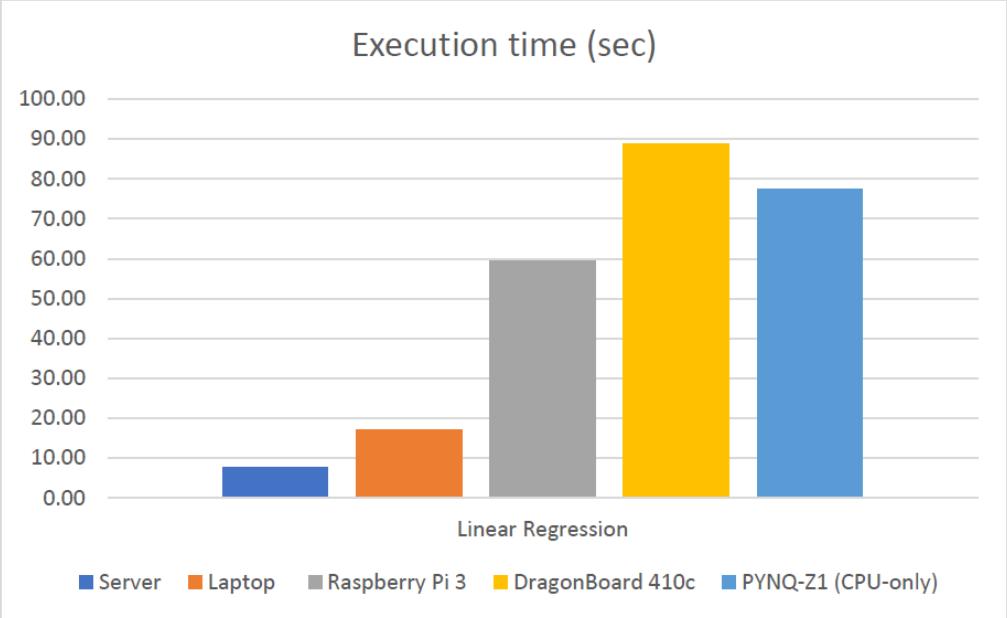


Figure 5.2: Execution time for Linear Regression

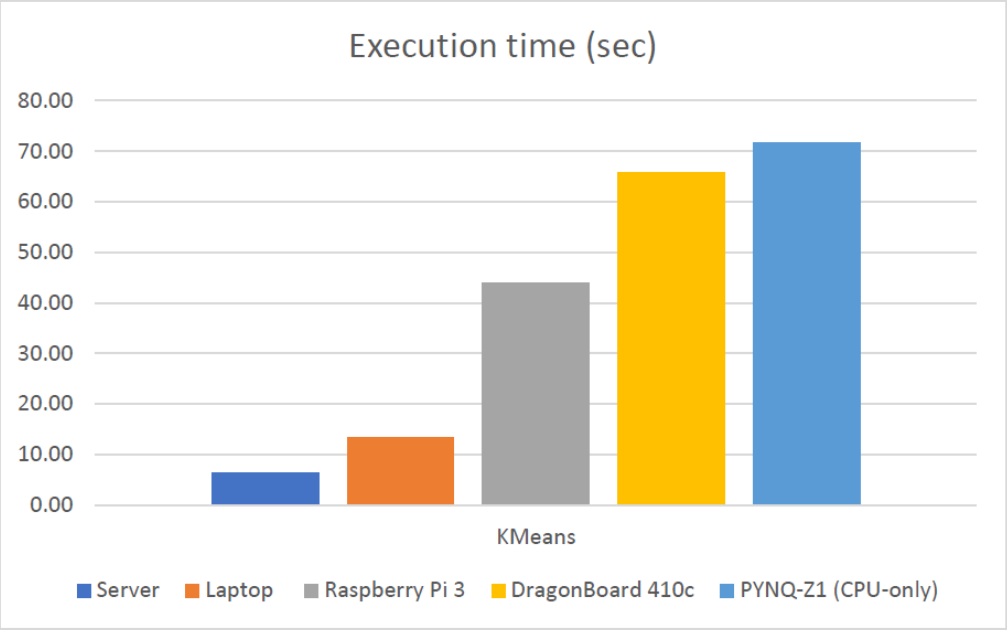


Figure 5.3: Execution time for KMeans

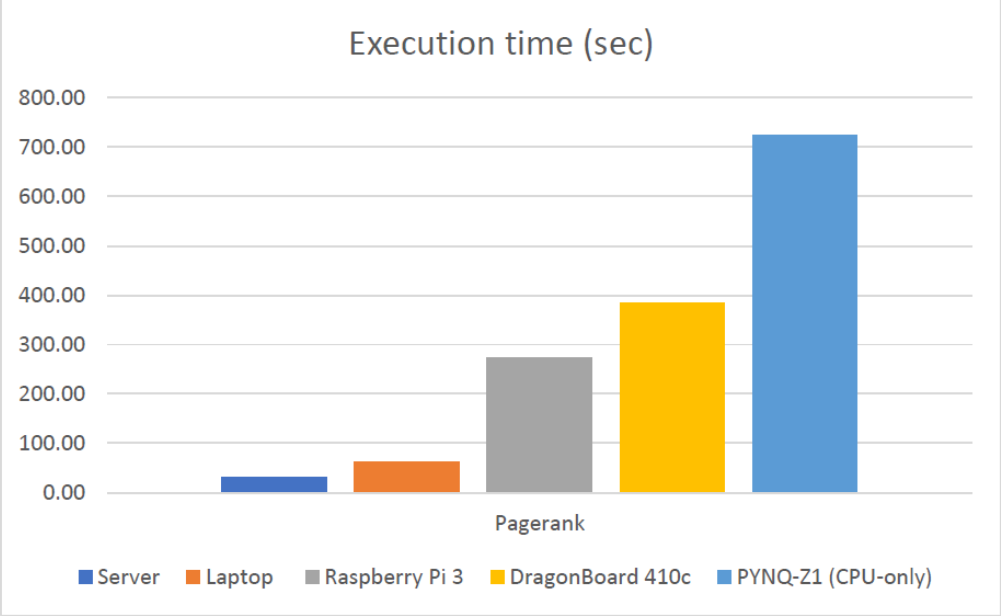


Figure 5.4: Execution time for Pagerank

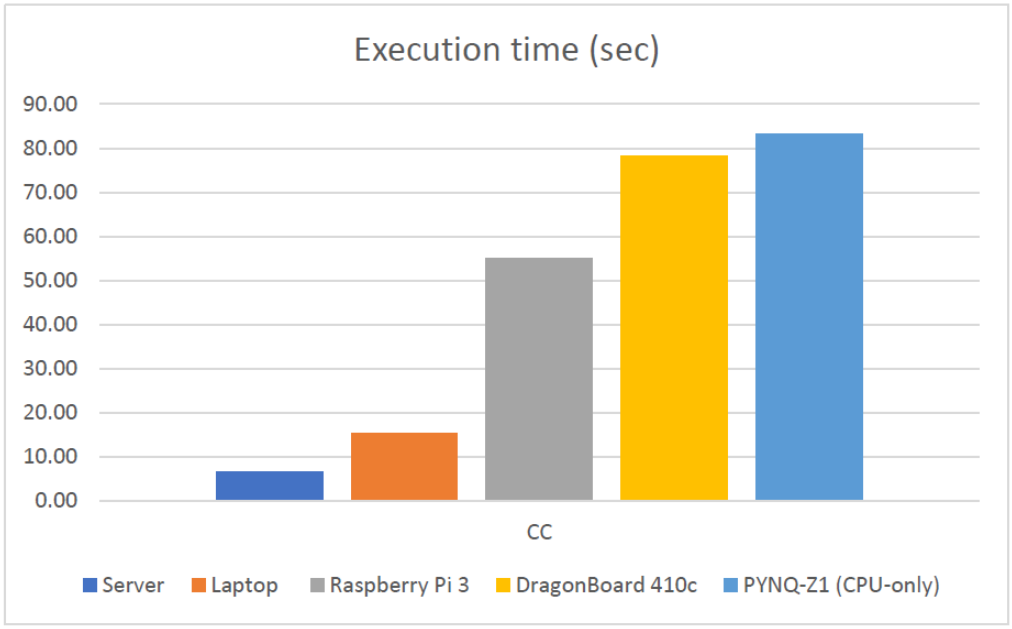


Figure 5.5: Execution time for Connected Components

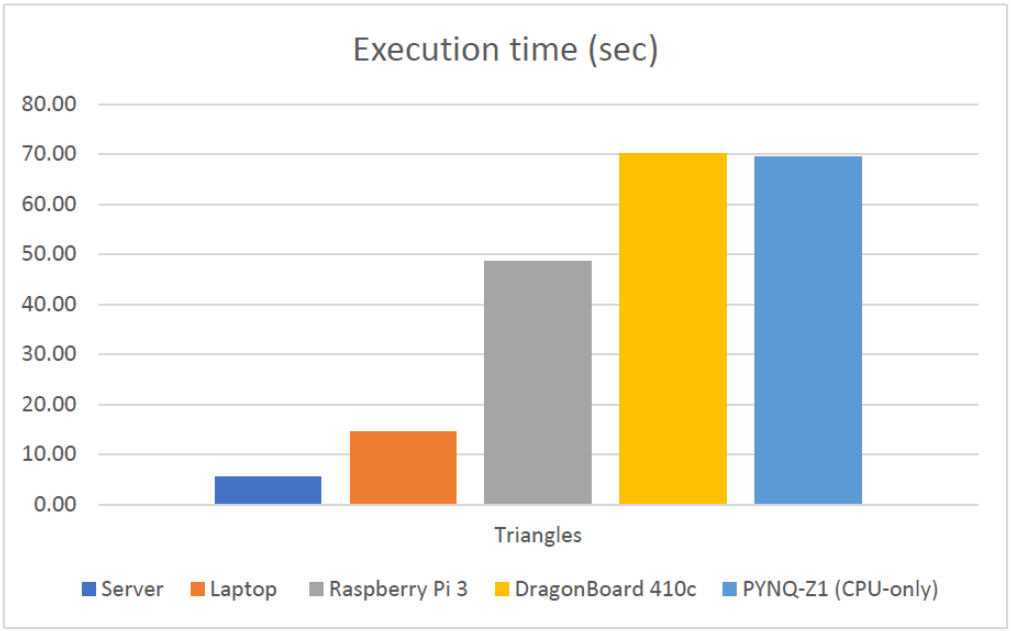


Figure 5.6: Execution time for Triangles

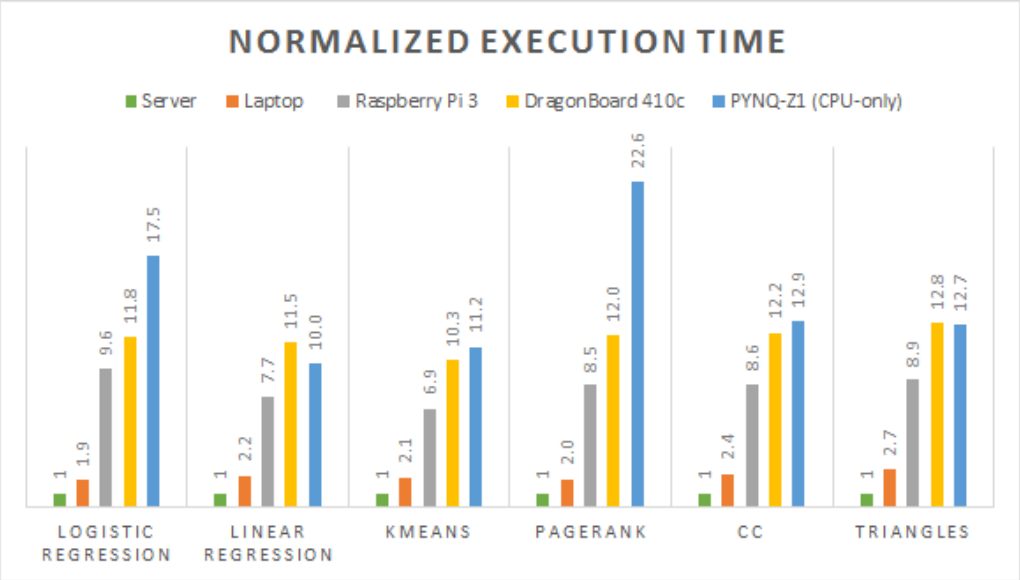


Figure 5.7: Comparison of the execution time for the Spark applications. The execution times are normalized to the Intel Xeon platform

embedded devices by being only 14x times slower in average. In fact, in the four out of six applications, PYNQ-Z1 performs almost the same as the DragonBoard 410c does.

Features	Server	Laptop	Raspberry Pi 3	DragonBoard 410c	PYNQ-Z1
Logistic Regression	1	1.9	9.6	11.8	17.5
Linear Regression	1	2.2	7.7	11.5	10.0
KMeans	1	2.1	6.9	10.3	11.2
Pagerank	1	2.0	8.5	12.0	22.6
CC	1	2.4	8.6	12.2	12.9
Triangles	1	2.7	8.9	12.8	12.7

Table 5.6: Execution time normalized to the time of the server for each evaluated platform and each application

It is very interesting to note that although the Raspberry Pi 3 and the DragonBoard 410c are using the same 64-bit A53 processors, which are clocked at the exact same frequency of 1.2GHz, there is an overall higher execution time in the Snapdragon platform. In more detail, the average execution time of the Raspberry platform based on the Broadcom SoC is 8x longer than the Xeon processor while the execution time of the DragonBoard platform based on the Snapdragon SoC is 11x longer than the Xeon processor. The prevailing reason why DragonBoard is

worse than the Raspberry in terms of performance, could be that the RAM at the Snapdragon platform is clocked at a much lower frequency of 533MHz than this of the Pi 3 which is clocked at 900MHz. Spark is so fast compared to other big data analytics frameworks, because of its abilotu to run computations on memory, so the speed and the amount of a platform's available memory, are a determining factor for Spark's overall performance. Another reason concerning the performance of the two platforms, could be the different process in which each SoC is fabricated. As we have already mentioned, the Broadcom SoC is manufactured at a 40nm process while the Snapdragon SoC is manufactured at a 28nm process. Finally, it is important to point out that the operating system of each platform is not tha same. Both operating systems are based on the debian distribution, but Raspbian is a 32-bit OS, while the 410c's OS is a 64-bit one. Of course, the degree to which each operating system is optimized for each platform plays also a very significant role for each platform's overall performance.

5.1.2 Energy Efficiency Results

