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

Dataflow Acceleration with Maxeler Technologies for Machine Learning Applications

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΔΟΥΚΑΣ ΜΙΧΑΗΛ-ΧΡΗΣΤΟΣ

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2016

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Dataflow Acceleration with Maxeler Technologies for Machine Learning Applications

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΔΟΥΚΑΣ ΜΙΧΑΗΛ-ΧΡΗΣΤΟΣ

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 30η Ιουνίου 2016.

....	....	....
Δημήτριος Ι. Σούντρης Αν. Καθηγητής Ε.Μ.Π.	Κιαμάλ Ζ. Πεκμεστζή Καθηγητής Ε.Μ.Π.	Γεώργιος Οικονομάκος Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2016

....

Δούκας Μιχαήλ-Χρήστος

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

Copyright © Δούκας Μιχαήλ-Χρήστος, 2016.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

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

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

Περίληψη

Στις μέρες μας οι επιχειρήσεις και οι οργανώσεις συλλέγουν δεδομένα από πλήθος πηγών, όπως επιχειρησιακές δοσοληψίες, μέσα κοινωνικής δικτύωσης και αισθητήρες. Ο όρος Big data περιγράφει αυτόν τον μεγάλο όγκο δεδομένων, τα οποία μπορούν να αναλυθούν ώστε να παρθούν έξυπνες αποφάσεις βασιζόμενες σε μοτίβα των δεδομένων. Η ανάλυση των Big data βοηθά τις επιχειρήσεις και τους οργανισμούς να εμεταλλευτούν τα δεδομένα τους και να τα χρησιμοποιήσουν ώστε να εντοπίσουν αποδοτικότερες επιχειρησιακές κινήνεις, αποτελεσματικότερες δράσεις, καλύτερες υπηρεσίες και να αποκομίσουν μεγαλύτερα κέρδη.

Ο μεγάλος αυτός όγκος δεδομένων που παράγεται συνεχώς έχει πυροδοτήσει την ευρεία χρήση της μηχανικής μάθησης (machine learning), η οποία είναι μια μέθοδος ανάλυσης δεδομένων. Οι διεργασίες μηχανικής μάθησης είναι πολύ απαιτητικές από άποψη υπολογιστικής ισχύος. Η αποθήκευση και η επεξεργασία μεγάλου όγκου δεδομένων σε ένα λογικό χρονικό διάστημα είναι πολύ σημαντικές, και το γεγονός αυτό πιέζει τους προγραμματιστές και τους αρχιτέκτονες υπολογιστών να αναπτύξουν αποδοτικότερα υπολογιστικά συστήματα που να ανταποκρίνονται στις απαιτήσεις υψηλής απόδοσης. Πολλές λύσεις έχουν προταθεί, συμπεριλαμαμβανομένων και τών επιταχυντών GPU, παρ'ολα αυτά ο παράγοντας της χαμηλής κατανάλωσης ισχύος δεν έχει ληφθεί υπόψιν.

Μια πιθανή λύση θα μπορούσε να είναι το υπολογιστικό μοντέλο ροής δεδομένων (dataflow computing), ένας εντελώς διαφορετικός τρόπος επεξεργασίας δεδομένων. Η εταιρία Maxeler Technologies χρησιμοποιεί ολοκληρωμένα κυκλώματα FPGAs για να υλοποίησει το υπολογιστικό μοντέλο ροής δεδομένων και παρέχει τάξεις μεγέθους βελτίωση στην απόδοση και την κατανάλωση ισχύος. Σκοπός της εργασίας αυτής είναι η χρήση του υπολογιστικού μοντέλου ροής δεδομένων της Maxeler Technologies, για την επιτάχυνση επιλεγμένων προγραμμάτων μηχανικής μάθησης της Python βιβλιοθήκης scikit-learn, με στόχο την επίτευξη υψηλότερης απόδοσης και χαμηλότερης κατανάλωσης ισχύος.

Αρχικά, εκτελούμε μια ανάλυση του χρόνου εκτέλεσης διάφορων αλγορίθμων μηχανικής μάθησης, ώστε να εντοπίσουμε τους υπολογιστικά απαιτητικότερους. Στη συνέχεια, παρουσιάζουμε το μαθηματικό μοντέλο και τους αλγορίθμους των Gaussian process regression, Gaussian process classification και Kernel ridge regression, τα οποία προγμάμματα επιλέχθηκαν να επιταχυνθούν. Ύστερα, αναλύουμε τα προγμάμματα αυτά και εντοπίζουμε ποιές συναρτήσεις των Python βιβλιοθηκών numpy και scipy μπορούν να υλοποιηθούν στο υπολογιστικό μοντέλο ροής δεδομένων της Maxeler. Τέλος περιγράφουμε τις υλοποιήσεις στο μοντέλο ροής δεδομένων και παρουσιάζουμε τα αποτελέσματα του χρόνου εκτέσης των προγραμμάτων.

Λέξεις κλειδιά

μηχανική μάθηση, υπολογιστικό μοντέλο ροής δεδομένων.

Abstract

Nowadays, organizations collect data from a variety of sources, including business transactions, social media and sensors. Big data is a term that describes this large volume of data, which can be analysed to make intelligent decisions based on patterns. Big data analytics helps organizations exploit their data and use it to identify smarter business moves, more efficient operations, better services and achieve higher profits.

The huge amount of data being constantly generated has sparked the widespread use of machine learning, which is a data analysis method. Machine learning tasks are highly demanding in terms of computation power. Storing and analysing massive data in a reasonable amount of time has become very important, and this fact stresses software developers and computer architects to deliver more efficient design solutions able to address the increased performance requirements. Many solutions have been proposed including GPU accelerators, which deliver high speedup, however the power efficiency factor has been neglected.

A possible solution is dataflow computing, a revolutionary way of performing computation, completely different to computing with conventional CPUs. Maxeler Technologies uses FPGA devices to exploit dataflow computing and provide order of magnitude benefits in performance, space and power consumption. The purpose of this thesis is the utilization of Maxeler's dataflow model for the acceleration of selected machine learning programs from scikit-learn Python library in order to achieve high performance and low power consumption.

First, we perform an execution time analysis on various machine learning algorithms, to determine the most computationally demanding applications of scikit-learn. Next, we present the mathematical model and algorithms of Gaussian process regression and classification (GPR, GPC) and Kernel ridge regression (KRR), which were selected for acceleration. Subsequently, we demonstrate the profiling results of these machine learning techniques and point out which NumPy and SciPy functions should be implemented in a dataflow engine. Finally, we present and describe the dataflow accelerated applications, and demonstrate the speedup results.

Key words

machine learning, dataflow computing, Maxeler Technologies, DFE acceleration, scikit-learn acceleration, Gaussian process acceleration, Kernel ridge regression acceleration.

Ευχαριστίες

Με την εργασία αυτή κλείνει μια από τα σημαντικότερες περιόδους της ζωής μου. Η περίοδος που είχα την τιμή να είμαι φοιτητής στη Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του ΕΜΠ. Κατά την διάρκεια των σπουδών μου είχα την τύχη και την χαρά να γνωρίσω πολλούς αξιόλογους ανθρώπους, που με βοήθησαν να εξελιχθώ ως άτομο αλλά και ως μηχανικός.

Αρχικά θα ήθελα να ευχαριστήσω τον Αν. Καθ. Δημήτριο Σούντρη, που μου έδωσε την ευκαιρία να αναλάβω ένα τόσο ενδιαφέρον θέμα. Επίσης ευχαριστώ τον Δρ. Σωτήρη Ξύδη, για τη συνεργασία και τη συνεχή καθοδήγησή του καθ'όλη τη διάρκεια της διπλωματικής μου εργασίας. Ακόμα θα ήθελα να ευχαριστήσω όλους τους καθηγητές, οι οποίοι με εφοδίασαν με τις πολύτιμες γνώσεις και την εμπειρία τους.

Στη συνέχεια θέλω να ευχαριστήσω την οικογένεια μου που με στήριξε σε κάθε βήμα και μου επέτρεψε να αφοσιωθώ στις σπουδές μου. Τέλος, δεν θα μπορούσα να μην ευχαριστήσω όλους τους συμφοιτητές μου και ιδιαίτερα τον Νίκο, τον Αλέξανδρο, τον Μιχάλη και τον Ιάκωβο, που με βοήθησαν να ξεπεράσω όσες δυσκολίες και με τιμούν με τη φιλία τους.

Δούκας Μιχαήλ-Χρήστος,

Αθήνα, 30η Ιουνίου 2016

Περιεχόμενα

Περίληψη . . . 5

Abstract . . . 7

Ευχαριστίες . . . 9

Περιεχόμενα . . . 11

1. Εισαγωγή . . . 13
- 1.1 Υπολογιστικό μοντέλο ροής δεδομένων . . . 13
- 1.2 Επιτάχυνση αλγορίθμων μηχανικής μάθησης με χρήση του μοντέλου ροής δεδομένων . . . 14

2. Βασικά στοιχεία της μηχανικής μάθησης . . . 15
- 2.1 Μηχανική μάθηση . . . 15
  - 2.1.1 Τα είδη μηχανικής μάθησης . . . 15
- 2.2 Επιτηρούμενη μάθηση . . . 16
  - 2.2.1 Ταξινόμηση . . . 16
  - 2.2.2 Παλινδρόμηση . . . 16

3. Αλγόριθμοι μηχανικής μάθησης της βιβλιοθήκης scikit-learn . . . 17
- 3.1 Ανάλυση χρόνων εκτέλεσης . . . 18
  - 3.1.1 Stochastic Gradient Descent (SGD) classification . . . 18
  - 3.1.2 Quadratic Discriminant Analysis classification . . . 19
  - 3.1.3 Gaussian processes classification . . . 20
  - 3.1.4 Naive Bayes classification . . . 21
  - 3.1.5 Support Vector Machines classification . . . 22
  - 3.1.6 Kernel ridge regression . . . 23
  - 3.1.7 Gaussian processes regression . . . 24
- 3.2 Αλγόριθμοι της scikit-learn για επιτάχυνση με DFEs . . . 25

4. Gaussian Process Regression . . . 27
- 4.1 Αλγόριθμος Gaussian process regression . . . 27
- 4.2 scikit-learn υλοποίηση του Gaussian process regression . . . 27
- 4.3 Ανάλυση scikit-learn Gaussian process regression . . . 30

5. Gaussian Process Classification . . . 33
- 5.1 Αλγόριθμος δυαδικού Gaussian process classification . . . 33
- 5.2 scikit-learn υλοποίηση του Gaussian process classification . . . 34
- 5.3 Ανάλυση scikit-learn Gaussian process classification . . . 38

6. Kernel Ridge Regression . . . 41
6.1 Αλγόριθμος Kernel ridge regression . . . 41
6.2 scikit-learn υλοποίηση του kernel ridge regression . . . 41
6.3 Ανάλυση scikit-learn kernel ridge regression . . . 43

7. Υπολογιστικό μοντέλο ροής δεδομένων της Maxeler . . . 45
7.1 Το υπολογιστικό μοντέλο ροής δεδομένων . . . 45
7.2 Μηχανήματα ροής δεδομένων (DFEs) . . . 46

8. Επιτάχυνση του Πολλαπλασιασμού πινάκων και της Απόστασης πινάκων με το μοντέλο ροής δεδομένων . . . 49
8.1 Γινόμενο πινάκων . . . 49
8.2 Ευκλείδια τετραγωνική απόσταση πινάκων . . . 49
8.3 Οι συναρτήσεις των scipy και numpy που θα επιταχυνθούν . . . 50
8.4 Υλοποίηση στο μοντέλο ροής δεδομένων της Maxeler . . . 50
8.4.1 Πολλαπλασιασμός πινάκων σε tiles . . . 51
8.5 Διάγραμμα αρχιτεκτονικής . . . 52

9. Επιτάχυνση της Cholesky παραγοντοποίησης με το μοντέλο ροής δεδομένων 53
9.1 Οι αλγόριθμοι Cholesky-Banachiewicz και Cholesky-Crout . . . 53
9.2 Οι συναρτήσεις των scipy και numpy που θα επιταχυνθούν . . . 55
9.3 Υλοποίηση στο μοντέλο ροής δεδομένων της Maxeler . . . 56
9.3.1 Kernel . . . 59
9.3.2 Διάγραμμα αρχιτεκτονικής . . . 62

10. Επιτάχυνση του Cholesky solve και Lower triangular solve με το μοντέλο ροής δεδομένων . . . 63
10.1 Lower triangular solve . . . 63
10.2 Cholesky solve . . . 64
10.3 Οι συναρτήσεις των scipy και numpy που θα επιταχυνθούν . . . 65
10.4 Υλοποίηση στο μοντέλο ροής δεδομένων της Maxeler . . . 65
10.4.1 Μπροστά αντικατάσταση με tiles . . . 65
10.4.2 Cholesky Solve με tiles . . . 68
10.4.3 Kernel . . . 69

11. Σύγκριση των υλοποιήσεων σε ροή δεδομένων με τις numpy και scipy συναρτήσεις 71
11.1 Χρόνος εκτέλεσης και επιτάχυνση . . . 71
11.1.1 Πολλαπλασιασμός πινάκων . . . 71
11.1.2 Απόσταση πινάκων . . . 72
11.1.3 Cholesky παραγοντοποίηση . . . 73
11.1.4 Cholesky Solve . . . 74
11.1.5 Solve για συμμετρικό θετικά ορισμένο πίνακα . . . 75
11.1.6 Solve για κάτω τριγωνικό πίνακα . . . 75

Βιβλιογραφία . . . 77

Κεφάλαιο 1

Εισαγωγή

1.1 Υπολογιστικό μοντέλο ροής δεδομένων

Ο νόμος του Moore είναι η παρατήρηση ότι ο αριθμός των τρανζίστορ σε ένα ολοκληρωμένο κύκλωμα διπλασιάζεται περίπου κάθε δύο χρόνια. Η βιομηχανία υπολογιστών κράτησε τον νόμο του Moore ζωντανό μέχρι σήμερα, παρ'ολα αυτά η ισχύς του έχει αρχίσει να φτάνει στο τέλος της.

Εναλλακτικοί τρόποι για να αυξήσουμε την υπολογιστική ισχύ είναι η βελτίωση της σχεδίασης των ολοκληρωμένων κυκλωμάτων και η δημιουργία ειδικού υλικού για την επιτάχυνση συγκεκριμένων αλγορίθμων. Πολλές εταιρίες αναμεσά τους η Microsoft και η Intel δουλεύουν προς την κατεύθυνση να τρέχουν κάποια κομμάτια κώδικα σε έναν νέο τύπο προγραμματιζόμενου ολοκληρωμένου κυκλώματος γενικής χρήσης, γνωστό ως FPGA.

Η συστοιχία επιτόπια προγραμματιζόμενων πυλών (FPGA) είναι ένα ολοκληρωμένο κύκλωμα το οποίο μπορεί να προγραμματιστεί κάθε φορά με διαφορετικό τρόπο. Αντί να είναι περιορισμένο σε μια προκαθορισμένη συνάρτηση υλικού, το FPGA μας επιτρέπει να επαναπρογραμματίζουμε το υλικό του για συγκεκριμένες εφαρμογές, όσες φορές επιθυμούμε. Το FPGA μπορεί να χρησιμοποιηθεί για την υλοποίηση οποιασδήποτε λογικής συνάρτησης και η δυνατότητά του να αλλάζει λειτουργικότητα αποτελεί μεγάλο πλεονέκτημα για πολλές εφαρμογές.

Η εταιρία Maxeler Technologies χρησιμοποιεί τις συσκευές FPGA για την υλοποίηση του υπολογιστικού μοντέλου ροής δεδομένων (dataflow computing), ενός καινοτόμου τρόπου εκτέλεσης υπολογισμών, εντελώς διαφορετικό από τον υπολογισμό με κλασσικούς επεξεργαστές (CPUs). Οι υπολογιστές ροής δεδομένων εστιάζουν στη βελτίωση της κίνησης των δεδομένων μιας εφαρμογής και χρησιμοποιούν παραλληλισμό ανάμεσα σε χιλιάδες πηρύνες ροής δεδομένων (dataflow cores) ώστε να παρέχουν τάξεις μεγέθους βελτίωση στην απόδοση και στην κατανάλωση ισχύος. Για την δημιουργία υπολογιστών ροής δεδομένων η Maxeler τοποθετεί FPGAs της Xilinx μαζί με Intel Xeon επεξεργαστές. Επίσης παρέχει προγράμματα διαχείρησης και μεταγλωτιστές για την εκμετάλλευση του υλικού, το οποίο μπορεί συνεχώς να επαναπρογραμματίζεται ωστε να εκτελεί διαφορετικές εργασίες.

Με την χρήση των μηχανημάτων ροής δεδομένων (DFEs) ο αλγόριθμος αντιστοιχίζεται πάνω σε daraflow cores και τα δεδομένα περνούν από την μνήμη στο DFE όπου εκτελονται οι υπολογισμοί ενώ τα δεδομένα μετακινούνται από το ένα dataflow core στο επόμενο χωρίς να πρέπει να γραφτούν στη μνήμη. Η τεχνολογία αυτή μπορεί να επιτύχει πολυ υψηλή απόδοση λόγω τριών παραγόντων: της υλοποίσης των αριθμητικών συναρτήσεων με σωλήνωση (pipelining), της δημιουργία πολλών παράλληλων σωληνώσεων ανά DFE συσκευή και της χρήσης πολλών DFE συσκευών ανά υπολογιστικό κόμβο.