As far as the energy consumption is now concerned, the main advantage of the SoCs is that they are optimized for low power consumption. In this section we evaluate the energy efficiency ($power * execution_time$) for the evaluated platforms when running Apache Spark. Apart from the Xeon processor that has bult-in tools for monitoring in detail its energy consumption and gives the option to take consumption metrics for specific running processes, all of the other platforms don't have any kind of embedded tool for doing so. This leaves us with only two choices: we could either place a multimeter in series with the power supply to measure the current it draws and then multiply it by the supply voltage to compute the consumption in Watts, or measure the energy consumption base on the TDP (Thermal Density Power) features of the processors. But the first approach needs a more complex setup and since we did not have all the required tools for taking the measures in this way, we decided to go with the second solution. Therefore, this evaluation for the SoCs' energy consumption, is based on the TDP features of the processors. TDP refers to the maximum dissipated power of the processors when working at a full load. It is important to note here that the comparison on

the energy efficiency between the platforms is just indicative and is used as a first approximation for the potential energy savings based on the SoCs.

Before advancing to this subsection's results, we should point out a few things about the Raspberry Pi3 and PYNQ-Z1 platforms. As we saw in Table 5.1 we were not able to find the TDP feature of their processor. For the Raspberry Pi 3, we were able to find benchmarks that proved that even in full load the whole board wouldn't draw more than 0.6 Amperes [29]. Assuming a power supply of 5V input voltage was used, we get a consumption of just $0.6A * 5V = 3W$. To be fair enough we made a pessimistic assumption that the TDP of the Broadcom SoC is about 4W. As far as the PYNQ-Z1 board is now concerned, we measured the consumption of an equivalent (to the first) board called ZedBoard which provided tools for monitoring its consumption. The latter platform, has the exact same Zynq SoC as the PYNQ-Z1. The measured consumption even when using Zynq's programmable logic (the FPGA) did not go over 3.2W, so we again make a pessimistic assumption that Zynq's TDP is 3.5W.

We are now ready to move on the energy efficiency results of this evaluation. Figure 5.8 depicts the normalized energy efficiency of the Spark for the six evaluated platforms.

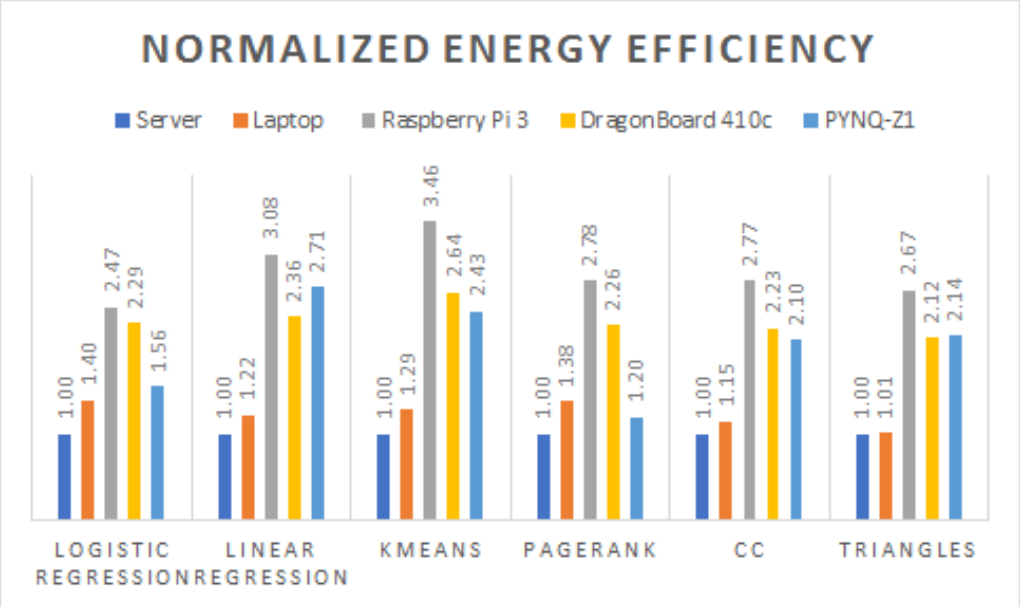


Figure 5.8: Comparison of the energy efficiency. The energy efficiency is normalized to the Intel Xeon platform

As it is shown in the above figure, the energy consumption of the low-power SoC is 2-3.5x better than the energy consumption based on the Intel Xeon processor. While the latter platform comes first in performance terms, the power it draws causes an overall high energy consumption. On the other hand, the low-power SoCs provide a much greater energy efficiency at cost of performance. It is now obvious, that in cases the energy efficiency is the main criteria upon choosing a computing platform, the low-power SoCs are undoubtedly the best choice.

Features	Server	Laptop	Raspberry Pi 3	DragonBoard 410c	PYNQ-Z1
Logistic Regression	1	1.4	2.47	2.29	1.56
Linear Regression	1	1.22	3.08	2.36	2.71
KMeans	1	1.29	3.46	2.64	2.43
Pagerank	1	1.38	2.78	2.26	1.2
CC	1	1.15	2.77	2.23	2.10
Triangles	1	1.01	2.67	2.12	2.14

Table 5.7: Energy efficiency of each platform, normalized to the energy efficiency of the Intel Xeon platform

5.2 Results of Hardware Accelerated Execution

In this section of the results, we are going to evaluate the performance of the SPynq framework we proposed in chapter 4, for the use case scenario on Logistic Regression of the subsection 4.4.3. As a case study, we built a classification model with 784 features and 10 labels using 40k available training samples, for a handwritten digits recognition problem.

To evaluate the performance of the SPynq cluster,we also built a Spark cluster consisting of four worker nodes that run on a server's Xeon cores. Table 5.8 shows the features of the two platforms.

Spark executor JVM processes consume most of the available 512 MB RAM on PYNQ-Z1, placing a restriction on our application, which requires main memory to cache and repeatedly access the working dataset from FPGA's offchip RAM once read from HDFS.

On the other hand, the Xeon system has a 12 core CPU with a total of 24 threads. As already mentioned, only 4 out of the 24 threads are being used by the Spark cluster in order to make a fair comparison with the 4 nodes of the Pynq cluster.

Features	Server	PYNQ-Z1
Vendor	Intel	Xilinx
Device	E5-2658	Zynq XC7Z020
Cores(threads)	12(24)	2
Processor	E5-2658	A9
Architecture	64-bit	32-bit
INstruction Set	CISC	RISC
Process	22nm	28nm
Clock Frequency	2.2GHz	667MHz
Level 1 cache	380kB	32kB
Level 2 cache	3072kB	512kB
Level 3 cache	30MB	-
TDP	105W	3.5
Operating System	Ubuntu	Ubuntu

Table 5.8: Main features of the evaluated Xeon and Zynq Platforms

We also compare the accelerated platform with a CPU-only execution on the ARM cores for embedded applications where only embedded processors can be used and high performance processors like Xeon cannot be supported due to strict power constraints.

The latency in the communication between the processor and the programmable logic, especially in cases that is frequent and bidirectional, can be a major overhead and may diminish the speedup that the accelerator provides. However, in applications where the processor sends a bulk amount of data (e.g. through the AXI streaming interface), the communication overhead is overlapped by the computation time. In our case now, of the logistic regression with BGD, the processor needs to send a large amount of data to train the model and therefore the communication overhead is overlapped by the computation time. A simple DMA, however, cannot handle more than 8 MB (2600 data lines in our case) in a single transfer. This means that by splitting each partition of the RDD into chunks between 4k and 5.2k lines (we have 2 simple DMAs available for the data transfer) we can exploit our accelerator to the maximum.

5.2.1 Performance Results

Below, the results of the performance evaluation are shown in terms of execution time.

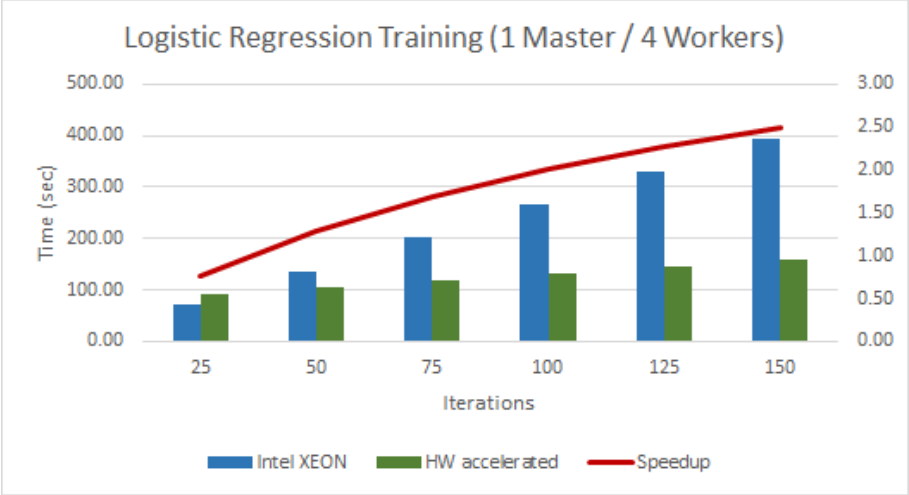


Figure 5.9: Speedup versus the number of iterations, using the proposed Python API (1)

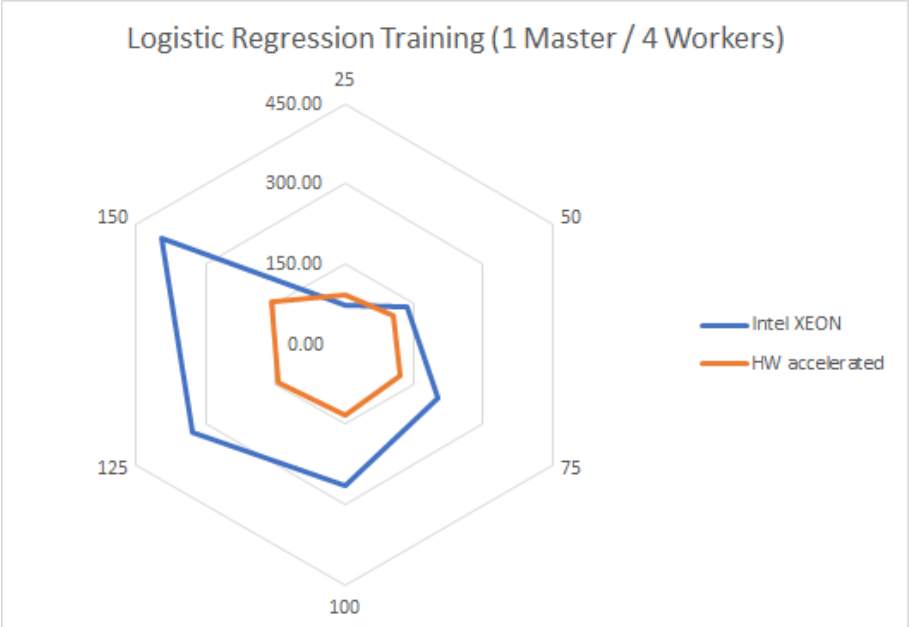


Figure 5.10: Footprint of the execution time, using the proposed Python API (1)

Figure 5.9 depicts the execution time of the logistic regression application running on a high-performance x86_64 Intel processor (Xeon E5 2658) clocked at 2.2 GHz and on a Pynq cluster which makes use of the programmable logic, for an input

dataset of 40000 lines splitted in chunks of 5000 lines for various numbers of iterations. As it is shown, the acceleration factor is equivalent to the number of the iterations. Figure 5.10 depicts the footprint of the execution on each Sprak cluster, using the same results.

In more detail, in the PYNQ-Z1 boards the data extraction, takes about 81 sec while every iteration of the algorithm is completed in 0.53 sec since the train input data is already cached into the previously allocated buffers. On the other hand, Xeon reads and transforms the data in only 7.5 sec, but every BGD iteration takes about 2.6 sec.

So the speedup actually depends on the number of iterations that are performed. For the specific application we can achieve up to 91.5% accuracy with 100 iterations in which we achieve up to 2x system speedup compared to the Xeon processor. However, there are applications in which much higher number of iterations are required. In that case, also much higher speedup can be achieved.

Worker	**Data Extraction**	**BGD Algorithm Computations (per iteration)**
Xeon	7.5	2.6
ARM	78.4	46.6
Pynq (ARM+FPGA)	80.5 (ARM)	0.51 (FPGA)

Table 5.9: Execution time (sec) of the functions executed in the workers

Table 5.9 shows the execution time of the two main functions that are executed on the worker nodes. In the Xeon platform and the ARM case, both the data extraction and the BGD are executed on the processors, while in the Zynq platform the data extraction is executed on the ARM core and the BGD function is executed on the programmable logic (accelerator). Therefore, for applications where a greater number of iterations is required the system speedup (vs Xeon) converges to 5x.

Figure 5.11 shows the speedup and the execution time of the accelerated platform compared to the software only solution running on the same cluster but using only the ARM processors, while figure 5.12 depicts in another view the footprint of the execution time. In this case, we can achieve up to 36x speedup compared to the software only case. This comparison is useful for applications in which high-performance processors cannot be used due to power limitations (e.g. embedded systems).