1.2 Επιτάχυνση αλγορίθμων μηχανικής μάθησης με χρήση του μοντέλου ροής δεδομένων

Η μηχανική μάθηση αποτελεί κλάδο της επιστήμης των υπολογιστών που εξελήχθηκε από την μελέτη της αναγνώρισης μοτίβων και της θεωρίας υπολογιστικής εκμάθησης στην τεχνητή νοημοσύνη. Η μηχανική μάθηση ερευνά την μελέτη και κατασκευή αλγορίθμων οι οποίοι μπορούν να μάθουν και να κάνουν προβλέψεις από σύνολα δεδομένων. Οι αλγόριθμοι αυτοί λειτουργούν χτίζοντας ένα μοντέλο από τις εισόδους-παραδείγματα, ώστε να κάνουν προβλέψεις βασιζόμενοι στα δεδομένα αυτά.

Αν και η μηχανική μάθηση υπάρχει εδώ και δεκαετίες, δύο νέες τάσεις συνέβαλαν στην ευρεία χρήση της τα τελευταία χρόνια:

- Το πλήθος των διαθέσιμων δεδομένων: Στις μέρες μας το πλήθος των ψηφιακών δεδομένων που παράγεται είναι τεράστιο, λόγω των μέσων κοινωνικής δικτύωσης, των κινητών, των αισθητήρων. Τα δεδομένα αυτά μπορούν να αναλυθούν και να παρθούν έξυπνες αποφάσεις βασιζόμενες με μοτίβα που παρατηρούνται.
- Η υπολογιστική ισχύς: Το νέο υλικό έχει την δυνατότητα να αποθηκεύει και να αναλύει τεράστιο όγκο δεδομένων και να εκτελεί πολλούς υπολογισμούς σε λογικά χρονικά πλαίσια, κάτι το οποίο δεν ήταν εφικτό σε προηγούμενες δεκαετίες.

Η μηχανική μάθηση έχει ευρεία εφαρμογή στα οικονομικά, την υγεία, τη διασκέδαση, την ρομποτική και σε πολλούς άλλους τομείς.

Ο σκοπός αυτής της εργασίας είναι η χρήση του μοντέλου ροής δεδομένων της Maxeler για την επιτάχυνση επιλεγμένων αλγορίθμων από την βιβλιοθήκη μηχανικής μάθησης scikit-learn. Η βιβλιοθήκη scikit-learn είναι γραμμένη σε Python και περιλαμβάνει διάφορους αλγορίθμους ταξινόμησης (classification), παλινδρόμησης (regression) και συσταδοποίησης (clustering). Επίσης είναι σχεδιασμένη ώστε πολλοί υπολογισμοί να γίνονται με την χρήση των επιστημονικών βιβλιοθηκών NumPy και SciPy της Python. Ανάμεσα στους αλγορίθμους που παρέχει, οι Gaussian process regression (GPR), Gaussian process classification (GPC) και Kernel ridge regression (KRR) είναι από τους πιο αιτητικούς από άποψη υπολογιστικών πόρων.

Οι κυρίαρχες και με μεγαλύτερη πολυπλοκότητα συναρτήσεις στους GPR, GPC και KRR ειναι οι:

- Cholesky factorization: $A_{n \times n} = L_{n \times n} \cdot L_{n \times n}^{\top}$
- Cholesky Solve: $L_{n \times n} \cdot L_{n \times n}^{\top} \cdot X_{n \times m} = B_{n \times m}$
- Lineal system Solve: $A_{n \times n} \cdot X_{n \times m} = B_{n \times m}$, όπου ο πίνακας A είναι συμμετρικός θετικά ορισμένος ή κάτω τριγωνικός.
- Matrices Multiplication: $C_{m \times n} = A_{m \times k} \cdot B_{k \times n}$

Αξίζει να σημειωθεί ότι οι αντίστοιχες Python συναρτήσεις scipy.linalg.cholesky(), scipy.linalg.cho_solve(), scipy.linalg.solve() και numpy.dot() χρησιμοποιούνται συχνά και για την λύση άλλων προβλημάτων, οπότε αποδοτικές υλοποιήσεις τους στο μοντέλο ροής δεδομένων της Maxeler μπορούν να επηρεάσουν την αποτελεσματικότητα πολλών άλλων εφαρμογών.

Κεφάλαιο 2

Βασικά στοιχεία της μηχανικής μάθησης

2.1 Μηχανική μάθηση

Ορίζουμε την μηχανική μάθηση ως ένα σύνολο μεθόδων που μπορούν να εντοπίσουν αυτόματα μοτίβα στα δεδομένα και στη συνέχεια να τα χρησιμοποιήσουν για να κάνουν προβλέψεις σε μελλοντικά δεδομένα ή να πάρουν αποφάσεις.

2.1.1 Τα είδη μηχανικής μάθησης

Η μηχανική μάθηση μπορεί να χωριστεί σε δύο είδη:

Στην επιτηρούμενη μάθηση, στόχος είναι η μάθηση της αντιστοιχίας από εισόδους $\mathbf{x}$ σε εξόδους y, δεδομένου ενός συνόλου ζευγών εισόδων-εξόδων $\mathcal{D} = \{(\mathbf{x}_i, y_i)|i = 1, ..., n\}$. Το $\mathcal{D}$ ονομάζεται σύνολο εκπαίδευσης (training set) και το n είναι ο αριθμός των δειγμάτων του συνόλου εκπαίδευσης. Στην πιο απλή περίπτωση κάθε είσοδος εκπαίδευσης $\mathbf{x}_i$ είναι ένα D-διάστατο διάνυσμα αριθμών που μπορεί για παράδειγμα να αντιπροσωπεύει το ύψος και το βάρος ενός ατόμου, τα οποία ονομάζονται χαρακτριστικά (features). Γενικότερα όμως, το $\mathbf{x}_i$ μπορεί να είναι ένα πιο περίπλοκο αντικείμενο, όπως μια εικόνα ή μια πρόταση. Με τον ίδιο τρόπο μια έξοδος μπορεί να είναι ουσιαστικά οτιδήποτε, αν και οι περισσότερες μέθοδοι υποθέτουν ότι το y_i είναι στοιχείο ενός συνόλου, $y_i \in \{1, ..., C\}$, ή ότι το y_i είναι ένας πραγματικός αριθμός. Στην πρώτη περίπτωση το πρόβλημα ονομάζεται ταξινόμηση (classification), ενώ στην δεύτερη παλινδρόμηση (regression).

Το δεύτερο είδος μηχανικής μάθησης ονομάζεται μη επιτηρούμενη μάθηση. Σε αυτήν την περίπτωση δίνονται μόνο είσοδοι, $\mathcal{D} = \{\mathbf{x}_i|i = 1, ..., n\}$ και ο στόχος είναι να βρεθούν ενδιαφέροντα μοτίβα στα δεδομένα. Το πρόβλημα αυτό δεν είναι ορισμένο με μεγάλη σαφήνεια, καθώς δεν γνωρίζουμε για τι είδους μοτίβα ψάχνουμε και δεν υπάρχει κάποια προφανής μετρική λάθους που μπορεί να χρησιμοποιηθεί.

2.2 Επιτηρούμενη μάθηση

2.2.1 Ταξινόμηση

Σκοπός της ταξινόμησης είναι η εκμάθηση της αντιστοιχίας μεταξύ των εισόδων $\mathbf{x}$ και των εξόδων y, όπου $y_i \in \{1, ..., C\}$ και C είναι ο αριθμός των κλάσεων. Αν $C = 2$, το πρόβλημα ονομάζεται δυαδική ταξινόμηση και θεωρούμε ότι $y \in -1, +1$. Αν οι κλάσεις δεν είναι αμοιβαία αποκλειόμενες και ένα στοιχείο μπορεί να ανήκει σε παραπάνω από μια κλάση το προβλημα ονομάζεται multi-label ταξινόμηση.

Ένας τρόπος να δούμε το πρόβλημα είναι με προσέγγιση συνάρτησης. Υποθέτουμε ότι $y = f(\mathbf{x})$ για κάποια άγνωστη συνάρτηση f, και ο στόχος της μάθησης είναι να προσεγγίσουμε την συνάρτηση f εφόσον μας δωθεί ένα σύνολο εισόδων εκπαίδευσης, και τη συνέχεια να κάνουμε προβλέψεις για νέες εισόδους οι οποίες δεν υπάρχουν στις εισόδους εκπαίδευσης.

Παράδειγμα ταξινόμησης ψηφίων

Ένα πολύ γνωστό παράδειγμα είναι η ταξινόμηση εικόνων χειρόγραφων αριθμών (ψηφίων). Το σύνολο εκπαίδευσης αποτελείται από μικρές ψηφιοποιημένες εικόνες, κάθε μια από τις οποίες συνοδεύεται με έναν αριθμό από τους $0, ..., 9$. Σκοπός μας είναι να μάθουμε την αντιστοίχιση ανάμεσα σε εικόνα και κλάση $(0, ..., 9)$ και να την εφαρμόσουμε σε νέες εικόνες ψηφίων. Η επιτηρούμενη μάθηση είναι ένας πιθανός τρόπος αντιμετώπισης αυτού του προβλήματος, μιας και δεν είναι εύκολο με κάποιον άλλο συστηματικό τρόπο να καθορίσουμε τα χαρακτηριστικά ενός χειρόγραφου ψηφίου.

2.2.2 Παλινδρόμηση

Η παλινδρόμηση είναι παρόμοιο πρόβλημα με την ταξινόμηση, με την διαφορά ότι οι έξοδοι είναι πραγματικοί αριθμοί. Στο σχήμα 2.1 φαίνεται ένα απλό παράδειγμα: έχουμε εισόδους που είναι πραγματικοί αριθμοί (ένα χαρακτηριστικό) $\mathbf{x}_i \in \mathbb{R}$ και εξόδους πραγματικούς αριθμούς $y_i \in \mathbb{R}$. Εκπαιδεύουμε δύο μοντέλα με τα δεδομένα εκπαίδευσης, ένα γραμμικό και ένα τετραγωνικό.

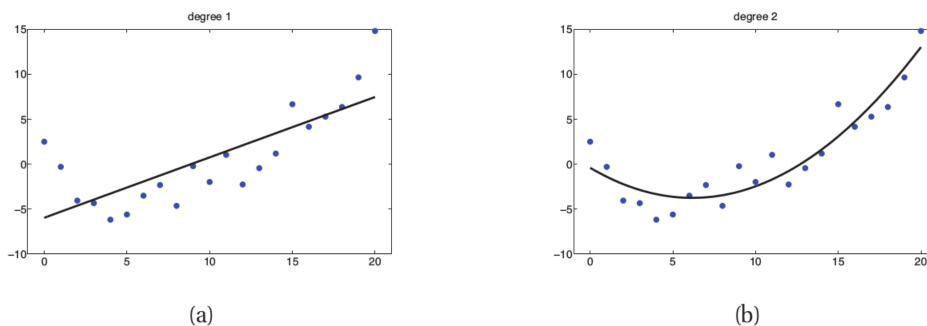


Figure 2.1: (a) Γραμμική παλινδρόμηση (linear regression) σε 1-D δεδομένα. (b) Πολυωνυμική παλινδρόμηση (polynomial regression) στα ίδια δεδομένα.

Κεφάλαιο 3

Αλγόριθμοι μηχανικής μάθησης της βιβλιοθήκης scikit-learn

Το scikit-learn είναι μια βιβλιοθήκη μηχανικής μάθησης σε Python. Παρέχει διάφορους αλγορίθμους για ταξινόμηση, παλινδρόμηση και συσταδοποίηση, συμπεριλαμβανομένων των Gaussian processes, linear models, random forests, gradient boosting, k-means. Οι περισσότεροι υπολογισμοί γίνονται με κλήσεις σε συναρτήσεις των NumPy και SciPy βιβλιοθηκών.

Ένα μέρος της εργασίας αυτής είναι η εύρεση αλγορίθμων ταξινόμησης και παλινδρόμησης που θα μπορούσαν να επιταχυνθούν με την χρήση του μοντέλου ροής δεδομένων της Maxeler. Για τον σκοπό αυτόν επιλέχθηκαν οι παρακάτω αλγόριθμοι:

- Quadratic Discriminant Analysis classification
- Gaussian processes classification
- Naive Bayes classification
- Support Vector Machines classification
- Stochastic Gradient Descent (SGD) classification
- Kernel ridge regression
- Gaussian processes regression

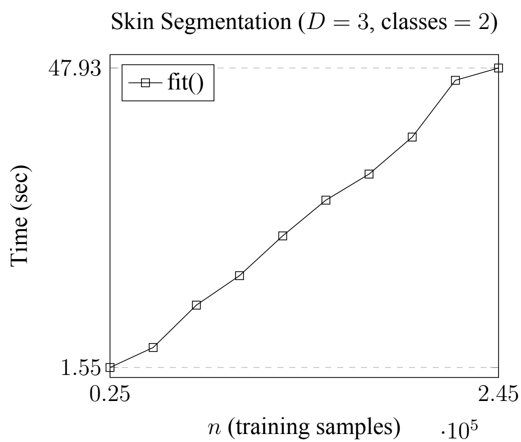
Κάθε αλγόριθμος ταξινόμησης ή παλινδρόμησης της βιβλιοθήκης scikit-learn περιέχεται σε μια Python κλάση που υλοποιεί τις μεθόδους fit($X, \mathbf{y}$) και predict(X_*), όπου X είναι ο πίνακας εκπαίδευσης μεγέθους $n \times D$, $\mathbf{y}$ είναι το διάνυσμα εκπαίδευσης και X_* είναι ο πίνακας με τα δεδομένα στα οποία θα γίνει η πρόβλεψη, με μέγεθος $n_* \times D$. Το πρώτο βήμα για την επιλογή των αλγορίθμων που θα επιταχυνθούν είναι μια απλή ανάλυση του χρόνου εκτέλεσης των μεθόδων fit() και predict() για διάφορα μεγέθη των n και n_*. Τα σύνολα δεδομένων που χρησιμοιήθηκαν για τη διαδικασία αυτή είναι από το UCI Machine Learning Repository.

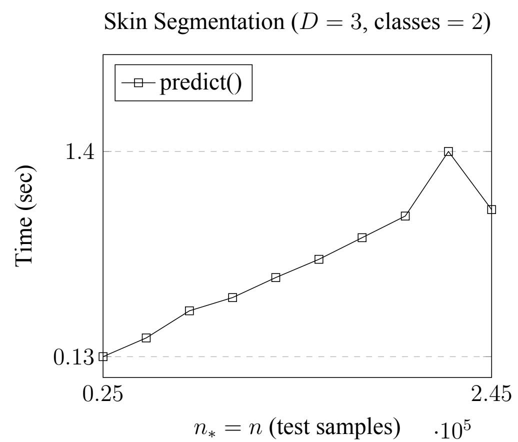
3.1 Ανάλυση χρόνων εκτέλεσης

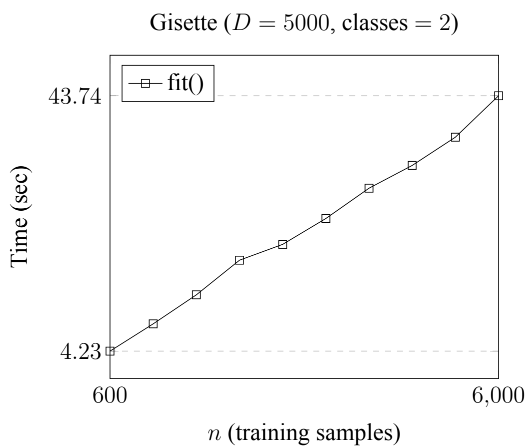
Στην ενότητα αυτή παρουσιάζονται τα αποτελέσματα για τους χρόνους εκτέλεσης των μεθόδων fit() και predict() των αλγορίθμων που απαριθμούνται στην αρχή του κεφαλαίου. Οι πίνακες εισόδου κατασκευάστηκαν από υποσύνολα των δεδομένων του UCI Machine Learning Repository.

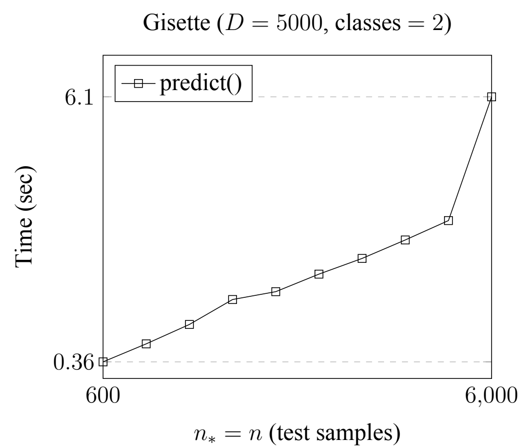
Στόχος μας είναι να εντοπίσουμε τους αλγόριθμους ταξινόμησης και παλινδρόμησης με την μεγαλύτερη χρονική πολυπλοκότητα σε σχέση με το μέγεθος των εισόδων (n και n_*).