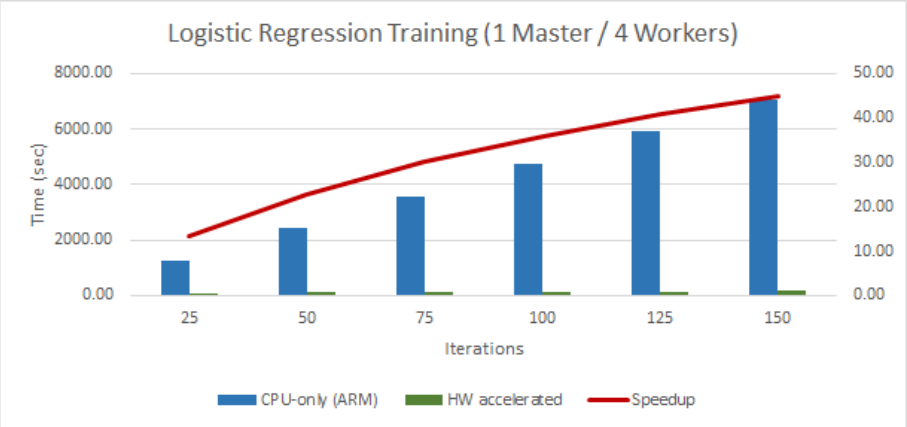


Figure 5.11: Speedup versus the number of iterations, using the proposed Python API (2)

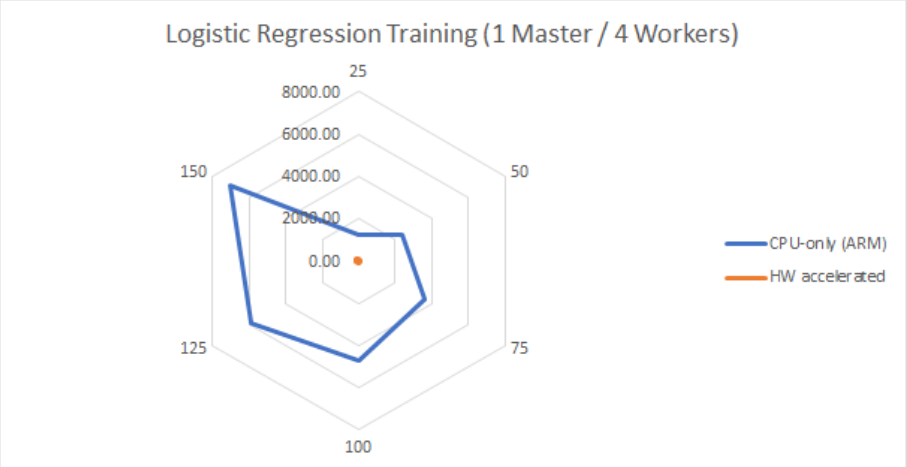


Figure 5.12: Footprint of the execution time, using the proposed Python API (2)

5.2.2 Energy Efficiency Results

To evaluate the energy savings we measured the average power, running the algorithm both in the CPU-only (for the Xeon and the ARM processors), and the HW accelerated case. In order to measure the energy consumption of the Xeon server, Intel's Processor Counter Monitor (PCM) API was used. Among others, PCM API enables capturing the energy consumed by the CPU and DRAM memory for executing an application. We also measured the power consumption using the ZC702 Evaluation board, which hosts the same Zynq device as the PYNQ-Z1 board, using the on-board power controllers.

Figures 5.13 and 5.14 show the energy consumption between the Xeon and the Zynq platforms, while figures 5.15 and 5.16 show the energy consumption for the

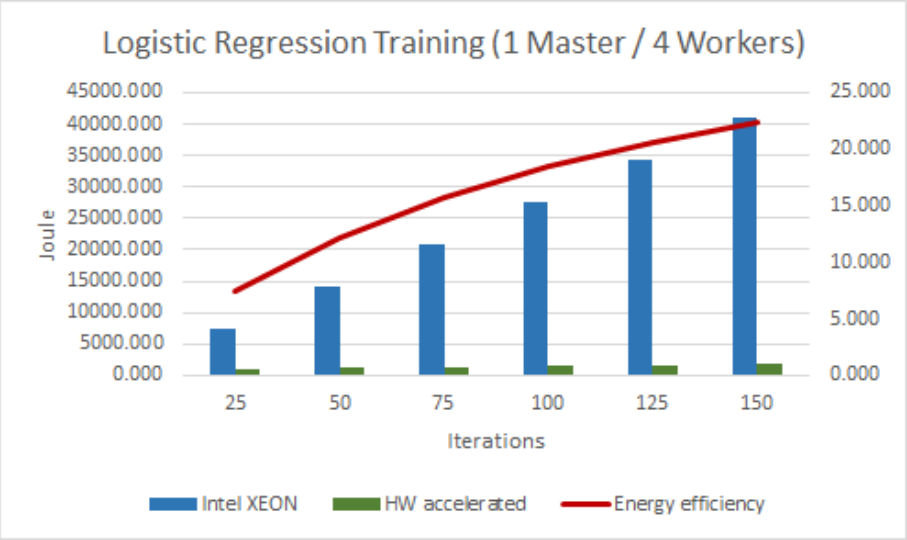


Figure 5.13: Energy consumption of the Xeon and Zynq platforms based on the number of iterations

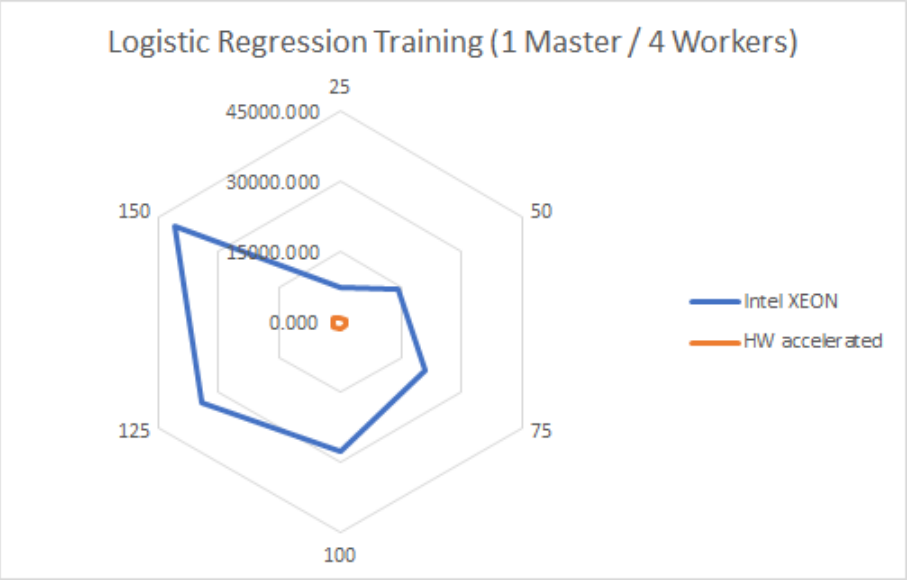


Figure 5.14: Energy consumption footprint of the Xeon and Zynq platforms based on the number of iterations

Zynq platform for both execution cases (CPU-only and HW accelerated). In the first case, the average power consumption of the Xeon processor and the DRAMs is 103 Watt, while a single Zynq node (both the MPSoC FPGA and the DRAM) consumes about 2.6 Watt during the data extraction and 3.2 Watt during the hardware computations accordingly. So in this first case, we can achieve up to 18x better energy efficiency (at 100 iterations) due to the lower power consumption and the lower execution time.

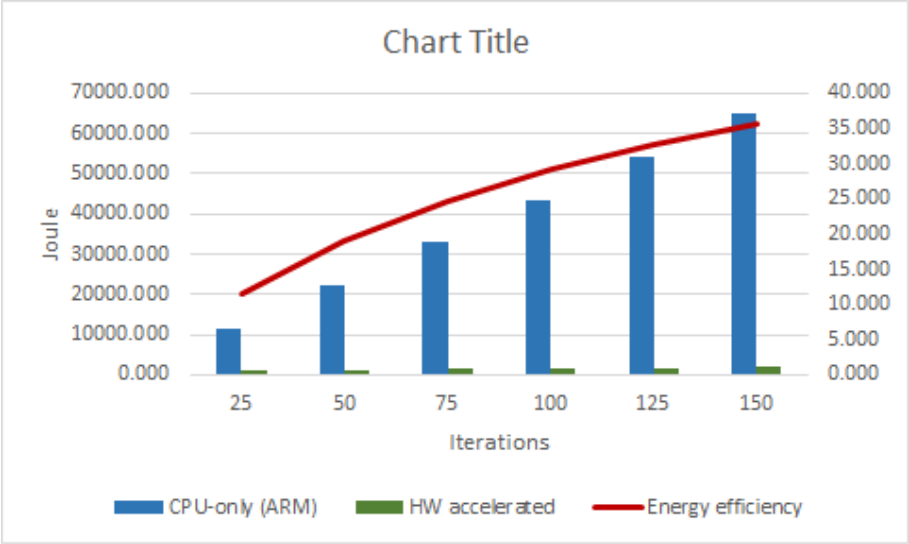


Figure 5.15: Energy consumption of the ARM-only and Zynq platforms based on the number of iterations

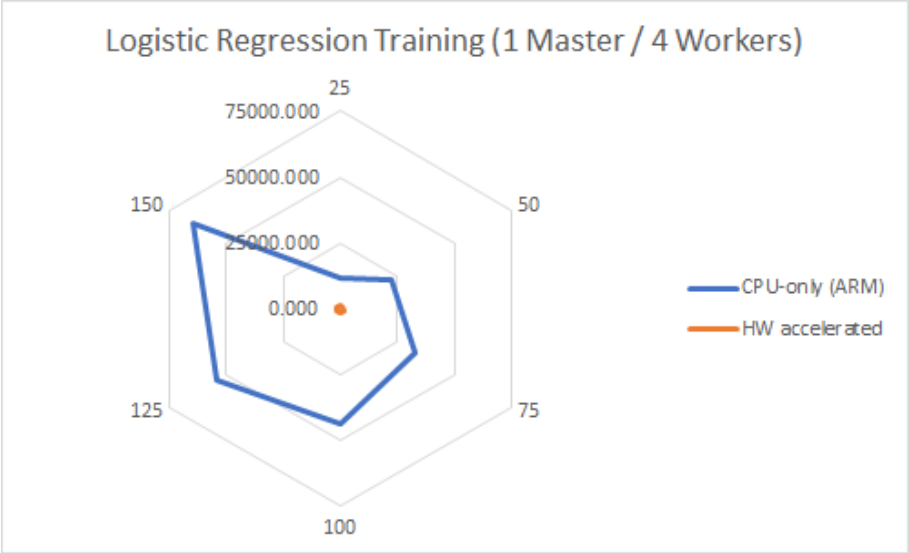


Figure 5.16: Energy consumption footprint of the ARM-only and Zynq platforms based on the number of iterations

In figures 5.15 and 5.16 now, we can see that although the power consumption of the accelerated execution on Zynq is slightly higher than the power consumption of the execution using only the A9 CPU cores (2.3 Watt average, per Worker), we can finally achieve up to 29x lower energy consumption due to the significant higher execution time on the CPU-only solution.

6 Conclusions - Future Work

6.1 Conclusions

In this thesis, we demonstrated all the required steps for building and mapping Spark on low-power embedded platforms like Raspberry Pi 3 and DragonBoard 410, including all the necessary modifications we made for being able to run Spark. Apart from that, we built a cluster consisting of four PYNQ-Z1 nodes, and Spark was deployed on it. Then, we implemented the SPynq framework for the seamless utilization of hardware accelerators in Spark applications. Finally, we performed two evaluations. In the first one, we evaluated the execution time of Spark's six widely used applications from the field of machine learning and graph processing. We then compared these results, with the results of the same execution on a typical mainstream server and a personal computer and in the end, we used the TDP metric to approximate the energy efficiency of all the involved platforms. As far as the second evaluation process is concerned, an a accelerator for a use case scenario on Logistic Regression was used. We implemented its Python API as well as new libraries for Spark that invoke this API. Finally performance and energy efficiency metrics where taken. The results were compared to a typical datacenter server hosting an Intel Xeon processor.

Overall, the mapping of Spark on embedded systems comes with challenges. The are mainly two types of problems: compatibility issues and problems regarding the available hardware resources. Over the time, the compatibility issues are reduced since applications, frameworks and packages are adding support for more and more system architectures. On the other hand, it seems that for being able to run applications on big data analytics frameworks, a major factor is the system's

available memory. The lack of a large RAM could mean that the data doesn't fit in memory. Especially for Spark, where most computations occur in-memory, this means that its major advantage of being so fast is lost.

Also, we tried and built Spark from source on both the Pi 3 and DragonBoard 410c. Of course, this was done in order to test the capabilities of these embedded platforms and for research purposes only. The difference in time that was spent for building Apache Spark on embedded devices compared to the time necessary for building a redistributable package of it on a mainstream server and then have it copied on any embedded devices is very big. If a user does not need to set his own parameters when building Spark, the best option he has, is to download a pre-built version of it. In that way, not only he can save a lot of time but he could also avoid himself running into more trouble.

Finally, from the evaluation we performed, we saw that the SoC-based processors are 8x to 11x worse in terms of performance while due to the lower power consumption, they have the potential to offer much better energy efficiency. For machine learning applications based on Spark they can provide up to 3x better energy efficiency while for graph computations they can provide up to 3.5x better energy efficiency. Therefore, in cases where we care more about the energy efficiency and not mainly for the execution time, the SoC-based servers could provide a promising alternative in order to reduce the power and the total cost of ownership (TCO) of data centers. The results of this evaluation can be also found in the paper entitled "*Performance and Energy evaluation of Spark applications on low-power SoCs*" [47]

Concerning the second evaluation, currently data analytics frameworks like Spark do not support the seamless utilization of hardware accelerators. However, in this thesis the implemented framework showed that the accelerators can improve significantly the performance and the energy efficiency of data analytic applications. It was found that the proposed setup can be used both in high performance systems to reduce the energy consumption (up to 18x) and the execution time by a factor of up to 2x, while in embedded systems it can achieve up to 36x speedup compared to the embedded processors and up to 29x lower energy consumption. It was also shown that the proposed framework can be utilized to support any kind

of hardware accelerators, in order to speedup the execution time of computational intensive machine learning and data analytics applications based on Spark.
The results of this evaluation can be also found in the paper entitled "*SPynq: Acceleration of Machine Learning Applications over Spark on Pynq*" [48].

6.2 Future Work

As far as the future work is concerned, there could be several extensions and improvements.
First of all, although the evaluation of the SPynq framework showed promising results, we need to enrich our *mllib_accel* lbrary to also support other machine learning algorithms like KMeans, Linear Regression etc. Therefore, more Python APIs have to be implemented for new hardware accelerators.
Furthermore, as we saw in chapter 4, although the heterogeneity is genreally supported in Spark, right now we can't take advantage of the heterogeneity of a cluster. In more detail, we saw that one can easily use environment variables (like the *SPARK_WORKER_TYPE* one) to separate for example nodes that host a programmable logic from nodes that don't, but this does not really offer any advantages. Spark's task scheduler distributes the tasks to its workers in parallel and waits for them to be computed in return. Until every worker has returned the piece of work it was assigned, Spark can't continue by sending new tasks. So, upon having a heterogeneous cluster of workers, the acceleration that is provided from the nodes that host an FPGA, is overlapped by the execution time on the nodes that host only CPU cores. In order to take advantage of heterogeneous clusters, a new task scheduler should be implemented which is aware of the nodes' resources and can schedule in a more smart way the distribution of the tasks depending on the cluster heterogeneity.
Finally, to exploit to the maximum the benefits of hardware accelerators, our framework could be modified to be deployed on Amazon's F1 instances or on other platforms hosting FPGAs connected on PCI-express cards. In this way, the benefits of the acceleration could be combined with the powerful processors of today's mainstream servers, therefore combining great performance with better energy efficiency.

Appendices

A′ Scripts and Libraries

In this appendix, you can find the bash script we used for the evaluation of the embedded low-power systems, as well as the whole Python API for the corresponding hardware accelerator and the *mllib_accel* library which is used for acceleration in Apache Spark.

A′.1 Bash script

The bash script that we used for the first part of the evaluation is the following. This script takes one input argument, which determines how many times each application will be executed.

```
#!/bin/bash

LINE="$1"
: > adv_results.txt
echo 'Logistic Regression' >> adv_results.txt
for ((i=1; i <= $1; i++)); do
  /usr/bin/time &>> logistic_tmp.txt -v ../bin/run-example ml.
    LogisticRegressionExample --regParam 0.001 --maxIter 100000000 --
    tol 1.0E-20 --elasticNetParam 0.9 ../data/mllib/sample_libsvm_data.
    txt
done

grep "Elapsed (wall clock) time (h:mm:ss or m:ss):" logistic_tmp.txt >>
     adv_results.txt
rm logistic_tmp.txt

echo 'Analytics - pagerank' >> adv_results.txt
for ((i=1; i <= $1; i++)); do
```

```
15   /usr/bin/time &>> an_pagerank_tmp.txt -v ../bin/run-example graphx.
       Analytics pagerank ../data/mllib/pagerank_data.txt --numEPart=50
16 done
17
18 grep "Elapsed (wall clock) time (h:mm:ss or m:ss):" an_pagerank_tmp.txt
       >> adv_results.txt
19 rm an_pagerank_tmp.txt
20
21 echo 'Analytics - cc' >> adv_results.txt
22 for ((i=1; i <= $1; i++)); do
23   /usr/bin/time &>> an_cc_tmp.txt -v ../bin/run-example graphx.
       Analytics cc ../data/mllib/pagerank_data.txt --numEPart=50
24 done
25
26 grep "Elapsed (wall clock) time (h:mm:ss or m:ss):" an_cc_tmp.txt >>
       adv_results.txt
27 rm an_cc_tmp.txt
28
29 echo 'Analytics - triangles' >> adv_results.txt
30 for ((i=1; i <= $1; i++)); do
31   /usr/bin/time &>> an_triangles_tmp.txt -v ../bin/run-example graphx.
       Analytics triangles ../data/mllib/pagerank_data.txt --numEPart=50
32 done
33
34 grep "Elapsed (wall clock) time (h:mm:ss or m:ss):" an_triangles_tmp.
       txt >> adv_results.txt
35 rm an_triangles_tmp.txt
36
37 echo 'KMeans' >> adv_results.txt
38 for ((i=1; i <= $1; i++)); do
39   /usr/bin/time &>> KMeans_tmp.txt -v ../bin/run-example ml.
       KMeansExample
40 done
41
42 grep "Elapsed (wall clock) time (h:mm:ss or m:ss):" KMeans_tmp.txt >>
       adv_results.txt
43 rm KMeans_tmp.txt
44
45 echo 'Linear Regression' >> adv_results.txt
```

```
for ((i=1; i <= $1; i++)); do
  /usr/bin/time &>> linear_regression_tmp.txt -v ../bin/run-example ml.
    LinearRegressionExample --regParam 0.001 --tol 1.0E-20 --
    elasticNetParam 1.0 ../data/mllib/sample_linear_regression_data.txt
done

grep "Elapsed (wall clock) time (h:mm:ss or m:ss):"
    linear_regression_tmp.txt >> adv_results.txt
rm linear_regression_tmp.txt

awk -v var="$LINE" -F: '{ {if (NR % (var+1) == 1) print; else s+=60*$5+
    $6;}; if (NR % (var+1) == 0) {print s/var; s=0;}}' adv_results.txt
    > adv_results1.txt
mv adv_results1.txt adv_results.txt
```