3.1.1 Stochastic Gradient Descent (SGD) classification

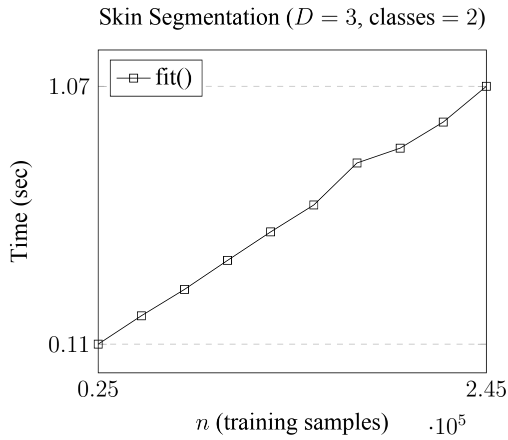


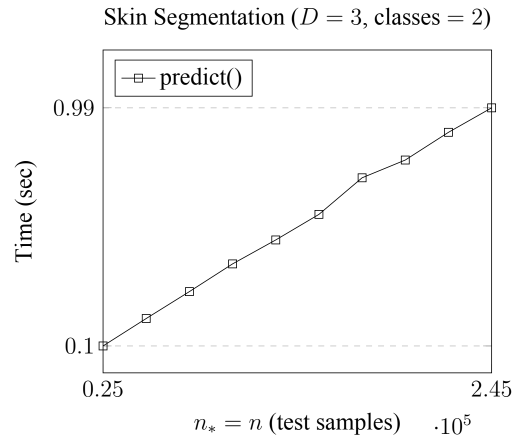


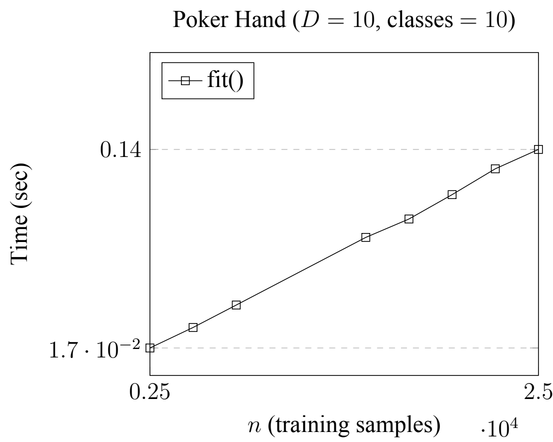


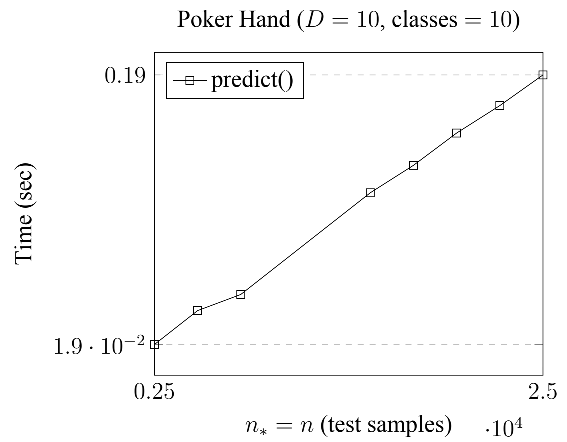


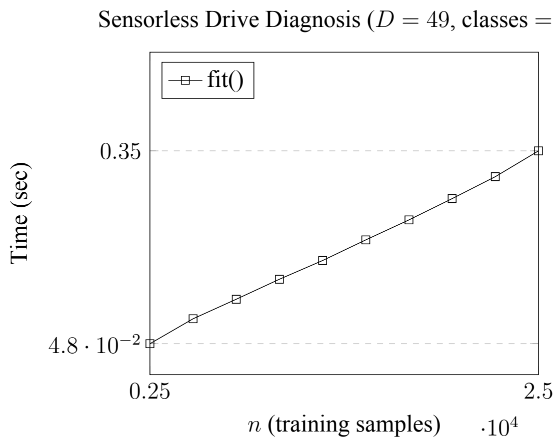
3.1.2 Quadratic Discriminant Analysis classification