Listing A′.1: Bash script for evaluating the ML and GraphX applications

A′.2 mllib_accel

Below you can find the Python API for the hardware accelerator of the logistic regression use case we presented.

```
import cffi
import numpy as np
import os
import platform
import re

from itertools import tee
from math import ceil
from pynq import MMIO, Overlay, PL
from .drivers import DMA

chunkSizeMax = 5200
numClassesMax = 10
numFeaturesMax = 784

BS_SEARCH_PATH = os.path.dirname(os.path.realpath(__file__)) + "/
    overlays/"

```

```
ffi = cffi.FFI()

ffi.cdef("""
static uint32_t xlnkBufCnt = 0;
uint32_t cma_mmap(uint32_t phyAddr, uint32_t len);
uint32_t cma_munmap(void *buf, uint32_t len);
void *cma_alloc(uint32_t len, uint32_t cacheable);
uint32_t cma_get_phy_addr(void *buf);
void cma_free(void *buf);
uint32_t cma_pages_available();
""")

LIB_SEARCH_PATH = os.path.dirname(os.path.realpath(__file__)) + "/
    drivers/"
if platform.machine() == "x86_64":
  # load 64bit ELF
  libxlnk = ffi.dlopen(LIB_SEARCH_PATH + "libsds_lib64.so")
elif platform.machine() == "i686":
  # load 32bit ELF
  libxlnk = ffi.dlopen(LIB_SEARCH_PATH + "libsds_lib32.so")
elif platform.machine() == "armv7l":
  # load 32bit ELF compiled for ARM
  libxlnk = ffi.dlopen(LIB_SEARCH_PATH + "libsds_lib.so")
else:
  print("Machine type not supported. Exiting!")
  exit(1)

DMA_TO_DEV = 0     # DMA sends data to PL.
DMA_FROM_DEV = 1   # DMA receives data from PL.

def cma(LabeledPoints):

  # ----------------------------
  #   Download Overlay.
  # ----------------------------

  ol = Overlay(BS_SEARCH_PATH + "LogisticRegression.bit")
  ol.download()

```

```
56  elements = []
57
58  LabeledPoints1, LabeledPoints2 = tee(LabeledPoints, 2)
59
60  numLabeledPoints = sum(1 for _ in LabeledPoints1)
61  numChunks = int(ceil(numLabeledPoints / chunkSizeMax))
62  chunkSize = int(ceil(numLabeledPoints / numChunks))
63  if bool(chunkSize & 1):
64    chunkSize += 1
65  paddingSize = (numChunks * chunkSize) - numLabeledPoints
66
67  c = 1
68
69  # ------------------------
70  #   Allocate physically contiguous memory buffers.
71  #   Cast underlying buffers to a specific C-Type.
72  #   Get the physical address of the buffers.
73  #   Get the virtual address of the buffers.
74  # ------------------------
75
76  data1_buf = libxlnk.cma_alloc(int(chunkSize / 2) * (numClassesMax +
     (1 + numFeaturesMax)) * 4, 1)
77  if data1_buf == ffi.NULL:
78    raise RuntimeError("Memory allocation failed.")
79  data1 = ffi.cast("float *", data1_buf)
80  data2_buf = libxlnk.cma_alloc(int(chunkSize / 2) * (numClassesMax +
     (1 + numFeaturesMax)) * 4, 1)
81  if data2_buf == ffi.NULL:
82    raise RuntimeError("Memory allocation failed.")
83  data2 = ffi.cast("float *", data2_buf)
84
85  buffers = []
86  buffers.append(int(libxlnk.cma_get_phy_addr(data1_buf)))
87  buffers.append(int(re.split("<| |>|0x", str(data1_buf))[5], 16))
88  buffers.append(int(libxlnk.cma_get_phy_addr(data2_buf)))
89  buffers.append(int(re.split("<| |>|0x", str(data2_buf))[5], 16))
90  buffers.append(chunkSize * (numClassesMax + (1 + numFeaturesMax)) *
     4)
```

```
91  buffers.append((chunkSize - paddingSize) if c == numChunks else
     chunkSize)
92  elements.append(buffers)
93
94  i = 0
95  for LabeledPoint in LabeledPoints2:
96    if i < int(chunkSize / 2):
97      data1[i * (numClassesMax + (1 + numFeaturesMax)) + int(
    LabeledPoint.label)] = 1.0
98      data1[i * (numClassesMax + (1 + numFeaturesMax)) + numClassesMax]
     = 1.0
99      f = ffi.from_buffer(LabeledPoint.features.astype(np.float32))
100     features = ffi.cast("float *", f)
101     offset = i * (numClassesMax + (1 + numFeaturesMax)) +
    numClassesMax + 1
102     data1[offset:offset + len(LabeledPoint.features)] = features[0:
    len(LabeledPoint.features)]
103   else:
104     data2[(i - int(chunkSize / 2)) * (numClassesMax + (1 +
    numFeaturesMax)) + int(LabeledPoint.label)] = 1.0
105     data2[(i - int(chunkSize / 2)) * (numClassesMax + (1 +
    numFeaturesMax)) + numClassesMax] = 1.0
106     f = ffi.from_buffer(LabeledPoint.features.astype(np.float32))
107     features = ffi.cast("float *", f)
108     offset = (i - int(chunkSize / 2)) * (numClassesMax + (1 +
    numFeaturesMax)) + numClassesMax + 1
109     data2[offset:offset + len(LabeledPoint.features)] = features[0:
    len(LabeledPoint.features)]
110
111   i += 1
112   if i == chunkSize:
113     c += 1
114     if c <= numChunks:
115
116     # ------------------------
117     #   Allocate physically contiguous memory buffers.
118     #   Cast underlying buffers to a specific C-Type.
119     #   Get the physical address of the buffers.
120     #   Get the virtual address of the buffers.
```

```
121       # ----------------------------
122
123       data1_buf = libxlnk.cma_alloc(int(chunkSize / 2) * (numClassesMax
      + (1 + numFeaturesMax)) * 4, 1)
124       if data1_buf == ffi.NULL:
125         raise RuntimeError("Memory allocation failed.")
126       data1 = ffi.cast("float *", data1_buf)
127       data2_buf = libxlnk.cma_alloc(int(chunkSize / 2) * (numClassesMax
      + (1 + numFeaturesMax)) * 4, 1)
128       if data2_buf == ffi.NULL:
129         raise RuntimeError("Memory allocation failed.")
130       data2 = ffi.cast("float *", data2_buf)
131
132       buffers = []
133       buffers.append(int(libxlnk.cma_get_phy_addr(data1_buf)))
134       buffers.append(int(re.split("<| |>|0x", str(data1_buf))[5], 16))
135       buffers.append(int(libxlnk.cma_get_phy_addr(data2_buf)))
136       buffers.append(int(re.split("<| |>|0x", str(data2_buf))[5], 16))
137       buffers.append(chunkSize * (numClassesMax + (1 + numFeaturesMax))
      * 4)
138       buffers.append((chunkSize - paddingSize) if c == numChunks else
     chunkSize)
139       elements.append(buffers)
140
141       i = 0
142
143   return elements
144
145 def gradients_kernel_accel(buffers, weights):
146   chunkSize = int(buffers[4] / (numClassesMax + (1 + numFeaturesMax)) /
        4)
147   numClasses = len(weights)
148   numFeatures = len(weights[0]) - 1
149
150   # ----------------------------
151   #   Physical address of the Accelerator Adapter IP.
152   # ----------------------------
153
```

```
154 ADDR_Accelerator_Adapter_BASE = int(PL.ip_dict["
      SEG_LR_gradients_kernel_accel_0_if_Reg"][0], 16)
155 ADDR_Accelerator_Adapter_RANGE = int(PL.ip_dict["
      SEG_LR_gradients_kernel_accel_0_if_Reg"][1], 16)
156
157 # ----------------------------
158 #    Initialize new MMIO object.
159 # ----------------------------
160
161 bus = MMIO(ADDR_Accelerator_Adapter_BASE,
      ADDR_Accelerator_Adapter_RANGE)
162
163 # ----------------------------
164 #   Physical addresses of the DMA IPs.
165 # ----------------------------
166
167 ADDR_DMA0_BASE = int(PL.ip_dict["SEG_dm_0_Reg"][0], 16)
168 ADDR_DMA1_BASE = int(PL.ip_dict["SEG_dm_1_Reg"][0], 16)
169 ADDR_DMA2_BASE = int(PL.ip_dict["SEG_dm_2_Reg"][0], 16)
170 ADDR_DMA3_BASE = int(PL.ip_dict["SEG_dm_3_Reg"][0], 16)
171
172 # ----------------------------
173 #    Initialize new DMA objects.
174 # ----------------------------
175
176 dma0 = DMA(ADDR_DMA0_BASE, direction = DMA_TO_DEV)     # data1 DMA
177 dma1 = DMA(ADDR_DMA1_BASE, direction = DMA_TO_DEV)     # data2 DMA
178 dma2 = DMA(ADDR_DMA2_BASE, direction = DMA_TO_DEV)     # weights DMA
179 dma3 = DMA(ADDR_DMA3_BASE, direction = DMA_FROM_DEV)  # gradients DMA
180
181 # ----------------------------
182 #    Assign/Allocate physically contiguous memory buffers.
183 # ----------------------------
184
185 dma0._bufPtr = ffi.cast("uint32_t *", buffers[0])
186 dma0.buf = ffi.cast("void *", buffers[1])
187 dma0.bufLength = int(buffers[4] / 2)
188 dma1._bufPtr = ffi.cast("uint32_t *", buffers[2])
189 dma1.buf = ffi.cast("void *", buffers[3])
```

```
190 dma1.bufLength = int(buffers[4] / 2)
191 dma2.create_buf((numClassesMax * (1 + numFeaturesMax)) * 4)
192 dma3.create_buf((numClassesMax * (1 + numFeaturesMax)) * 4)
193
194 # ----------------------------
195 #    Get CFFI pointers to objects' internal buffers.
196 # ----------------------------
197
198 weights_buf = dma2.get_buf(data_type = "float")
199 gradients_buf = dma3.get_buf(data_type = "float")
200
201 w = ffi.from_buffer(np.pad(weights, ((0, 0), (0, numFeaturesMax -
      numFeatures)), "constant").flatten().astype(np.float32))
202 w_buf = ffi.cast("float *", w)
203 weights_buf[0:numClasses * (1 + numFeaturesMax)] = w_buf[0:numClasses
      * (1 + numFeaturesMax)]
204
205 # ----------------------------
206 #   Write data to MMIO.
207 # ----------------------------
208
209 CMD = 0x0028              # Command.
210 ISCALAR0_DATA = 0x0080  # Input Scalar-0 Write Data FIFO.
211
212 bus.write(ISCALAR0_DATA, chunkSize)
213 bus.write(CMD, 0x00010001)
214 bus.write(CMD, 0x00020000)
215 bus.write(CMD, 0x00000107)
216
217 # ----------------------------
218 #   Transfer data using DMAs (Non-blocking).
219 #   Block while DMAs are busy.
220 # ----------------------------
221
222 dma0.transfer(int(buffers[4] / 2), direction = DMA_TO_DEV)
223 dma1.transfer(int(buffers[4] / 2), direction = DMA_TO_DEV)
224 dma2.transfer((numClassesMax * (1 + numFeaturesMax)) * 4, direction =
      DMA_TO_DEV)
225
```

```
226   dma0.wait()
227   dma1.wait()
228   dma2.wait()
229
230   dma3.transfer((numClassesMax * (1 + numFeaturesMax)) * 4, direction =
        DMA_FROM_DEV)
231
232   dma3.wait()
233
234   g_buf = ffi.buffer(gradients_buf, (numClasses * (1 + numFeaturesMax))
        * 4)
235   g = np.frombuffer(g_buf, dtype = np.float32)
236   gradients = np.copy(np.reshape(g, (numClasses, 1 + numFeaturesMax))
        [:, :1 + numFeatures])
237
238   # ----------------------------
239   #   Destructors for DMA objects.
240   # ----------------------------
241
242   dma0.buf = None
243   dma0._bufPtr = None
244   dma0.bufLength = None
245   dma1.buf = None
246   dma1._bufPtr = None
247   dma1.bufLength = None
248
249   dma0.__del__()
250   dma1.__del__()
251   dma2.__del__()
252   dma3.__del__()
253
254   return gradients
255
256 def cmf(buffers):
257
258   # ----------------------------
259   #   Free previously allocated buffers.
260   # ----------------------------
261
```

```
262   data1_buf = ffi.cast("void *", buffers[1])
263   data2_buf = ffi.cast("void *", buffers[3])
264   libxlnk.cma_free(data1_buf)
265   libxlnk.cma_free(data2_buf)
266
267   return 0
```

Listing A′.2: Python API (LogisticRegression.py) of the logistic regression hardware accelerator

Below you can find the new *classification.py* that was implemented.

```
1  import numpy as np
2
3  from math import exp
4  from pyspark import RDD
5  from time import time
6  from .accelerators.LogisticRegression import cma,
       gradients_kernel_accel, cmf
7
8  __all__ = ['LogisticRegression']
9
10 class LogisticRegression(object):
11   """
12   Multiclass Logistic Regression Model.
13
14     :param numClasses:      Number of possible outcomes.
15
16     :param numFeatures:     Dimension of the features.
17
18     :param weights:         Weights computed for every feature.
19     (The intercepts will be part of the weights.)
20   """
21
22   def __init__(self, numClasses, numFeatures, weights = None):
23     self.numClasses = numClasses
24     self.numFeatures = numFeatures
25     self.weights = weights
26
27   def train(self, trainRDD, alpha = 1.0, numIterations = 100, _accel_ =
       0):
```

```
28    """
29    Train a logistic regression model on the given data.
30
31      :param trainRDD:          The training data, an RDD of
    LabeledPoint.
32
33      :param alpha:             The learning rate (default: 1.0).
34
35      :param numIterations:     The number of iterations (default: 100).
36
37      :param _accel_:           0: SW-only, 1: HW accelerated (deafult:
    0).
38
39      :note:                    Labels used in logistic regression
    should be
40                  {0, 1, ..., k - 1} for k classes classification
    problem.
41    """
42
43    def gradients_kernel(data):
44      # Compute the gradients given the (label, features) pair of a
    single data point.
45      gradients = np.zeros((self.numClasses, 1 + self.numFeatures))
46
47      for k in range (0, self.numClasses):
48        # margin (rawPrediction)
49        margin = self.weights[k][0] + np.dot(data.features, self.
    weights[k][1:])
50
51        # score (probability)
52        if margin < 0:
53          score = 1.0 - 1.0 / (1.0 + exp(margin))
54        else:
55          score = 1.0 / (1.0 + exp(-margin))
56
57        multiplier = score - (1.0 if k == int(data.label) else 0.0)
58
59        gradients[k][0] = multiplier
60        gradients[k][1:] = multiplier * data.features
```

```
61
62          return gradients
63
64      # Reduction of multiclass classification to binary classification.
65      # Performs reduction using one against all strategy (OneVsRest).
66      # For a multiclass classification with k classes, train k models (
    one per class).
67      print("     * LogisticRegression Training *")
68
69      if _accel_:
70          trainRDD = trainRDD.mapPartitions(cma).cache()
71          numExamples = trainRDD.map(lambda buffers: buffers[5]).sum()
72      else:
73          trainRDD = trainRDD.cache()
74          numExamples = trainRDD.count()
75
76      print("      # numExamples:                {:d}".format(numExamples))
77      print("      # numClasses:                 {:d}".format(self.
    numClasses))
78      print("      # numFeatures:                {:d}".format(self.
    numFeatures))
79      print("      # alpha:                      {:g}".format(alpha))
80      print("      # numIterations:              {:d}".format(numIterations)
    )
81
82      start = time()
83
84      # Run Batch Gradient Descent (BGD) in parallel.
85      # Averaging the subgradients over different partitions is
86      # performed using one standard spark map-reduce in each iteration.
87      if self.weights is None:
88          self.weights = np.zeros((self.numClasses, 1 + self.numFeatures))
89
90      # Momentum Optimization.
91      gamma = 0.9
92      velocity = np.zeros_like(self.weights)
93
94      for t in range(0, numIterations):
95          print("{:3d}% |{:35s}| {:d}/{:d}".format(int((t / numIterations)
```

```
    * 100), u"\u25A5" * int((t / numIterations) * 35), t, numIterations
    ), end = "\r")
96
97      if _accel_:
98        gradients = trainRDD.map(lambda buffers: gradients_kernel_accel
    (buffers, self.weights)).reduce(lambda a, b: np.add(a, b))
99      else:
100       gradients = trainRDD.map(gradients_kernel).reduce(lambda a, b:
    np.add(a, b))
101
102     velocity = np.add(gamma * velocity, (alpha / numExamples) *
    gradients)
103     self.weights = np.subtract(self.weights, velocity)
104
105   end = time()
106   print("{:3d}% |{:35s}| {:d}/{:d} Time: {:.3f} sec".format(100, u"\
    u25A5" * 35, numIterations, numIterations, end - start))
107
108   if _accel_:
109     trainRDD.map(cmf).collect()
110
111   return self
112
113 def test(self, testRDD):
114   """
115   Test a logistic regression model on the given data.
116
117     :param testRDD:    The testing data, an RDD of LabeledPoint.
118
119     :note:             Labels used in logistic regression should be
120                {0, 1, ..., k - 1} for k classes classification
    problem.
121   """
122
123   # Each example is scored against all k models and the model with
    highest score
124   # is picked to label the example.
125   print("    * LogisticRegression Testing *")
126
```

```
127     true = testRDD.map(lambda data: 1 if data.label == self.predict(
        data.features) else 0).reduce(lambda a, b: a + b)
128     false = testRDD.count() - true
129
130     print("     # accuracy:                {:.3f} ({:d}/{:d})".format(
        true / (true + false), true, true + false))
131     print("     # true:                    {:d}".format(true))
132     print("     # false:                   {:d}".format(false))
133
134   def predict(self, features):
135     """
136     Predict values for a single data point using the model trained.
137
138       :param features:    Features to be labeled.
139     """
140
141     # Compute and find the one with maximum margins.
142     maxMargin = -np.inf
143     bestClass = -1
144
145     for k in range(0, self.numClasses):
146       # margin (rawPrediction)
147       margin = self.weights[k][0] + np.dot(features, self.weights[k
        ][1:])
148
149       if margin > maxMargin:
150         maxMargin = margin
151         bestClass = k
152
153     return bestClass
154
155   def save(self, path):
156     """
157     Save this model to the given path.
158     """
159
160     np.savetxt(path, self.weights)
161
162   def load(self, path):
```

```
163     """
164     Load a model from the given path.
165     """
166
167     self.weights = np.loadtxt(path)
```