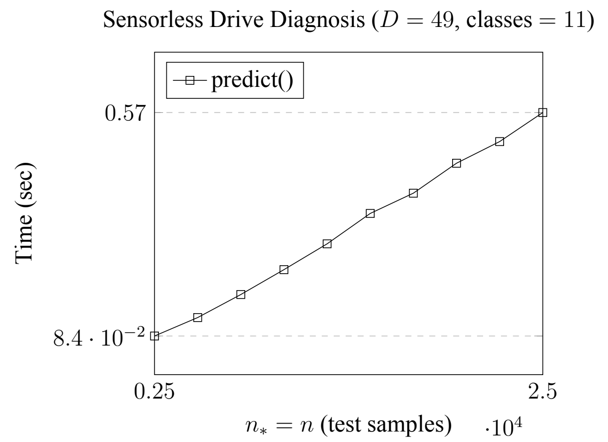


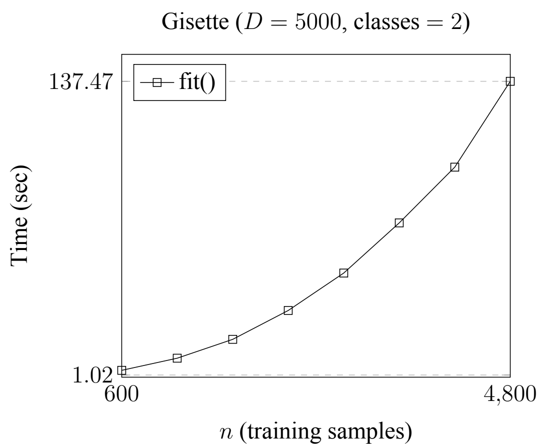


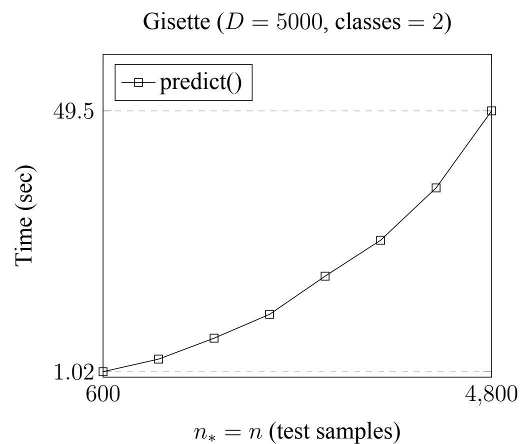




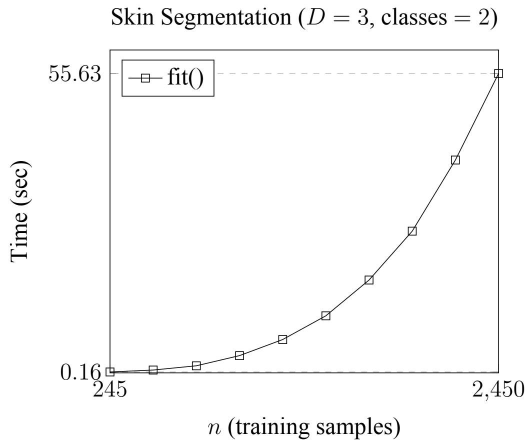


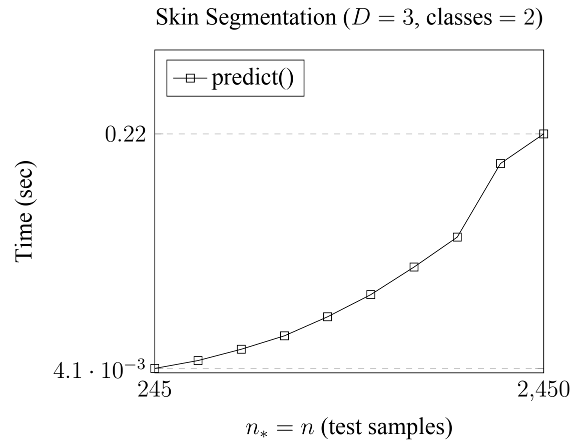


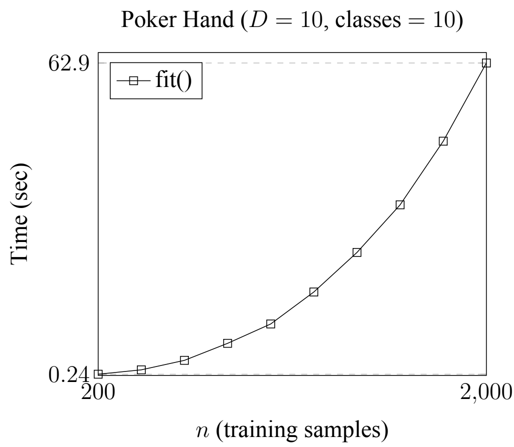


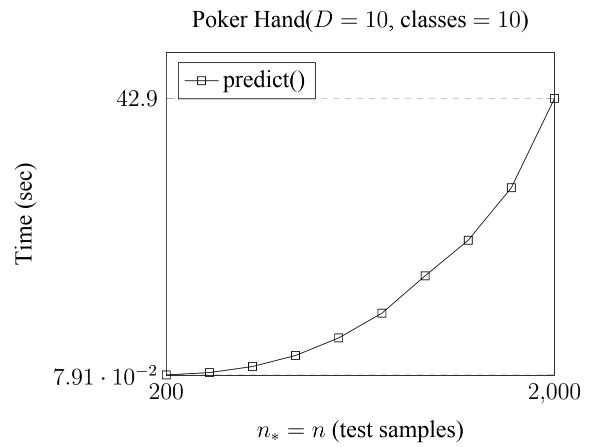


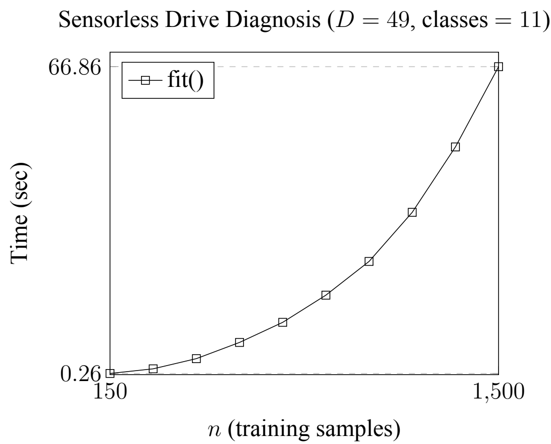
3.1.3 Gaussian processes classification

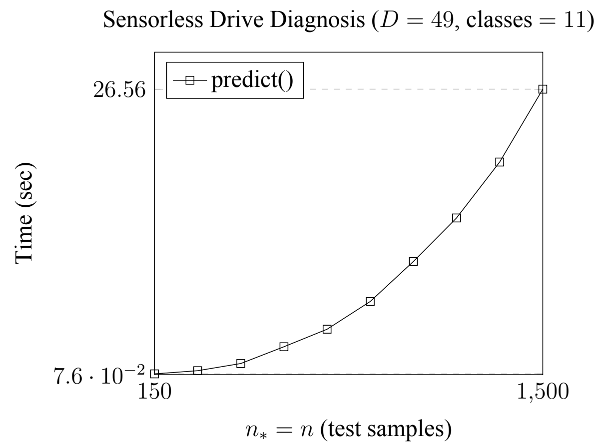


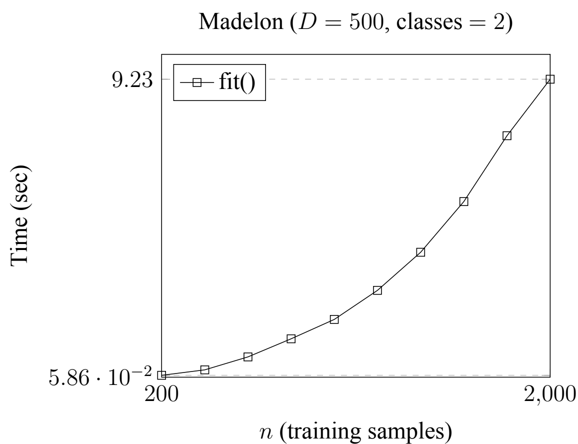


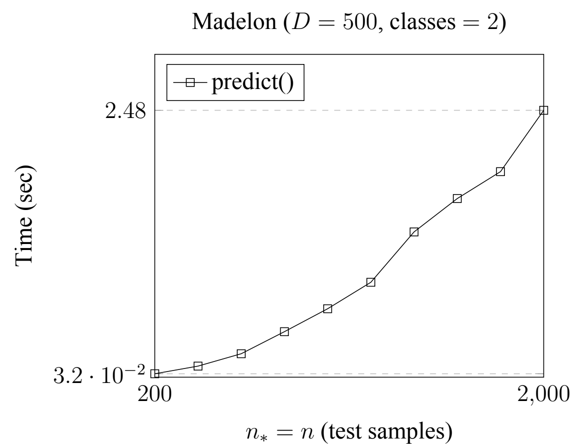




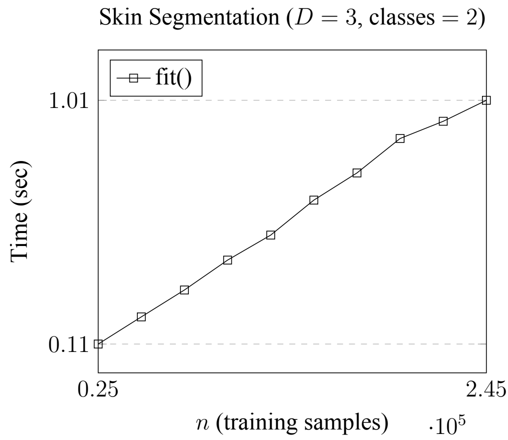


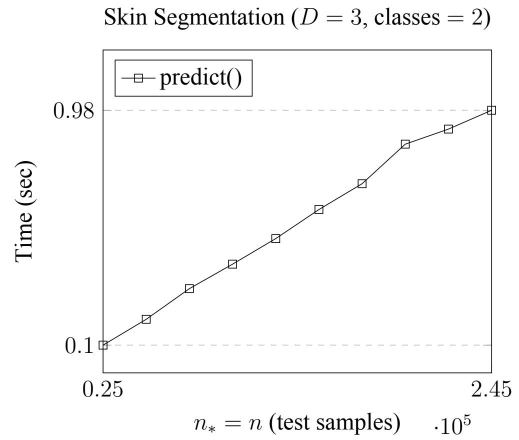


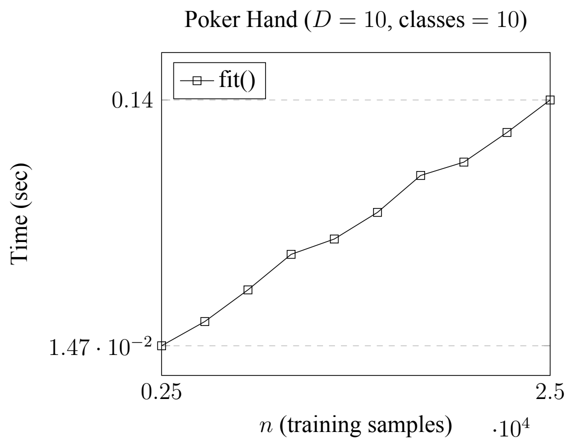


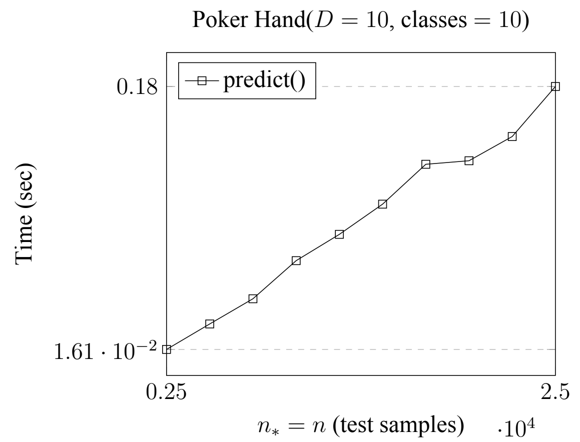


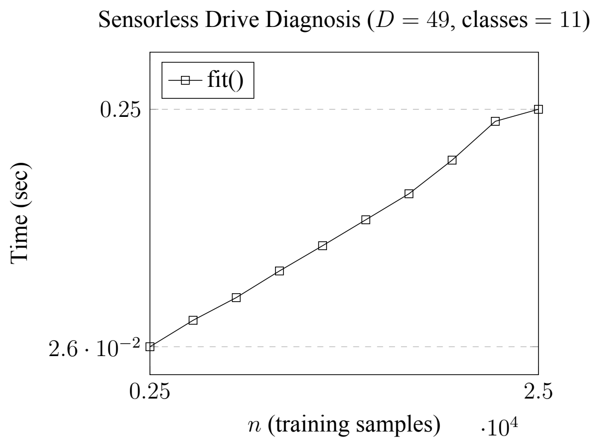
3.1.4 Naive Bayes classification

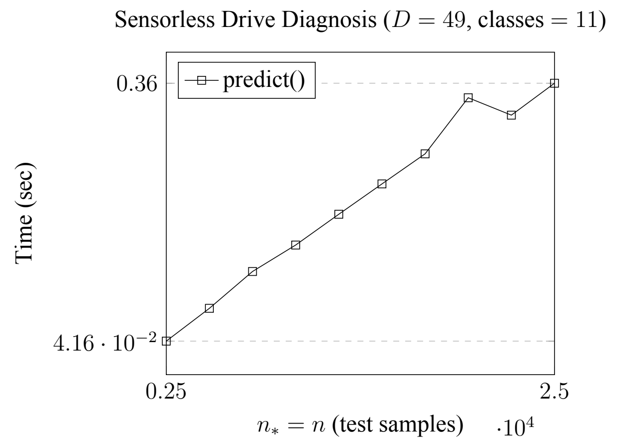


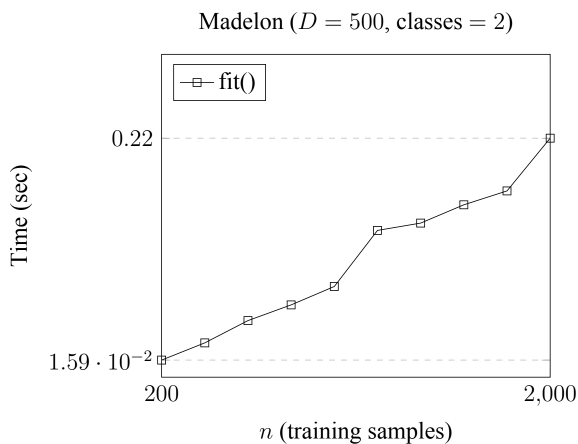


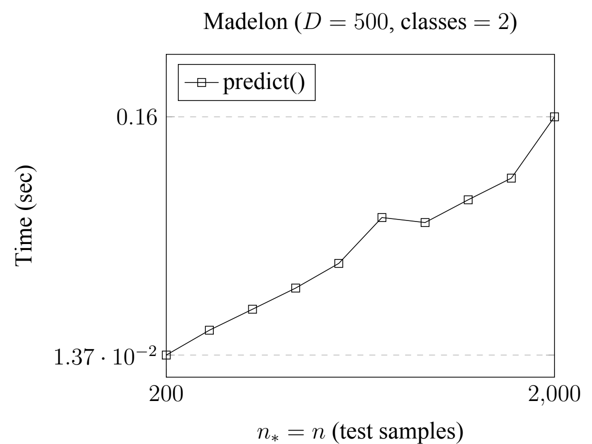




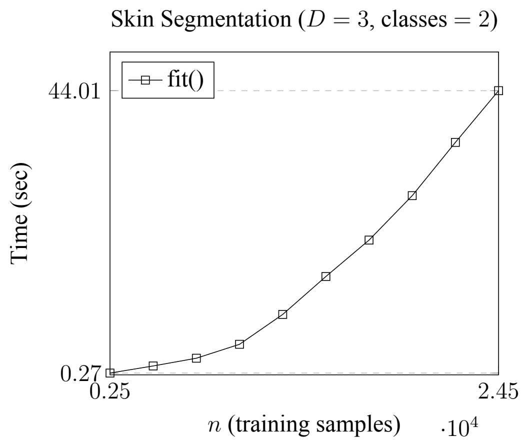


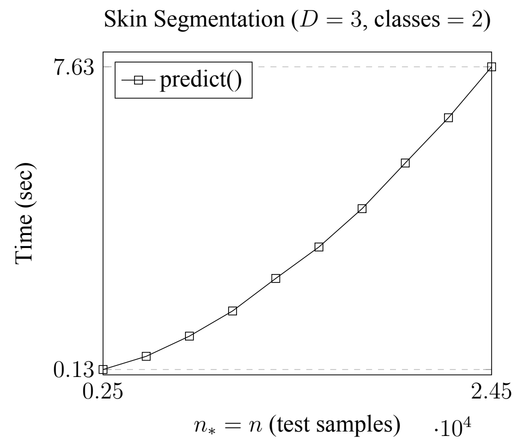


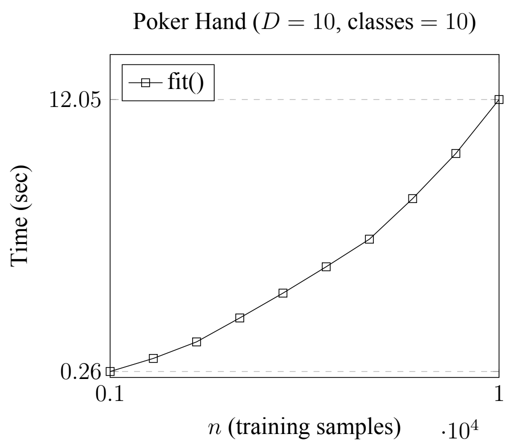


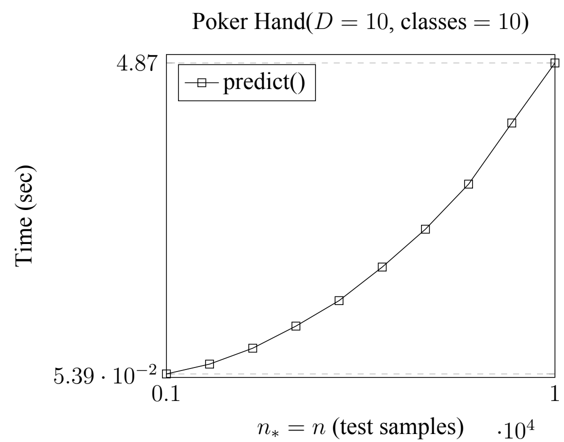


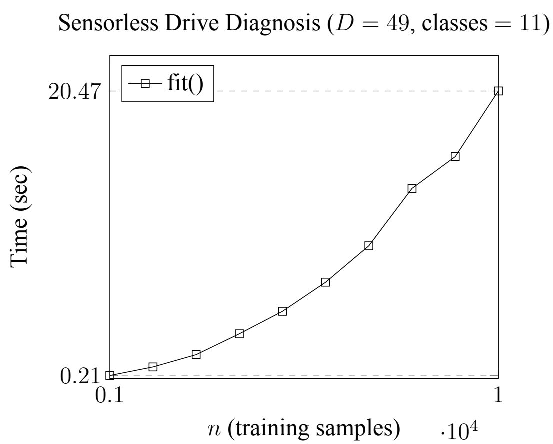
3.1.5 Support Vector Machines classification

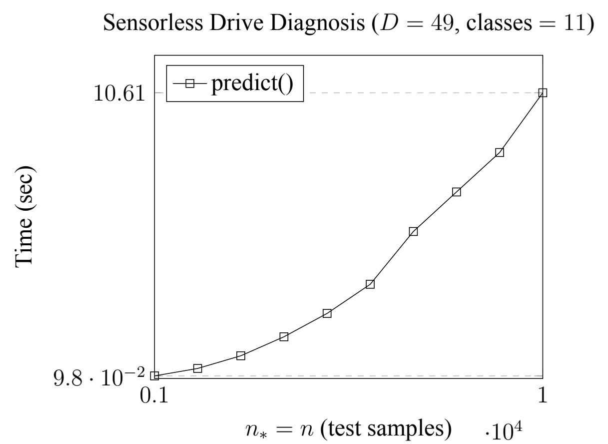


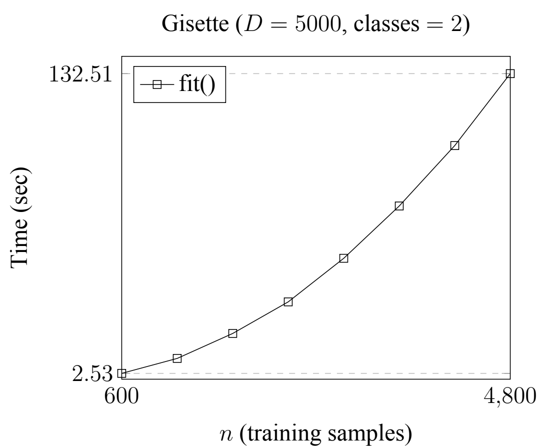


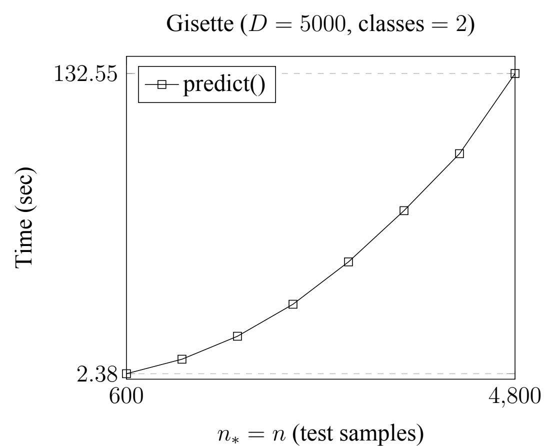




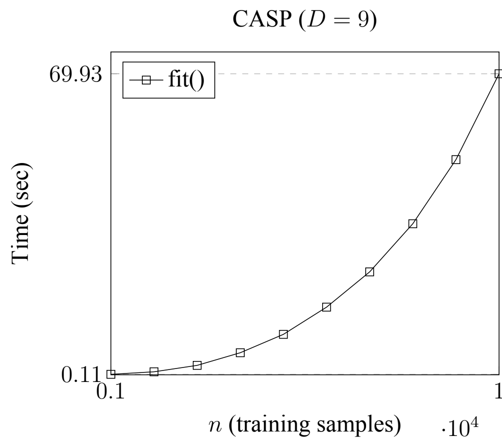


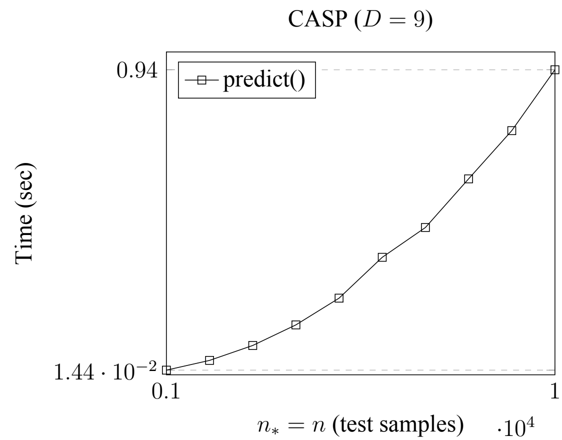


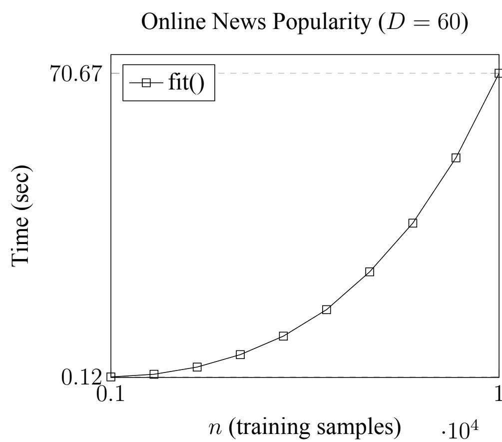


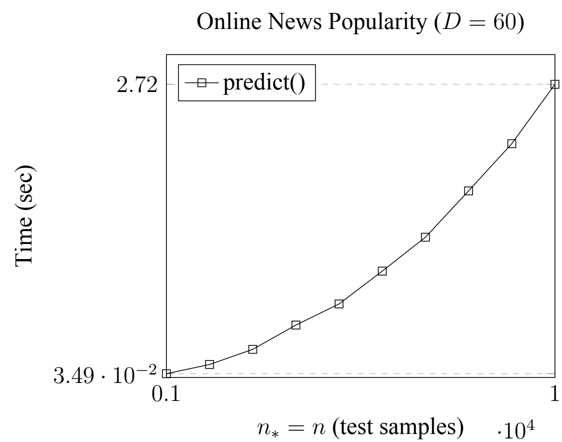


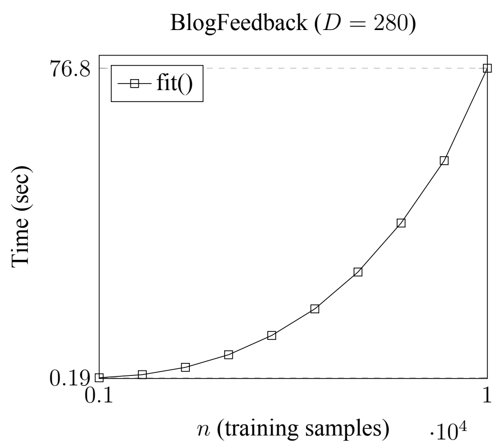
3.1.6 Kernel ridge regression

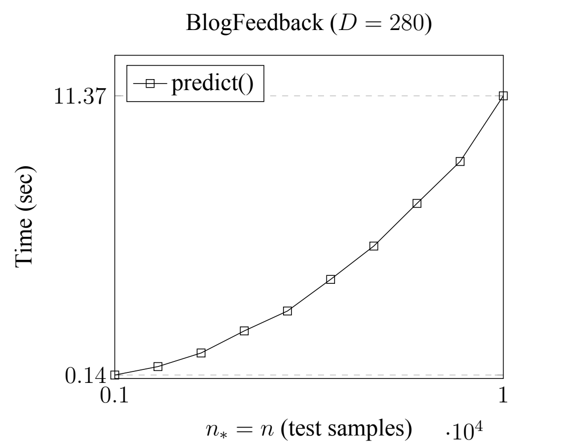


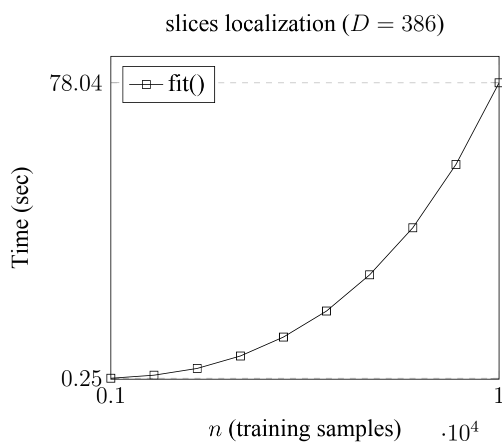


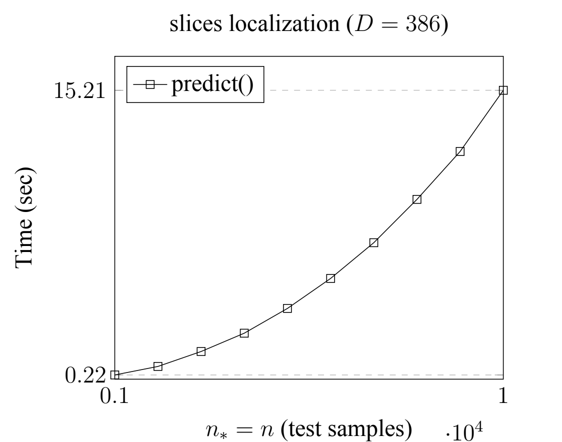




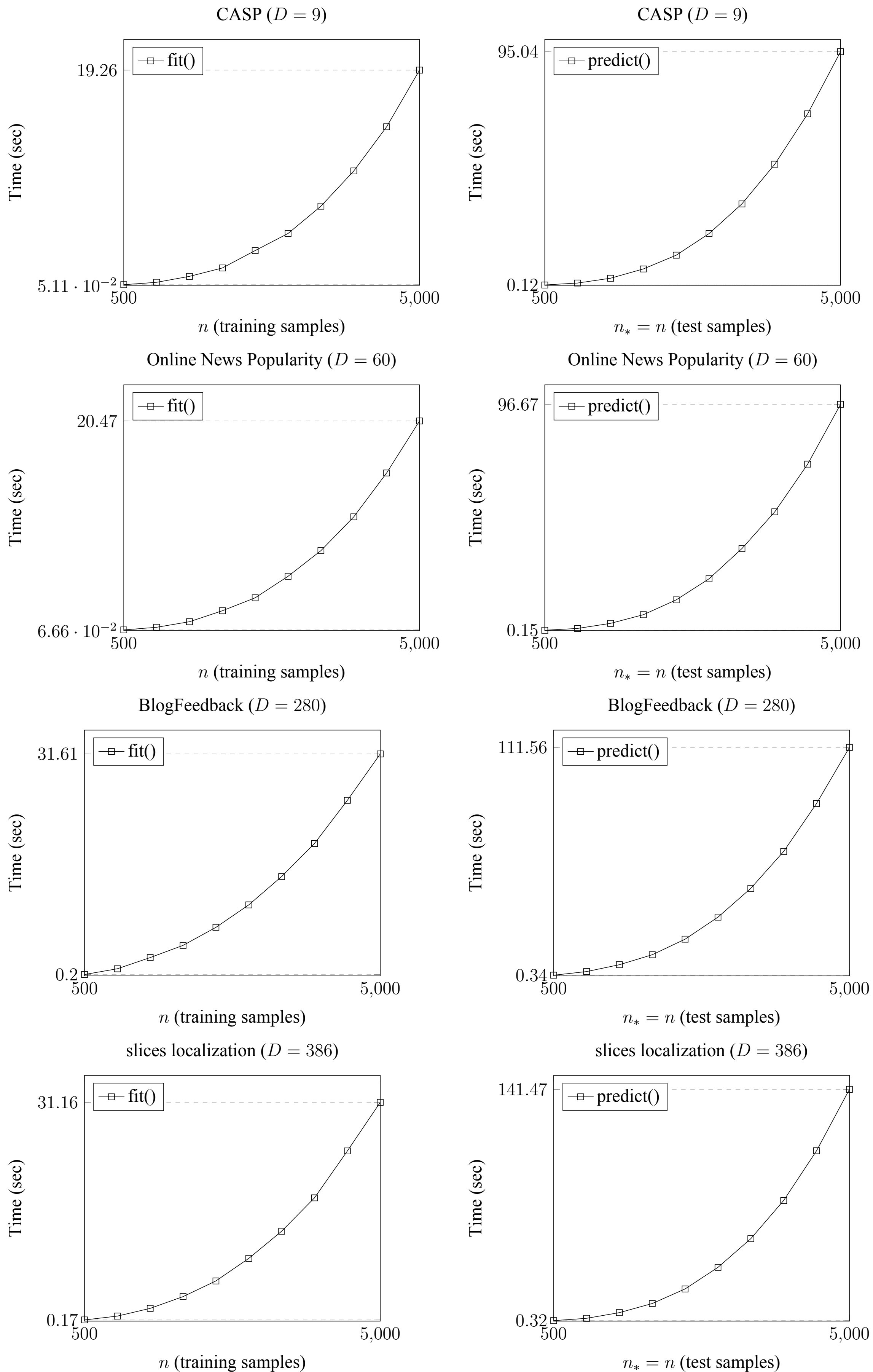








3.1.7 Gaussian processes regression



3.2 Αλγόριθμοι της scikit-learn για επιτάχυνση με DFEs

Οι αλγόριθμοι με χρόνους εκτέλεσης που δεν αυξάνονται γραμμικά με το n και το n_* είναι αυτοί που θα ληφθούν υπόψιν για επιτάχυνση με χρήση του μοντέλου ροής δεδομένων. Γενικότερα ψάχνουμε για αλγορίθμους με υπολογιστική πολυπλοκότητα μεγαλύτερη από το μέγεθος είσόδου, καθώς το γεγονός αυτό μπορεί να οδηγήσει σε αποδοτικές DFE υλοποιήσεις με παράλληλες σωληνώσεις.