Listing A′.3: New Spark library (classification.py) that invokes the hardware accelerator Python API

The example application we used for running the accelerated logistic regression version is the following:

```
1  from pyspark import SparkContext
2  from pyspark.mllib.regression import LabeledPoint
3  from pyspark.mllib_accel.classification import LogisticRegression
4  from sys import argv
5  from time import time
6
7    def parsePoint(line):
8      """
9      Parse a line of text into an MLlib LabeledPoint object.
10     """
11
12     data = [float(s) for s in line.split(',')]
13
14     return LabeledPoint(data[0], data[1:])
15
16 if __name__ == "__main__":
17
18   if len(argv) != 7:
19     print("Usage: LogisticRegressionApp <dataset> <fraction> <
      numPartitions> <alpha> <numIterations> <_accel_>")
20     exit(-1)
21
22   dataset = argv[1]
23   fraction = float(argv[2])
24   numPartitions = int(argv[3])
25   alpha = float(argv[4])
26   numIterations = int(argv[5])
27   _accel_ = int(argv[6])
```

```
28
29  train_file = "inputs/" + dataset + "_train.dat"
30  test_file = "inputs/" + dataset + "_test.dat"
31
32  with open(dataset, 'r') as f:
33    for line in f:
34      if line[0] != '#':
35        parameters = line.split(',')
36        numClasses = int(parameters[0])
37        numFeatures = int(parameters[1])
38  f.close()
39
40  sc = SparkContext(appName = "Python Logistic Regression on " +
     dataset)
41
42  print("* LogisticRegression Application *")
43  print(" # train file:                {:s}".format(train_file))
44  print(" # fraction:                  {:g}".format(fraction))
45  print(" # test file:                 {:s}".format(test_file))
46  print(" # numPartitions:             {:d}".format(numPartitions))
47
48  trainRDD = sc.textFile(train_file, numPartitions).sample(False,
     fraction).map(parsePoint)
49  testRDD = sc.textFile(test_file, numPartitions).map(parsePoint)
50
51  start = time()
52
53  # Train a logistic regression model given an RDD of (label, features)
       pairs.
54  # We run a fixed number of iterations of Gradient Descent using the
     specified alpha.
55  # We use the entire data set to update the gradient in each iteration
       (Batch GD).
56  LR = LogisticRegression(numClasses, numFeatures).train(trainRDD,
     alpha, numIterations, _accel_)
57
58  end = time()
59  if _accel_:
60    print("! Time running LogisticRegression train in hardware: {:.3f}
```

```
    sec".format(end - start))
61  else:
62    print("! Time running LogisticRegression train in software: {:.3f}
      sec".format(end - start))
63
64  LR.save("outputs/weights.out")
65
66  LR.test(testRDD)
67
68  sc.stop()
```

Listing A′.4: Logistic Regression example application

Finally, below you can find the *lr.sh* bash script we used for running the example application:

```
1 #!/bin/bash
2
3 # $1 = _accel_ and $2 = dataset, or just $1 = dataset
4 if [ "$1" == "_accel_" ]
5 then
6   time spark-submit LogisticRegressionApp.py $2 1.0 4 0.75 100 1
7 else
8   time spark-submit LogisticRegressionApp.py $1 1.0 4 0.75 100 0
9 fi
```

Listing A′.5: Bash script for executing the Logistic Regression example

Bibliography

[1] **FPGA acceleration of Spark applications in a Pynq cluster**, Christofors Kachris, Elias Koromilas, Ioannis Stamelos, Dimitrios Soudris
FPL 2017

[2] **Big Data**,
https://en.wikipedia.org/wiki/Big_data, http://semeon.com/blog/big-data-ad-agencies-giving-reason-to-gut-decisions/

[3] **The Network Impact of Big Data**,
http://www.nojitter.com/post/240170228/the-network-impact-of-big-data

[4] **Cluster computing**, Lubna Luxmi Chowdhry, 20 Sep 2005
https://www.codeproject.com/Articles/11709/Cluster-Computing

[5] **Cluster**,
https://techterms.com/definition/cluster

[6] **Amdahl's Model for Scaling Efficiency**,
https://en.wikipedia.org/wiki/Amdahl%27s_law

[7] **Embedded Systems**,
https://en.wikipedia.org/wiki/Embedded_system,
http://internetofthingsagenda.techtarget.com/definition/embeddedsystem

[8] **Europe embedded system market size**,
https://www.gminsights.com/industry-analysis/embedded-system-market

[9] **AMD pins future growth to embedded marketplace**,
https://www.theregister.co.uk/2013/04/23/amd_gseries_embedded_chips

[10] **FPGAs**,
https://en.wikipedia.org/wiki/Field-programmable_gate_array

[11] **FPGAs Advantages**,
http://community.brocade.com/t5/Service-Providers/FPGA-or-ASIC-Pro-s-amp-Con-s-of-Each-Technology/ba-p/709

[12] **FPGAs in the Datacenter**,
https://www.forbes.com/sites/moorinsights/2016/11/14/xilinx-seeks-to-mainstream-fpgas-in-the-datacenter/

[13] **Xilinx All Programmable Devices**,
https://www.xilinx.com/support/documentation/white_papers/wp492-compute-intensive-sys.pdf

[14] **Amazon F1 Instances**,
https://aws.amazon.com/ec2/instance-types/f1/

[15] **Heterogeneous Computer Architectures**,
https://www.iti.uni-stuttgart.de/abteilungen/rechnerarchitektur/projekte/heterogeneous-computing.html

[16] **Heterogeneous Computing**,
https://en.wikipedia.org/wiki/Heterogeneous_ computing

[17] **Apache Spark - Databricks**,
https://databricks.com/spark/about

[18] **Apache Spark**,
https://spark.apache.org/

[19] **Hadoop**,
http://hadoop.apache.org/

[20] **Cassandra**,
http://cassandra.apache.org/

[21] **Safari Books - Apache Spark**,
https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/ch01.html

[22] **Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing**,Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
https://cs.stanford.edu/ matei/papers/2012/nsdi_ spark.pdf

[23] **Jacek Laskowski - Mastering Apache Spark 2**,
https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details

[24] **Architecture of Spark SQL**,
https://www.packtpub.com/mapt/book/big_ data_ and_ business_ intelligence/9781785884696/4/c of-spark-sql

[25] **Spark Cluster Managers**,
http://www.agildata.com/apache-spark-cluster-managers-yarn-mesos-or-standalone/

[26] **Cisco Visual Networking Index**, Global Mobile Data Traffic Forecast Update 20142019 White Paper

[27] **Power Challenges May End the Multicore Era**, Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger
http://www.cc.gatech.edu/ hadi/doc/paper/2013-cacm-dark_ silicon.pdf

[28] **Raspberry Pi - Wikipedia**,
https://en.wikipedia.org/wiki/Raspberry_Pi

[29] **MagPi - Pi 3 Specs, Benchmarks and More**,
https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/

[30] **DragonBoard 410c - Qualcomm**,
https://developer.qualcomm.com/hardware/dragonboard-410c

[31] **Pynq: PYTHON PRODUCTIVITY FOR ZYNQ**,
http://www.pynq.io/

[32] **Linear Regression - Wikipedia**,
https://en.wikipedia.org/wiki/Linear_regression

[33] **Logistic Regression - Wikipedia**,
https://en.wikipedia.org/wiki/Logistic_regression

[34] **K-means Clustering - Wikipedia**,
https://en.wikipedia.org/wiki/K-means_clustering

[35] **PageRank - Wikipedia**,
https://en.wikipedia.org/wiki/PageRank

[36] **Intel Acquires Altera**,
https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/

[37] **Zynq-7000 All Programmable SoC**,
https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[38] **Python in Big Data**,
https://www.continuum.io/why-python

[39] **PYNQ - GitHub**,
https://github.com/Xilinx/PYNQ

[40] **PYNQ - Quickstart Guide**,
http://pynq.readthedocs.io/en/latest/1_getting_started.html

[41] **A survey on reconfigurable accelerators for cloud computing**, C. Kachris and D. Soudris
http://ieeexplore.ieee.org/document/7577381/

[42] **Xilinx reconfigurable Acceleration Stack targets machine learning, data analytics and Video Streaming**,
Technical report, 2016.

[43] **FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack**, S. Byma, J.G. Steffan, H. Bannazadeh, A. Leon-Garcia and P. Chow
http://ieeexplore.ieee.org/document/6861604/

[44] **Invited: Heterogeneous datacenters: Options and opportunities**, Jason Cong, Muhuan Huang, Di Wu and Cody Hao Yu
http://ieeexplore.ieee.org/document/7544260/

[45] **Serialization - Wikipedia**,
https://en.wikipedia.org/wiki/Serialization

[46] **FPGA-Acceleration of Machine Learning in Cloud Computing, a case study using Logistic Regression**,Elias Koromilas
Master's Thesis, National Technical University of Athens, 2017

[47] **Performance and Energy evaluation of Spark applications on low-power SoCs**, Christofors Kachris, Ioannis Stamelos, Dimitrios Soudris
http://ieeexplore.ieee.org/document/7818362/

[48] **SPynq: Acceleration of Machine Learning Applications over Spark on Pynq**, Christofors Kachris, Elias Koromilas, Ioannis Stamelos, Dimitrios Soudris
SAMOS 2017