Όπως μπορούμε να δούμε από τα γραφήματα, οι χρόνοι εκτέλεσης των αλγορίθμων Stochastic Gradient Descent and Naive Bayes αυξάνονται σχεδόν γραμμικά με το n και το n_*, και επομένως μια υλοποίηση τους σε DFE μπορεί να μην είναι τόσο αποτελεσματική. Επίσης ο χρόνος εκτέλεσης για τον αλγόριθμο Quadratic Discriminant Analysis αυξάνει με μη γραμμικό τρόπο μόνο για το σύνολο δεδομένων Gisette, όπου ο αριθμός των χαρακτηριστικών είναι ασυνήθιστα μεγάλος ($D = 5000$). Όμως τα περισσότερα σύνολα δεδομένων έχουν μικρό αριθμό από χαρακτηριστικά και επομένως μια DFE υλοποίηση του αλγορίθμου δεν θα είχε μεγάλη αξία. Τα Support Vector Machines από την άλλη παρουσιάζουν μια πιο ενδιαφέρουσα συμπεριφορά στο χρόνο εκτελεσης.

Από την άλλη οι χρόνοι εκτέλεσης των fit και predict μεθόδων των αλγορίθμων Gaussian processes classification, Gaussian processes regression και Kernel ridge regression αυξάνουν μη γραμμικά με το n και το n_*. Για μεγάλα μεγέθη εισόδων οι μέθοδοι fit και predict είναι πολύ χρονοβόροι και το μοντέλο ροής δεδομένων της Maxeler θα μπορούσε να χρησιμοποιηθεί για να πάρουμε καλύτερη απόδοση.

Στα επόμενα κεφάλαια παρουσιάζονται οι αλγόριθμοι των **Gaussian processes classification**, **Gaussian processes regression** και **Kernel ridge regression**. Επίσης μετά από ανάλυση καταλήγουμε στα κομμάτια των αλγορίθμων που έχουν την μεγαλύτερη χρονική πολυπλοκότητα και μπορούν να επιταχυνθούν.

Κεφάλαιο 4

Gaussian Process Regression

4.1 Αλγόριθμος Gaussian process regression

Μια πρακτική υλοποίηση του Gaussian process regression (GPR) φαίνεται στον Αλγόριθμο 4.1. Ο αλγόριθμος χρησιμοποιεί την Cholesky παραγοντοποίηση, αντί για την απευθείας αντιστροφή πίνακα, εφόσον είναι ταχύτερη. Ο αλγόριθμος επιστρέφει τον μέσο όρο και τη διακύμανση για τα δεδομένα που γίνεται η πρόβλεψη. Τις περισσότερες φορές, ως πρόβλεψη παίρνουμε τον μέσο όρο, αλλά εδώ υπολογίζουμε και την διακύμανση των προβλέψεων.

Algorithm 4.1 Αλγόριθμος Gaussian process regression

1: **procedure** GPR(X(inputs), **y**(targets), k(covariance function), ${\sigma_n}^2$(noise), $\mathbf{x}_*$(test input))
2: $L \leftarrow \text{cholesky}(K + {\sigma_n}^2 I)$
3: $\alpha \leftarrow (\mathbf{y}/L)/L^\top$
4: $\bar{f}_* \leftarrow \mathbf{k}_*^\top \alpha$
5: $\mathbf{v} \leftarrow \mathbf{k}_*/L$
6: $\mathbb{V}[f_*] \leftarrow k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{v}^\top \mathbf{v}$
7: $\log p(\mathbf{y}|X) \leftarrow -\frac{1}{2}\mathbf{y}^\top \alpha - \sum_i \log L_{ii} - \frac{n}{2} \log 2\pi$
8: **return** $\bar{f}_*$ (mean), $\mathbb{V}[f_*]$ (variance), $\log p(\mathbf{y}|X)$ (log marginal likelihood)

Αλγόριθμος 4.1: Η υλοποίηση αυτή αντιμετωπίζει το πρόβλημα της αντιστροφής πίνακα χρησιμοποιώντας την Cholesky παραγοντοποίηση. Οι γραμμές 4-6 επαναλαμβάνονται στην περίπτωση παραπάνω από μιας εισόδου για πρόβλεψη. Η υπολογιστική πολυπλοκότητα είναι $n^3/6$ για την Cholesky παραγοντοποίηση στη γραμμή 2, $n^2/2$ για τη λύση του συστήματος στην γραμμή 3 και $n^2/2$ για κάθε είσοδο για πρόβλεψη στην γραμμή 5.

4.2 scikit-learn υλοποίηση του Gaussian process regression

Στην scikit-learn βιβλιοθήκη ο Gaussian process regression αλγόριθμος υλοποιείται από μια Python κλάση με τις μεθόδους fit(X,**y**) και predict(X_*). Η κλάση αυτή ονομάζεται GaussianProcessRegressor. Ο κατασκευαστής της κλάσης παίρνει ως ορίσματα τις παραμέτρους του μοντέλου. Στην περίπτωση που μας ενδιαφέρει ο κατασκευαστής της κλάσης, παίρνει μια παράμετρο, την kernel συνάρτηση RBF.

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
fixed_kernel = RBF(length_scale=1.0, length_scale_bounds="fixed")
gpr = GaussianProcessRegressor(kernel=fixed_kernel)
```

Αρχικά καλέσαμε τον κατασκευαστή της κλάσης. Σειρά έχει η εκπαίδευση του μοντέλου, οπότε καλούμε την μέθοδο fit με εισόδους τους πίνακες εκπαίδευσης.

```
X_train, y_train, X_test = initialize()
gpr.fit(X_train, y_train)
```

Τώτα μπορούμε να προβλέψουμε νέες τιμές, οι οποίες θα είναι οι μέσοι όροι της πρόβλεψης για τον πίνακα εισόδου του predict. Επίσης ζητάμε από την μέθοδο predict να μας επιστρέψει και την διακύμανση των προβλεψεων, θέτοντας return_cov=True.

```
y_pred, y_cov = gpr.predict(X_test, return_cov=True)
```

Στη συνέχεια παρουσιάζουμε τις μεθόδους fit() και predict() της Python κλάσης Gaussian-ProcessRegressor(), που βασίζονται στον Αλγόριθμο 4.1. (Παρακάτω το self.var σημαίνει ότι το var είναι attribute της Python κλάσης)

Algorithm 4.2 scikit-learn GaussianProcessRegressor().fit() μέθοδος

1: **procedure** fit(X(inputs), **y**(targets))
2: $X, \mathbf{y} \leftarrow$ check_X_y($X, \mathbf{y}$, ...)
3: $\mathbf{y} \leftarrow$ normalize_y()
4: self.log_marginal_likelihood_value_ $\leftarrow$ self.log_marginal_likelihood()
5: $K \leftarrow$ self.kernel_(X)
6: $K \leftarrow K + {\sigma_n}^2 I$
7: self.L_ $\leftarrow$ cholesky(K)
8: self.alpha_ $\leftarrow$ cho_solve(self.L_ , **y**)

Αλγόριθμος 4.2: Στη γραμμή 3, το διάνυσμα **y** μπορεί να κανονικοποιηθεί αφαιρώντας τον μέσο όρο του **y** από κάθε στοιχείο του. Η τιμή log_marginal_likelihood_value υπολογίζεται στην γραμμή 4, χρησιμοποιώντας την αντίστοιχη μέθοδο που παρουσιάζεται παρακάτω. Η kernel συνάρτηση στη γραμμή 5 είναι η προκαθορισμένη RBF και η συνάρτηση kernel_() παρουσιάζεται στον Αλγόριθμο 4.4. Επιπλέον η Cholesky παραγοντοποίηση στη γραμμή 6, υπολογίζεται με τη συνάρτηση scipy.linalg.cholesky() και η τιμή του alpha_ υπολογίζεται με τη συνάρτηση scipy.linalg.cho_solve(). Η συνολική πολυπλοκότητα της μεθόδου fit είναι $\mathcal{O}(n^3)$, λόγω της Cholesky παραγοντοποίησης.

Algorithm 4.3 scikit-learn GaussianProcessRegressor().log_marginal_likelihood() μέθοδος

1: **procedure** log_marginal_likelihood()
2: $K \leftarrow$ self.kernel_(X)
3: $K \leftarrow K + {\sigma_n}^2 I$
4: $L \leftarrow$ cholesky(K)
5: alpha $\leftarrow$ cho_solve(L , **y**)
6: log_likelihood $\leftarrow -\frac{1}{2}\mathbf{y}^\top \cdot (\text{alpha}) - \sum_i \log L_{ii} - \frac{n}{2}\log 2\pi$ **return** log_likelihood

Αλγόριθμος 4.3: Συνολικό κόστος: $\mathcal{O}(n^3)$.

Algorithm 4.4 scikit-learn kernels.RBF()

1: **procedure** kernel_rbf(X_1(test inputs), X_1(training inputs))
2: **if** $X_2 =$ None **then**
3: dists $\leftarrow$ pdist(X_1, metric='sqeuclidean')
4: K = np.exp(-0.5 * dists)
5: K = squareform(K)
6: np.fill_diagonal(K, 1)
7: **else**
8: dists = cdist(X_1, X_2, metric='sqeuclidean')
9: K = np.exp(-0.5 * dists)
10: **return** K

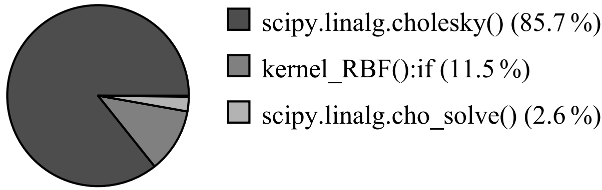
Algorithm 4.4: Για τον RBF kernel, η συνάρτηση kernel είναι $k(\mathbf{x}, \mathbf{x}') = \exp(-\frac{1}{2}|\mathbf{x} - \mathbf{x}'|^2)$, όπου $|\mathbf{x} - \mathbf{x}'|^2 = \sum_{d=1}^{D}(x_d - {x'}_d)^2$. Όλες οι αποστάσεις υπολογίζονται με τις συναρτήσεις scipy.spatial.distance.pdist() and scipy.spatial.distance.cdist(). Η συνολική χρονική πολυπλοκότητα είναι $\mathcal{O}(n_1^2 D)$ για το if και $\mathcal{O}(n_1 \cdot n_2 D)$ για το else. Σημειώνουμε ότι n_1 είναι ο αριθμός των δειγματών στον X_1, n_2 είναι ο αριθμός των δειγματών στον X_2 και D είναι ο αριθμός των χαρακτηριστικών.

Algorithm 4.5 scikit-learn GaussianProcessRegressor().predict() μέθοδος

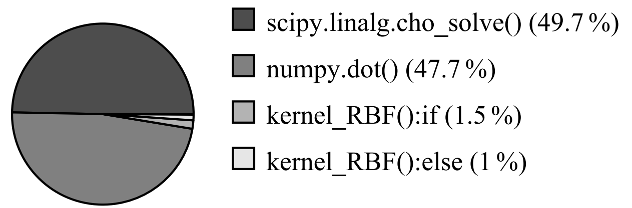
1: **procedure** predict(X_*(test inputs))
2: $X_* \leftarrow$ check_array(X_*)
3: $K_* \leftarrow$ self.kernel_(X_*, X)
4: $\mathbf{y}_{mean} \leftarrow K_* \cdot$ (self.alpha_)
5: $\mathbf{y}_{mean} \leftarrow$ undo_normal($\mathbf{y}_{mean}$)
6: $\mathbf{v} \leftarrow$ cho_solve(self.L_, $K_*^\top$)
7: $\mathbf{y}_{cov}$ = self.kernel_(X_*) $-K_* \cdot \mathbf{v}$
8: **return** $\mathbf{y}_{mean}$(predictions), $\mathbf{y}_{cov}$(variance matrix)

Algorithm 4.5: Οι κυρίαρχες συναρτήσεις στην μέθοδο predict() είναι οι cho_solve() στη γραμμή 6 με πολυπλοκότητα $\mathcal{O}(n^2 \cdot n_*)$ και το dot() στη γραμμή 7 με πολυπλοκότητα $\mathcal{O}(n \cdot n_*^2)$. Επιπλέον, ο υπολογισμός του kernel στη γραμμή 3 χρειάζεται $\mathcal{O}(n \cdot n_* \cdot D)$ πράξεις, ενώ το kernel στη γραμμή 7 έχει κόστος $\mathcal{O}(n_*^2 \cdot D)$ πράξεων.

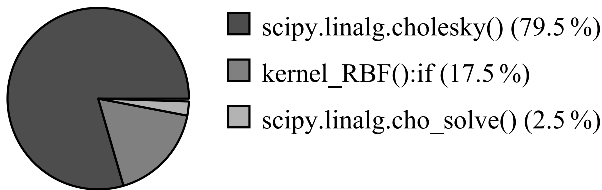
4.3 Ανάλυση scikit-learn Gaussian process regression



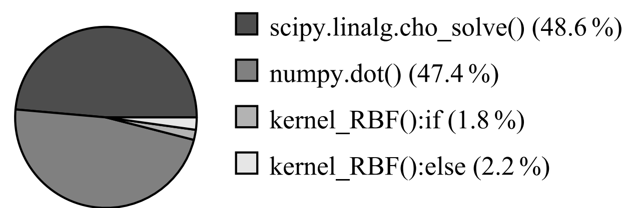
fit()
CASP Dataset $D = 9$, $n = 5000$
Συνολικός χρόνος: 19.38 sec



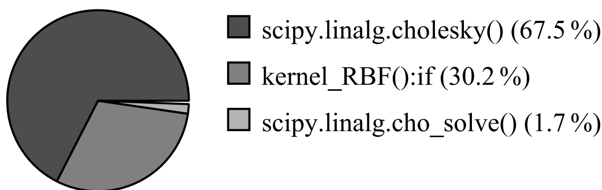
predict()
CASP Dataset $D = 9$, $n = 5000$, $n_* = 5000$
Συνολικός χρόνος: 96.93 sec



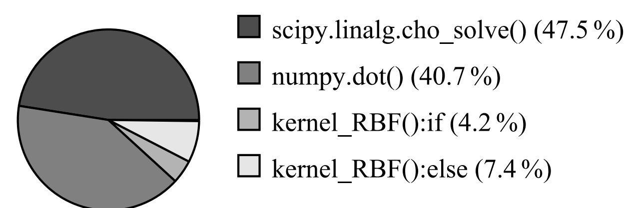
fit()
Online News Popularity Dataset $D = 60$, $n = 5000$
Συνολικός χρόνος: 20.53 sec



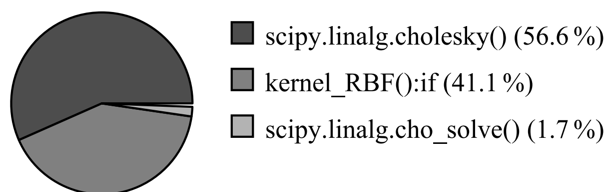
predict()
Online News Popularity Dataset $D = 60$, $n = 5000$, $n_* = 5000$
Συνολικός χρόνος: 98.12 sec



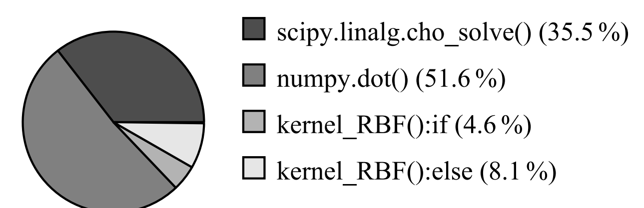
fit()
BlogFeedback Dataset $D = 280$, $n = 5000$
Συνολικός χρόνος: 31.57 sec



predict()
BlogFeedback Dataset $D = 280$, $n = 5000$, $n_* = 5000$
Συνολικός χρόνος: 113.54 sec



fit()
slices localization Dataset $D = 386$, $n = 5000$
Συνολικός χρόνος: 30.58 sec



predict()
slices localization Dataset $D = 386$, $n = 5000$, $n_* = 5000$
Συνολικός χρόνος: 138.1 sec

Ο σκοπός μας είναι να εντοπίσουμε τις πιο απαιτητικές χρονικά συναρτήσεις στην scikit-learn υλοποίηση του Gaussian process regression. Τοποθετούμε μετρητές χρόνου σε διάφορα σημεία του κώδικα των μεθόδων fit() και predict(). Τα προηγούμενα διαγράμματα παρουσιάζουν τα ποσοστά χρόνου που κατέχει κάθε συνάρτηση μέσα στις Python μεθόδους. Οι μέθοδοι fit και predict εκτελέστηκαν για τέσσερα διαφορετικά σύνολα δεδομένων (datasets) με διαφορετικά μεγέθη χαρακτηριστικών και μεγάλα μεγέθη εισόδων για εκπαίδευση και πρόβλεψη ($n = 5000, n_* = 5000$).

Πρέπει να επισημάνουμε ότι το kernel_RBF():if αποτελείται από το scipy.spatial.distance.pdist(), το numpy.exp(), το scipy.spatial.distance.squareform() και το numpy.fill_diagonal(). Επίσης, το kernel_RBF():if αποτελείται από τα scipy.spatial.distance.cdist() and numpy.exp().

Από τα διαγράμματα μπορούμε να παρατηρήσουμε ότι το scipy.linalg.cholesky() είναι η πιο απαιτητική χρονικά συνάρτηση στη μέθοδο fit, ενώ ο χρόνος εκτέλεσης του kernel_-RBF():if αυξάνει όσο μεγαλώνει ο αριθμός των χαρακτηριστικών του dataset. Επιπλέον, οι συναρτήσεις scipy.linalg.cho_solve() και numpy.dot() καταλαμβάνουν το μεγαλύτερο ποσοστό χρόνου στη μέθοδο predict, μοιραζόμενες σχεδόν τον συνολικό χρόνο στην μέση. Παρ'ολα αυτά, όσο ο αριθμός των χαρακτηριστικών αυξάνει τα kernel_RBF():if και kernel_RBF():else παίρνουν μεγαλύτερο ποσοστό του συνολικού χρόνου εκτέλεσης του predict.

Επομένως, για την επιτάχυνση των μεθόδων fit() και predict() του scikit-learn Gaussian process regression, θα μπορούσαμε να υλοποιήσουμε τις παρακάτω συναρτήσεις με το μοντέλο ροής δεδομένων της Maxeler.

- scipy.linalg.cholesky(K, lower=True)
  η οποία δέχεται έναν $n \times n$ πίνακα ως είσοδο.
- scipy.linalg.cho_solve(L, B)
  όπου ο L έχει μέγεθος $n \times n$ και ο B μέγεθος $n \times n_*$.
- numpy.dot(A, B)
  με το A να είναι ένας $n_* \times n$ πίνακας και τον B ένας $n \times n_*$ πίνακας.
- scipy.spatial.distance.pdist(A, metric='sqeuclidean')
  np.exp(b)
  scipy.spatial.distance.squareform(c)
  numpy.fill_diagonal(E)
  όπου το A έχει μέγεθος $n \times D$ ή $n_* \times D$ και τα διανύσματα b, c έχουν μέγεθος $n(n+1)/2$ ή $n_*(n_*+1)/2$, ενώ το E είναι ένας $n \times n$ ή $n_* \times n_*$ πίνακας.
- scipy.spatial.distance.cdist(A, B, metric='sqeuclidean')
  numpy.exp(C)
  με το A και το B να είναι $n_* \times D$ και $n \times D$ πίνακες και το C ένας πίνακας μεγέθους $n_* \times n$.

Κεφάλαιο 5

Gaussian Process Classification

5.1 Αλγόριθμος δυαδικού Gaussian process classification

Το πρώτο βήμα είναι η εύρεση του posterior mode $\hat{\mathbf{f}}$, με την βοήθεια της επαναληπτικής μεθόδου του Newton.

Algorithm 5.1 Αλγόριθμος εύρεσης $\hat{\mathbf{f}}$

1: **procedure** GPC_mode-finding(K (covariance matrix), $\mathbf{y}$ (± 1 targets), $p(\mathbf{y}|\mathbf{f})$ (likelihood function))
2: $\mathbf{f} \leftarrow \mathbf{0}$ — Αρχικοποίηση
3: **repeat** — Μέθοδος Newton
4: $W \leftarrow -\nabla\nabla \log p(\mathbf{y}|\mathbf{f})$
5: $L \leftarrow \text{cholesky}(I + W^{\frac{1}{2}} K W^{\frac{1}{2}})$
6: $\mathbf{b} \leftarrow W\mathbf{f} + \nabla \log p(\mathbf{y}|\mathbf{f})$
7: $\mathbf{a} \leftarrow \mathbf{b} - W^{\frac{1}{2}}((W^{\frac{1}{2}} K\mathbf{b})/L)/L^{\top}$
8: $\mathbf{f} \leftarrow K\mathbf{a}$
9: **until** convergence
10: $\log q(\mathbf{y}|X,\theta) \leftarrow -\frac{1}{2}\mathbf{a}^{\top}\mathbf{f} + \log p(\mathbf{y}|\mathbf{f}) - \sum_i \log L_{ii}$ **return** $\hat{\mathbf{f}} \leftarrow \mathbf{f}$ (posterior mode), $\log q(\mathbf{y}|X,\theta)$ (approx. log marg. likelihood)

Αλγόριθμος 5.1: Αλγόριθμος εύρεσης $\hat{\mathbf{f}}$. Η υπολογιστική πολυπλοκότητα προέρχεται από τη Cholesky παραγοντοποίηση στην γραμμή 5, η οποία χρειάζεται $n^3/6$ πράξεις (για κάθε επανάληψη της μεθόδου Newton), ενώ όλες οι άλλες πράξεις έχουν το πολύ τερταγωνική πολυπλοκότητα σε σχέση με το n.

Εφόσον έχουμε υπολογίσει το $\hat{\mathbf{f}}$ μπορούμε να κάνουμε νέες προβλέψεις.

Algorithm 5.2 Προβλέψεις για το διαδικό Gaussian process classification

1: **procedure** GPC_predictions($\hat{\mathbf{f}}$ (post. mode), X (inputs), $\mathbf{y}$ (± 1 targets), k (covariance function), $p(\mathbf{y}|\mathbf{f})$ (likelihood function), $\mathbf{x}_*$ (test input))
2: $\quad W \leftarrow -\nabla\nabla \log p(\mathbf{y}|\hat{\mathbf{f}})$
3: $\quad L \leftarrow \text{cholesky}(I + W^{\frac{1}{2}} K W^{\frac{1}{2}})$
4: $\quad \bar{f}_* \leftarrow \mathbf{k}(\mathbf{x}_*)^\top \nabla \log p(\mathbf{y}|\hat{\mathbf{f}})$
5: $\quad \mathbf{v} \leftarrow (W^{\frac{1}{2}} \mathbf{k}(\mathbf{x}_*))/L$
6: $\quad \mathbb{V}[f_*] \leftarrow k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{v}^\top \mathbf{v}$
7: $\quad \bar{\pi}_* \leftarrow \int \sigma(z) \mathcal{N}(z|\bar{f}_*, \mathbb{V}[f_*]) dz$ **return** $\bar{\pi}_*$ (predictive class probability (for class 1))

Αλγόριθμος 5.2: Προβλέψεις για το διαδικό Gaussian process classification. Το posterior mode $\hat{\mathbf{f}}$ (το οποίο μπορεί να υπολογιστεί χρησιμοποιώντας τον Αλγόριθμο 5.1) αποτελεί είσοδο του αλγορίθμου. Για πολλά δείγματα πρός πρόβλεψη οι γραμμές 4 - 7 επαναλαμβάνονται. Η συνολική πολυπλοκότητα είναι $n^3/6$ πράξεις (γραμμή 3) και n^2 πράξεις για κάθε δείγμα προς πρόβλεψη (γραμμή 5).

5.2 scikit-learn υλοποίηση του Gaussian process classification

Στη βιβλιοθήκη scikit-learn η Gaussian process ταξινόμηση είναι μια κλάση Python που υλοποιεί τις μεθόδους fit() και predict(). Η κλάση αυτή ονομάζεται GaussianProcessClassifier() και η συνάρτηση kernel που επιλέγουμε να χρησιμοποιήσουμε είναι η RBF. Παρακάτω παρουσιάζουμε ένα παράδειγμα, όπου αρχικά καλούμε τον κατασκευαστή της κλάσης, μετά καλούμε την fit μέθοδο να την εκπαίδευση του μοντέλου και τέλος κάνουμε προβλέψεις για τις κλάσεις που ανήκουν τα δείγματα εισόδου προς πρόβλεψη.

```
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF

fixed_kernel = RBF(length_scale=1.0, length_scale_bounds="fixed")
clf = GaussianProcessClassifier(kernel=fixed_kernel)

X_train, y_train, X_test = initialize()

clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
```

Η υλοποίηση αυτή βασίζεται στους αλγορίθμους 5.1 και 5.2. Η προσέγγιση Laplace χρησιμοποιείται για την προσέγγιση του μη Γκαουσιανού posterior με Γκαουσιανό. Οι αλγόριθμοι των μεθόδων fit και predict της Python κλασης GaussianProcessClassifier() δίνονται στη συνέχεια.

Algorithm 5.3 scikit-learn GaussianProcessClassifier().fit() μέθοδος

```
1:  procedure fit(X(inputs), y(targets))
2:      X, y ← check_X_y(X, y, ...)
3:      n_classes ← get_number_of_classes(y)
4:      if n_classes = 2 then
5:          self.estimator[1] ← _BinaryGaussianProcessClassifierLaplace()
6:          self.estimator[1].fit(X, y)
7:      else if n_classes > 2 then
8:          if one-vs-rest then
9:              for i ← 1, n_classes do
10:                 self.estimator[i] ← _BinaryGaussianProcessClassifierLaplace()
11:                 self.estimator[i].fit(X, y_all)
12:         else if one-vs-one then
13:             for i ← 1, n_classes(n_classes − 1)/2 do
14:                 self.estimator[i] ← _BinaryGaussianProcessClassifierLaplace()
15:                 self.estimator[i].fit(X_i, y_i)
```

Αλγόριθμος 5.3: Μας ενδιαφέρει κυρίως η στρατηγική one-vs-rest. Στην περίτωση αυτή $n_{classes}$ δυαδικοί ταξινομητές εκπαιδεύονται. Αν κάθε δυαδικός ταξινομητής χρειάζεται B_{fit} πράξεις για την μέθοδο fit(),η συνολική πολυπλοκότητα της μεθόδου fit του ταξινομητή είναι $n_{classes} \cdot B_{fit} = \mathcal{O}(n_{classes} \cdot (n^3 \cdot \bar{\text{reps}} + n^2 D))$. Η υλοποίηση του scikit-learn μας επιτρέπει να τρέξουμε n_{jobs} μεθόδους fit δυαδικών ταξινομητών παράλληλα, μειώνοντας τον συνολικό χρόνο εκτέλεσης. Παρ'ολα αυτά επιλέγουμε $n_{jobs} = 1$.

Algorithm 5.4 scikit-learn GaussianProcessClassifier().predict() method

```
1:  procedure predict(X_*(test inputs))
2:      X_* = check_array(X_*)                         check the test inputs matrix
3:      if n_classes = 2 then
4:          y_* ← self.estimator[1].predict(X_*)
5:      else if n_classes > 2 then
6:          if one-vs-rest then
7:              for i ← 1, n_classes do
8:                  y_proba[i] ← self.estimator[i].predict_proba(X_*)
9:              y_* ← decide(y_proba[])        combine the binary estimators predictions
10:
11:         else if one-vs-one then
12:             for i ← 1, n_classes(n_classes − 1)/2 do
13:                 y_proba[i] ← self.estimator[i].predict_proba(X_*)
14:             y_* ← decide(y_proba[])        combine the binary estimators predictions
15:
        return y_*(predictions)
```

Αλγόριθμος 5.4: Στην περίπτωση της στρατηγικής one-vs-rest, καλούνται $n_{classes}$ μέθοδοι predict_proba δυαδικών ταξινομητών. Αν κάθε δυαδικός ταξινομητής χρειάζεται $B_{predict_proba}$ πράξεις κατά την εκτέλεση της μεθόδου του predict_proba(), η συνολική πολυπλοκότητητα της μεθόδου predict της ταξινόμησης είναι $n_{classes} \cdot B_{predict_proba} = \mathcal{O}(n_{classes} \cdot n^2 \cdot n_*)$.

Ο δυαδικός ταξινομητής που χρησιμοποιείται παραπάνω είναι η Python κλάση _BinaryGaussianProcessClassifierLaplace() που υλοποιεί και αυτή τις μεθόδους fit και predict αλλά και τις μεθόδους _posterior_mode(), log_marginal_likelihood() και predict_proba().

Algorithm 5.5 scikit-learn _BinaryGaussianProcessClassifierLaplace().fit() μέθοδος

1: **procedure** fit(X(inputs), **y**(targets))
2: self.log_marginal_likelihood_value_ $\leftarrow$ self.log_marginal_likelihood()
3: $K \leftarrow$ self.kernel_(X)
4: lml, self.**pi**_, self.W_sr_ , self.L_ , **b**, **a** $\leftarrow$ self._posterior_mode(K)

Αλγόριθμος 5.5: Εκπαίδευση του δυαδικού ταξινομητή. Αυτή η μέθοδος καλεί τις μεθόδους log_marginal_likelihood() και _posterior_mode(). Η συνάρτηση kernel που χρησιμοποιείται στην γραμμή 3 είναι η RBF, και είναι ίδια με την Gaussian process regression συνάρτηση kernel που δίνεται από τον Αλγόριθμο 4.4. Η συνολική χρονική πολυπλοκότητα είναι $B_{fit} = \mathcal{O}(n^3 \cdot \bar{\text{reps}} + n^2 D)$.

Algorithm 5.6 scikit-learn _BinaryGaussianProcessClassifierLaplace() ._posterior_mode() μέθοδος

1: **procedure** _posterior_mode(K)
2: **f** $\leftarrow$ **0**
3: log_marginal_likelihood $\leftarrow -\infty$
4: **for** reps $\leftarrow 1$,max_reps **do**
5: **pi** $\leftarrow \mathbf{1}/(\mathbf{1} + \exp(-\mathbf{f}))$
6: W $\leftarrow$ **pi** $\cdot$ (**1** $-$ **pi**)
7: W_sr $\leftarrow$ sqrt(W)
8: B $\leftarrow I +$ W_sr $\cdot K \cdot$ W_sr
9: L $\leftarrow$ cholesky(B)
10: **b** $\leftarrow$ W$\cdot$**f** + (**y** $-$ **pi**)
11: **a** $\leftarrow$ **b**$-$W_sr $\cdot$ cho_solve(L, W_sr$\cdot K \cdot$ **b**)
12: **f** $\leftarrow K \cdot$ **a**
13: lml $\leftarrow -\frac{1}{2} \cdot \mathbf{a}^\top \cdot \mathbf{f} - \sum_i \log(1 + \exp(-(y_i \cdot 2 - 1) \cdot f_i)) - \sum_j \log(L_{jj})$
14: **if** lml $-$ log_marginal_likelihood $<$ 1e-10 **then**
15: break
16: log_marginal_likelihood $\leftarrow$ lml
17: **return** log_marginal_likelihood, **pi**, W_sr, L, **b**, **a**

Αλγόριθμος 5.6: Εύρεση του posterior mode για τον δυαδικό ταξινομητή με RBF συνάρτηση kernel. Αυτή η υλοποίηση της βιβλιοθήκης scikit-learn βασίζεται στον Αλγόριθμο 5.1 και χρησιμοποεί τις Python συναρτήσεις scipy.linalg.cholesky() και scipy.linalg.cho_solve(). Η συνολική πολυπλοκότητητα της μεθόδου είναι $\mathcal{O}(n^3 \cdot \bar{\text{reps}})$.

Algorithm 5.7 scikit-learn _BinaryGaussianProcessClassifierLaplace()
.log_marginal_likelihood() μέθοδος

1: **procedure** log_marginal_likelihood
2: $K \leftarrow$ self.kernel_(X)
3: lml, **pi**, W_sr, L, **b**, **a** ← self._posterior_mode(K)
4: **return** lml

Αλγόριθμος 5.7: Επιστρέφει την τιμή της log-marginal likelihood για τα δεδομένα εκπαίδευσης, η οποία υπολογίζεται στην Python μέθοδο _posterior_mode(). Η συνολική πολυπλοκότητα της μεθόδου είναι $\mathcal{O}(n^3 \cdot \text{r}\bar{\text{e}}\text{ps} + n^2 D)$.

Algorithm 5.8 scikit-learn _BinaryGaussianProcessClassifierLaplace().predict() method

1: **procedure** predict(X_*(test inputs))
2: $K_* \leftarrow$ self.kernel_(X, X_*) — K_* corresponding to Algorithm 5.2 Line 4 $k(\mathbf{x}_*)$
3: $\bar{\mathbf{f}}_* \leftarrow K_*^\top \cdot$ (**y** − self.**pi**_) — Algorithm 5.2 Line 4
4: **for** i ← 1 , n_* **do**
5: **if** $\bar{f}_{*i} > 0$ **then**
6: $y_{*i} = +1$ — belongs to first class prediction
7: **else**
8: $y_{*i} = -1$ — belongs to second class prediction
9: **return** $\mathbf{y}_*$

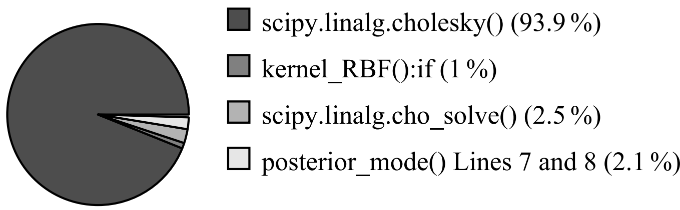
Αλγόριθμος 5.8: Η μέθοδος _BinaryGaussianProcessClassifierLaplace().predict() καλείται από το GaussianProcessClassifier().predict() μόνο στην περίπτωση δύο κλάσεων ταξινόμησης, όταν παίρνουμε απευθείας το αποτέλεσμα του δυαδικού ταξινομητή και δεν χρειάζεται να ακολουθήσουμε μια από τις one-versus-all ή one-versus-one στρατηγικές. Η πολυπλοκότητα είναι $\mathcal{O}(n \cdot n_* \cdot D)$.

Algorithm 5.9 scikit-learn _BinaryGaussianProcessClassifierLaplace().predict_proba() μέθοδος

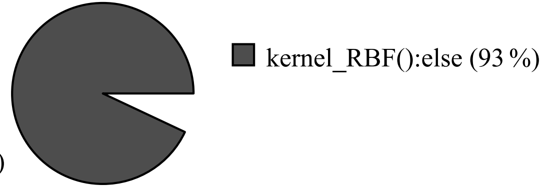
1: **procedure** predict_proba(X_*(test inputs))
2: $K_* \leftarrow$ self.kernel_(X, X_*) — K_* corresponding to Algorithm 5.2 Line 4 $k(\mathbf{x}_*)$
3: $\bar{\mathbf{f}}_* \leftarrow K_*^\top \cdot$ (**y** − self.**pi**_) — Algorithm 5.2 Line 4
4: **v** ← solve(self.L_, self.W_sr_ $\cdot K_*$) — Al. 5.2 Line 5
5: $\mathbb{V}[\mathbf{f}_*] \leftarrow$ self.kernel_.diag(X_*) - diag.($\mathbf{v}^\top \cdot \mathbf{v}$) — Al. 5.2 Line 6
6: $\bar{\pi}_* \leftarrow$ integral_approximation() — Al. 5.2 Line 7
7: **return** $\bar{\pi}_*$

Αλγόριθμος 5.9: Η μέθοδος αυτή επιστρέφει τις πιθανότητες της πρόβλεψης για τον πίνακα προς πρόβλεψη X_*. Η υλοποίηση αυτή της βιβλιοθήκης scikit-learn βασίζεται στον Αλγόριθμο 5.2. Στη γραμμή 4 καλείται η συνάρτηση scipy.linalg.solve(), η οποία απαιτεί $\mathcal{O}(n^2 \cdot n_*)$ πράξεις. Επομένως, η συνολική πολυπλοκότητα είναι $B_{predict_proba} = \mathcal{O}(n^2 \cdot n_*)$.

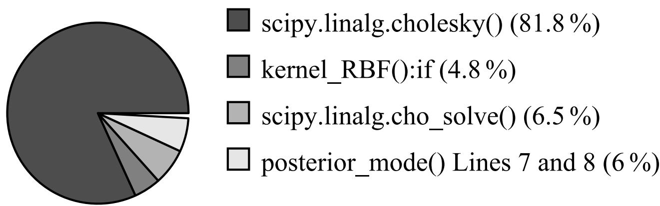
5.3 Ανάλυση scikit-learn Gaussian process classification



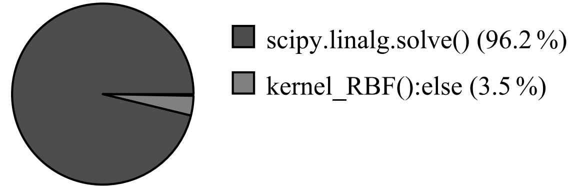
fit()
Skin Segmentation Dataset $D = 3$, $n_{classes} = 2$, $n = 2000$
Συνολικός χρόνος: 28.06 sec



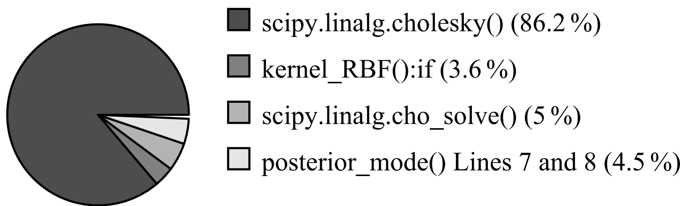
predict()
Skin Segmentation Dataset $D = 3$, $n_{classes} = 2$, $n = 2000$, $n_* = 2000$
Συνολικός χρόνος: 0.13 sec



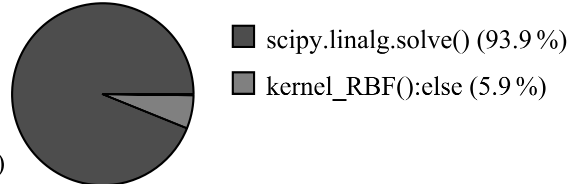
fit()
Poker Hand Dataset $D = 10$, $n_{classes} = 10$, $n = 2000$
Συνολικός χρόνος: 63.58 sec



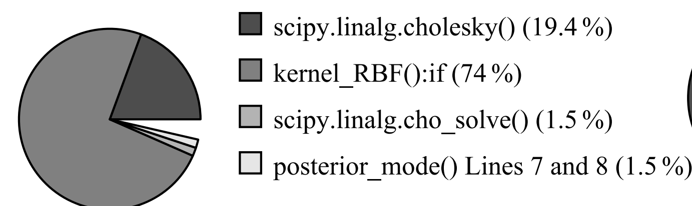
predict()
Poker Hand Dataset $D = 10$, $n_{classes} = 10$, $n = 2000$, $n_* = 2000$
Συνολικός χρόνος: 39.65 sec



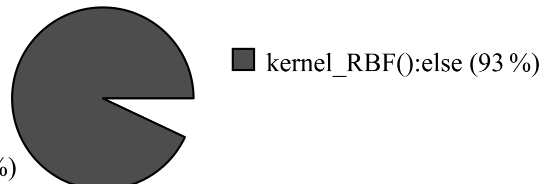
fit()
Sensorless Drive Diagnosis Dataset $D = 49$, $n_{classes} = 11$, $n = 2000$
Συνολικός χρόνος: 145.59 sec



predict()
Sensorless Drive Diagnosis Dataset $D = 49$, $n_{classes} = 11$, $n = 2000$, $n_* = 2000$
Συνολικός χρόνος: 58.84 sec



fit()
Gisette Dataset $D = 5000$, $n_{classes} = 2$, $n = 2000$
Συνολικός χρόνος: 29.27 sec



predict()
Gisette Dataset $D = 5000$, $n_{classes} = 2$, $n = 2000$, $n_* = 2000$
Συνολικός χρόνος: 22.9 sec

Με τον ίδιο τρόπο με την μέθοδο μηχανικής μάθησης Gaussian process regression, στοχεύουμε να εντοπίσουμε τις πιο χρονοβόρες συναρτήσεις στην υλοποίηση της βιβλιοθήκης scikit-learn για το Gaussian process classification. Διάφοροι μετρητές χρόνου τοποθετήθηκαν σε σημεία του κώδικα των μεθόδων της Python κλάσης _BinaryGaussianProcessClassifierLaplace(), εφόσον οι μέθοδοι fit και predict καλούν τον δυαδικό ταξινομητή. Τα διαγράμματα τη προηγούμενης σελίδας παρουσιάζουν το ποσοστού του χρόνου που καταναλώνεται σε κάθε συνάρτηση. Οι μέθοδοι fit και predict καλούνται για τα σύνολα δεδομένων Skin Segmentation, Poker Hand, Sensorless Drive Diagnosis και Gisette του UCI repository, τα οποία έχουν διαφορετικά μεγέθη χαρακτηριστικών, και μεγάλα μεγέθη εισόδων για εκπαίδευση και πρόβλεψη ($n = 2000, n_* = 2000$).

Σημειώνουμε ότι το kernel_RBF():if αποτελείται από το scipy.spatial.distance.pdist(), το numpy.exp(), το scipy.spatial.distance.squareform() και το numpy.fill_diagonal(). Επίσης, το kernel_RBF():if αποτελείται από τα scipy.spatial.distance.cdist() and numpy.exp().

Όπως φαίνεται από τα διαγράμματα, όπως και στο πρόβλημα της παλινδρόμησης, η συνάρτηση scipy.linalg.cholesky() καταλαμβάνει το μεγαλύτερο ποσοστό χρόνου. Η Cholesky παραγοντοποίηση υπολογίζεται για κάθε επανάληψη της επαναληπτικής μεθόδου του Newton για τον υπολογισμό του posterior mode. Από την άλλη ο χρόνος εκτέλεσης του kernel_RBF():if αυξάνεται για datasets με μεγάλο αριθμό χαρακτηριστικών. Επίσης στην περίπτωση ταξινόμησης με πολλές κλάσεις η συνάρτηση scipy.linalg.solve() είναι η πιο χρονοβόρα, ενώ στην περίπτωση της δυαδικής ταξινόμησης το kernel_RBF():else κατέχει το μεγαλύτερο ποσοστό στο χρόνο εκτέλεσης της μεθόδου predict.

Επομένως για την επιτάχυνση του Gaussian process classification της βιβλιοθήκης scikit-learn θα πρέπει να λάβουμε υπόψιν τις παρακάτω συναρτησεις:

- scipy.linalg.cholesky(K, lower=True)
  η οποία παίρνει έναν $n \times n$ πίνακα ως είσοδο.
- scipy.linalg.solve(L, B)
  όπου το L είναι ένας κάτω τριγωνικός πίνακας μεγέθους $n \times n$ και ο πίνακας B είναι μεγέθους $n \times n_*$.
- scipy.spatial.distance.pdist(A, metric='sqeuclidean')
  np.exp(b)
  scipy.spatial.distance.squareform(c)
  numpy.fill_diagonal(E)
  όπου ο A είναι μεγέθους $n \times D$ και τα διανύσματα b, c έχουν μήκος $n(n+1)/2$, ενώ το E είναι ένας $n \times n$ πίνακας.
- scipy.spatial.distance.cdist(A, B, metric='sqeuclidean')
  numpy.exp(C)
  με το A και το B να είναι $n_* \times D$ και $n \times D$ πίνακες και το C ένας πίνακας μεγέθους $n_* \times n$.

Κεφάλαιο 6

Kernel Ridge Regression

6.1 Αλγόριθμος Kernel ridge regression

Algorithm 6.1 Αλγόριθμος Kernel ridge regression

1: **procedure** KRR(X(inputs), $\mathbf{y}$(targets), k(kernel function), λ, $\mathbf{x}_*$(test input))
2: $\quad K_{ij} \leftarrow k(\mathbf{x}_i, \mathbf{x}_j)$
3: $\quad \alpha \leftarrow \mathbf{y}/(K + \lambda I)$
4: $\quad \mathbf{k}_{*i} \leftarrow k(\mathbf{x}_*, \mathbf{x}_i)$
5: $\quad f(\mathbf{x}_*) \leftarrow \mathbf{k}_*^\top \alpha$
6: **return** $f(\mathbf{x}_*)$ (prediction)

Αλγόριθμος 6.1: Η συνολική πολυπλοκότητα του αλγορίθμου είναι $\mathcal{O}(n^3)+\mathcal{O}(n^2D)$, εφόσον ο υπολογισμός του α χρειάζεται $\mathcal{O}(n^3)$ πράξεις και ο υπολογισμός του K χρειάζεται $\mathcal{O}(n^2D)$ πραξεις.

6.2 scikit-learn υλοποίηση του kernel ridge regression

Στην Python υλοποίηση του scikit-learn ο αλγόριθμος για το kernel ridge regression υλοποιείται από την κλάση KernelRidge(), η οποία διαθέτει τις μεθόδους fit($X, \mathbf{y}$) και predict(X_*).

Για να χρησιμοποιήσουμε την εφαρμογή πρώτα αρχικοποιούμε το μοντέλο με τον κατασκευαστή της κλάσης και στη συνέχεια καλούμε τη μέθοδο fit για την εκπαίδευση και την predict για να πάρουμε τις προβλέψεις για νέα δεδομένα. Παρακάτω ακολουθεί ένα παράδειγμα:

```
from sklearn.kernel_ridge import KernelRidge
X_train, y_train, X_test = initialize()
clf = KernelRidge()
clf.fit(X_train, y_train)
y_test = clf.predict(X_test)
```

Ακολούθως παρουσιάζουμε τις μεθόδους fit() και predict() της κλάσης KernelRidge(), οι οποίες βασίζονται στον Αλγόριθμο 6.1.

Algorithm 6.2 scikit-learn KernelRidge().fit() μέθοδος

1: **procedure** fit(X(inputs), **y**(targets))
2: $X, \mathbf{y} \leftarrow$ check_X_y($X, \mathbf{y}$, ...)
3: $K \leftarrow$ self._get_kernel(X)
4: self.dual_coef_ $\leftarrow$ _solve_cholesky_kernel(K, **y**, ...)

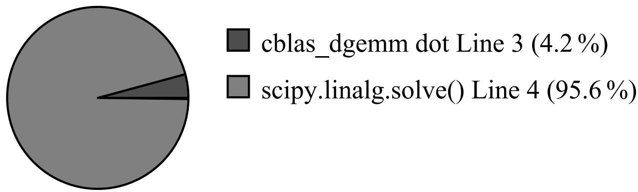
Αλγόριθμος 6.2: Η προκαθορισμένη συνάρτηση kernel είναι η $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\top \cdot \mathbf{x}'$ (γινόμενο), έτσι ώστε ο πίνακας K να είναι το γινόμενο των πινάκων X και $X^\top$ (γραμμή 3). Το γινόμενο αυτό υπολογίζεται με την χρήση της συνάρτησης cblas_dgemm dot. Η συνάρτηση _solve_-cholesky_kernel() στη γραμμή 4 καλεί το scipy.linalg.solve(), και εφόσον το K είναι συμμετρικός και θετικά ορισμένος πίνακας, το scipy.linalg.solve() υπολογίζει την Cholesky παραγοντοποίηση του K και στη συνέχεια λύνει το σύστημα $K\cdot$ (self.dual_coef_) $=$ **y**. Η συνολική χρονική πολυπλοκότητα είναι $\mathcal{O}(n^3) + \mathcal{O}(n^2 D)$, αφού ο πολλαπλασιασμός πινάκων στη γραμμή 3 απαιτεί $\mathcal{O}(n^2 D)$ πράξεις, και ο υπολογισμός του self.dual_coef_ στη γραμμή 4 χρειάζεται $\mathcal{O}(n^3)$ πράξεις.

Algorithm 6.3 scikit-learn KernelRidge().predict() μέθοδος

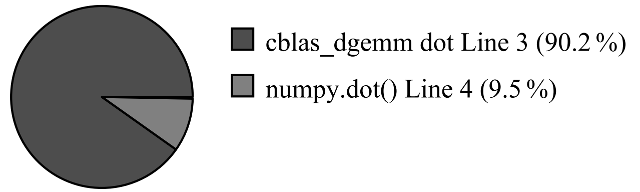
1: **procedure** predict(X_*(test inputs))
2: check_is_fitted(self,...)
3: $K_* \leftarrow$ self._get_kernel(X_*, X)
4: **return** $K_*\cdot$(self.dual_coef_)

Αλγόριθμος 6.3: Το γινόμενο στη γραμμή 4 υπολογίζεται με τη βοήθεια της συνάρτησης numpy.dot(), ενώ το γινόμενο στη γραμμή 3 με την χρήση της cblas_dgemm βιβλιοθήκης. Η συνολική πολυπλοκότητα είναι $\mathcal{O}(n \cdot n_* D) + \mathcal{O}(n \cdot n_*)$, όπου n_* είναι ο αριθμός των εισόδων για πρόβλεψη. Ο πρώτος όρος στην πολυπλοκότητα αντιστοιχεί στον αριθμό των πράξεων στη γραμμή 3 και ο δεύτερος όρος στις πράξεις της γραμμής 4.

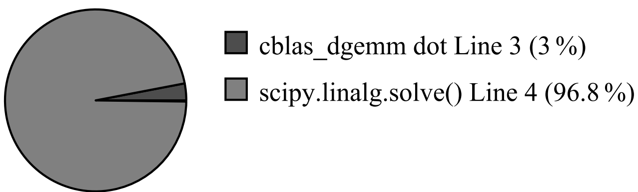
6.3 Ανάλυση scikit-learn kernel ridge regression



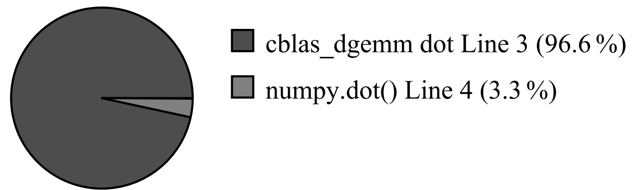
fit()
CASP Dataset $D = 9$, $n = 10000$
Συνολικός χρόνος: 70.44 sec



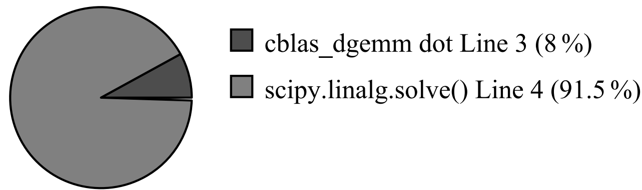
predict()
CASP Dataset $D = 9$, $n = 10000$, $n_* = 10000$
Συνολικός χρόνος: 0.94 sec



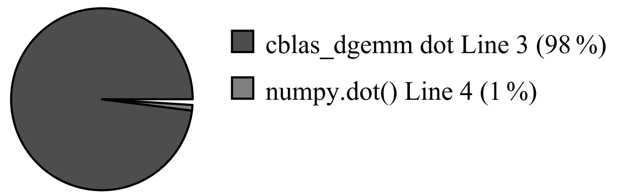
fit()
Online News Popularity Dataset $D = 60$, $n = 10000$
Συνολικός χρόνος: 74.35 sec



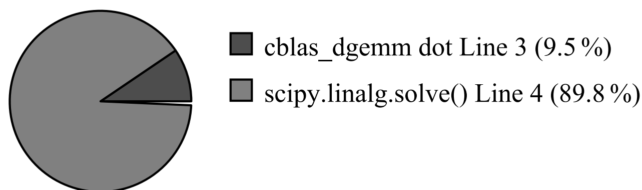
predict()
Online News Popularity Dataset $D = 60$, $n = 10000$, $n_* = 10000$
Συνολικός χρόνος: 2.7 sec



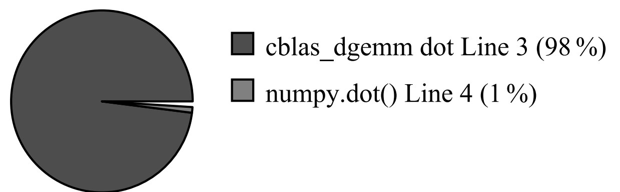
fit()
BlogFeedback Dataset $D = 280$, $n = 10000$
Συνολικός χρόνος: 77.8 sec



predict()
BlogFeedback Dataset $D = 280$, $n = 10000$, $n_* = 10000$
Συνολικός χρόνος: 11.64 sec



fit()
slices localization Dataset $D = 386$, $n = 10000$
Συνολικός χρόνος: 85.41 sec



predict()
slices localization Dataset $D = 386$, $n = 10000$, $n_* = 10000$
Συνολικός χρόνος: 17.49 sec

Μια απλή ανάλυση που παρέχει την κατανομή του χρόνου εκτέλεσης στα διάφορα κομμάτια της εφαρμογής, μπορούμε να την πάρουμε χρησιμοποιώντας μετρητές χρόνου. Τα διαγράμματα που παρουσιάστηκαν προηγουμένως, προέκυψαν από την τοποθέτηση μετρητών χρόνου σε διαφορετικά σημεία του Python κώδικα που αντιστοιχεί στους Αλγορίθμους 6.2 και 6.3. Οι μέθοδοι fit και predict εκτελέστηκαν για τέσσερα διαφορετικά σύνολα δεδομένων με διαφορετικό αριθμό χαρακτηριστικών και μεγάλα μεγέθη εισόδων για εκπαίδευση και πρόβλεψη ($n = 10000, n_* = 10000$).

Όπως μπορούμε να παρατηρήσουμε από τα διαγράμματα, η συνάρτηση scipy.linalg.solve() είναι η πιο απαιτητική στην μέθοδο fit και θα μπορούσε να επιταχυνθεί χρησιμοποιώντας το μοντέλο ροής δεδομένων της Maxeler. Επιπλέον, η συνάρτηση cblas_dgemm dot γίνεται όλο και πιο απαιτητική χρονικά καθώς το μέγεθος των χαρακτηριστικών αυξάνει.

Επομένως για την επιτάχυνση των μεθόδων fit και predict του kernel ridge regression της βιβλιοθήκης scikit-learn, θα μπορούσαμε να επιταχύνουμε τις παρακάτω συναρτήσεις:

- scipy.linalg.solve(K, **y**, sym_pos=True, overwrite_a=False),
  όπου το K είναι ένας συμμετρικός θετικά ορισμένος πίνακας μεγέθους $n \times n$ και το **y** ένα διάνυσμα μήκους n.

- cblas_dgemm dot,
  η οποία υπολογίζει το γινόμενο δύο πινάκων με μεγέθη $n \times D$ και $D \times n$ στη μέθοδο fit(), και το γινόμενο δύο πινάκων με μεγέθη $n_* \times D$ και $D \times n$ στη μέθοδο predict.

Κεφάλαιο 7

Υπολογιστικό μοντέλο ροής δεδομένων της Maxeler

7.1 Το υπολογιστικό μοντέλο ροής δεδομένων

Σε μια συμβατική εφαρμογή, ο κώδικας του προγράμματος μετατρέπεται σε μια σειρά από εντολές για κάποιον συγκεκριμένο επεξεργαστή και φορτώνεται στη μνήμη, όπως φαίνεται στο σχήμα 7.1. Οι εντολές αυτές περνούν από τον επεξεργαστή και περιστασιακά γράφονται και διαβάζονται δεδομένα από τη μνήμη. Αυτό το μοντέλο υπολογισμού είναι εκ φύσεως σειριακό και η απόδοση εξαρτάται σε πολύ μεγάλο βαθμό από την ταχύτητα πρόσβασης στη μνήμη και την ταχύτητα του ρολογιού του επεξεργαστή.

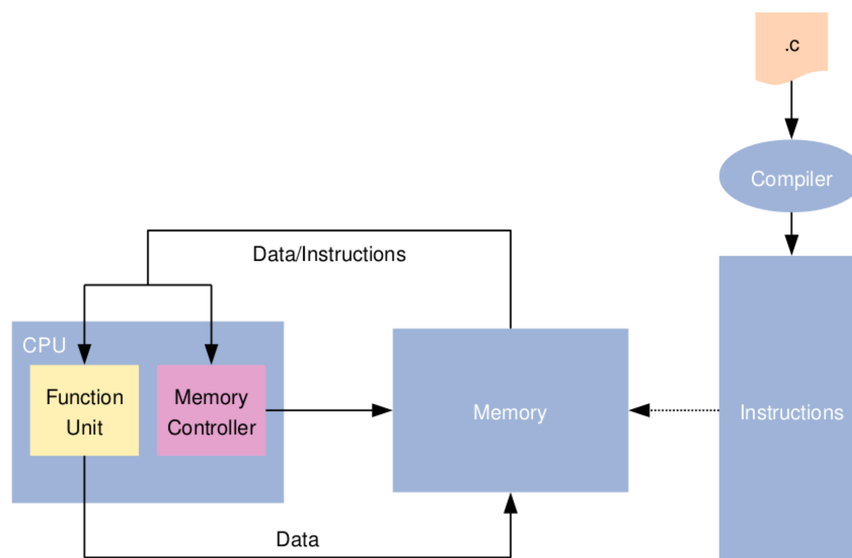


Figure 7.1: Control flow model

Από την άλλη, σε ένα πρόγραμμα ροής δεδομένων, περιγράφουμε της πράξεις και τον τρόπο που πρέπει να μετακινηθούν τα δεδομένα για την υλοποίηση ενός συγκεκριμένου αλγορίθμου. Σε μια μηχανή ροής δεδομένων (DFE), τα δεδομένα κινούνται από την μνήμη στο κύκλωμα επεξεργασίας, όπου μεταβαίνουν από τη μία αριθμητική μονάδα (dataflow core) στην επόμενη μέχρι να ολοκληρωθεί ο υπολογισμός. Κάθε αριθμητική μονάδα υπολογίζει μόνο μια συγκεκριμένη πράξη (για παράδειγμα μια πρόσθεση ή έναν πολλαπλασιασμό) και είναι αρκετά απλή, επομένως χιλιάδες τέτοιες μονάδες μπορούν να χωρέσουν σε ενα DFE. Σε μια σωλήνωση (pipeline) του

DFE κάθε αριθμητική μονάδα εκτελεί υπολογισμούς παράλληλα με τις άλλες, σε δεδομένα γειτονικά στη ακολουθία δεδομένων που διέρχεται από το DFE. Αντίθετα με τους επεξεργαστές όπου οι πράξεις υπολογίζονται στις ίδιες μονάδες επεξεργάσίας σε διαφορετικές χρονικές στιγμές (”υπολογισμός στον χρόνο”), ένας υπολογισμός ροής δεδομένων συμβαίνει σε διάφορα σημεία του DFE (”υπολογισμός στον χώρο”).

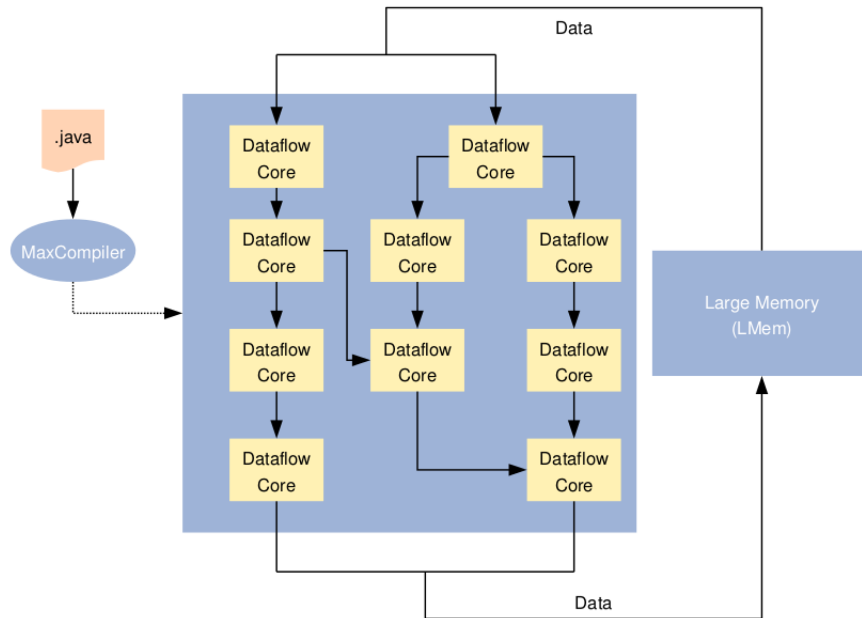


Figure 7.2: Dataflow model

Η δομή του DFE για κάθε διαφορετικό αλγόριθμο αναπαριστά τον υπολογισμό και επομένως δεν χρειάζονται εντολές, οι οποίες έχουν πλέον αντικατασταθεί από αριθμητικές μονάδες στο chip, που έχουν συνδεθεί έτσι ώστε να επιτελούν μια συγκεκριμένη επεξεργασία. Εφόσον δεν υπάρχουν εντολές, δεν υπάρχει και ανάγκη για κυκλώματα απόπλεξης των εντολών, κρυφής μνήμης εντολών, μηχανισμών πρόβλεψης άλματος και δυναμικής δρομολόγησης εντολών εκτός σειράς. ’Ετσι όλοι οι πόροι του μηχανήματος αξιοποιούνται σε υπολογισμούς.

7.2 Μηχανήματα ροής δεδομένων (DFEs)

Στο σχήμα 7.3 φαίνεται η αρχιτεκτονική ενός υπολογιστικού συστήματος ροής δεδομένων της Maxeler, το οποίο αποτελείται από DFEs με τις τοπικές μνήμες τους, συνδεδεμένα με έναν επεξεργαστή (CPU). Κάθε DFE μπορεί να έχει υλοποιημένους πολλούς Kernels, οι οποίοι εκτελούν τον υπολογισμό καθώς τα δεδομένα ρέουν ανάμεσα στον επεξεργαστή, το DFE και τις μνήμες. Ένα DFE διαθέτει δύο ειδών μνήμες: Την Fmem (γρήγορη μνήμη) που μπορεί να αποθηκεύσει αρκετά megabytes πάνω στο DFE, με ταχύτητα προσπέλασης της τάξης των terabytes/second, και την Lmem (μεγάλη μνήμη) η οποία μπορεί να αποθηκεύσει πολλά gigabytes εκτός του chip.

Η ταχύτητα και η ευελιξία της Fmem είναι ο κύριος λόγος που τα DFEs μπορούν να πετύχουν πολύ υψηλή απόδοση σε περίπλοκες εφαρμογές. Οι εφαρμογές μπορούν να εκμεταλευτούν τη συνολική χωρητικότητα της Fmem, επειδή η μνήμες και ο υπολογισμός βρίσκονται στον ίδιο χώρο, επομένως τα δεδομένα μπορούν να κρατούνται στην μνήμη πολύ κοντά στα σημεία

υπολογισμού. Σε αντίθεση, οι παραδοσιακές αρχιτεκτονικές επεξεργαστών έχουν πολλαπλά επίπεδα κρυφών μνημών και μόνο η μικρότερη και γρηγορότερη μνήμη βρίσκεται κοντά στις μονάδες υπολογισμού, ενώ τα δεδομένα αντιγράφονται σε πολλά επίπεδα της ιεραρχίας μνήμης.

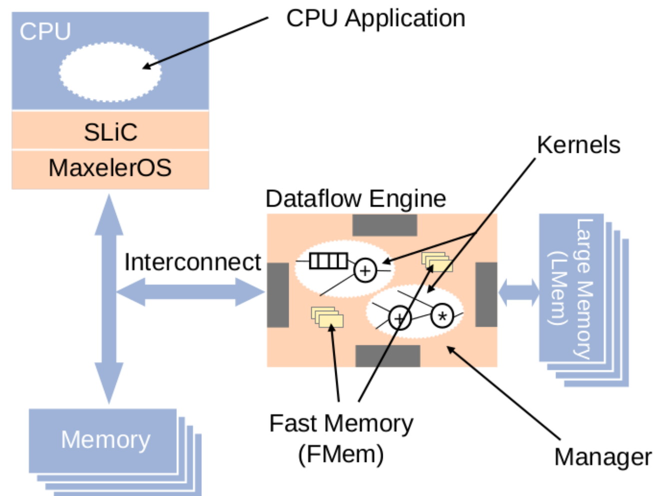


Figure 7.3: Dataflow engine architecture

Ένα DFE προγραμματίζεται με ένα ή περισσότερα Kernels και ένα Manager. Τα Kernels υλοποιούν τον υπολογισμό, ενώ το Manager οργανώνει την κίνηση των δεδομένων μέσα στο DFE. Εφόσον γραφτούν τα Kernels και ο Manager, ο MaxCompiler δημιουργεί εφαρμογές ροής δεδομένων, οι οποίες μπορούν στη συνέχεια να καλεθούν από κώδικα επεξεργαστή μέσω του SLiC interface. Το συνολικό σύστημα το διαχειρίζεται το MaxelerOS, το οποίο τρέχει πάνω σε Linux.

Κεφάλαιο 8

Επιτάχυνση του Πολλαπλασιασμού πινάκων και της Απόστασης πινάκων με το μοντέλο ροής δεδομένων

8.1 Γινόμενο πινάκων

Αν ο A είναι ένας $m \times k$ πίνακας και ο B είναι ένας $k \times n$ πίνακας

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,k} \\ a_{2,1} & a_{2,2} & \dots & a_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,k} \end{bmatrix}, B = \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,n} \\ b_{2,1} & b_{2,2} & \dots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k,1} & b_{k,2} & \dots & b_{k,n} \end{bmatrix}$$

το γινόμενο τους AB θα είναι ο $m \times n$ πίνακας

$$AB = \begin{bmatrix} ab_{1,1} & ab_{1,2} & \dots & ab_{1,n} \\ ab_{2,1} & ab_{2,2} & \dots & ab_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ ab_{m,1} & ab_{m,2} & \dots & ab_{m,n} \end{bmatrix}$$

όπου κάθε i, j στοιχείο δίνεται από το τύπο

$$ab_{i,j} = \sum_{p=1}^{k} a_{i,p} \cdot b_{p,j}$$

.
Η συνολική πολυπλοκότητα του αντίστοιχου αλγορίθμου είναι $\mathcal{O}(mkn)$.

8.2 Ευκλείδια τετραγωνική απόσταση πινάκων

Ας υποθέσουμε ότι ο A είναι ένας $m \times k$ πίνακας, ο οποίος αποτελείται από m διανύσματα μήκους k. Επιπλέον ο B είναι ένας $n \times k$ πίνακας από n διανύσματα, μήκους k.

$$A = \begin{bmatrix} \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,k} \end{pmatrix} \\ \begin{pmatrix} a_{2,1} & a_{2,2} & \dots & a_{2,k} \end{pmatrix} \\ \vdots \\ \begin{pmatrix} a_{m,1} & a_{m,2} & \dots & a_{m,k} \end{pmatrix} \end{bmatrix}, B = \begin{bmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,k} \end{pmatrix} \\ \begin{pmatrix} b_{2,1} & b_{2,2} & \dots & b_{2,k} \end{pmatrix} \\ \vdots \\ \begin{pmatrix} b_{n,1} & b_{n,2} & \dots & b_{n,k} \end{pmatrix} \end{bmatrix}.$$

Στόχος μας είναι να υπολογίσουμε την Ευκλείδια τετραγωνική απόσταση ανάμεσα σε κάθε διάνυσμα του A και του B. Ο πίνακας με τις αποστάσεις D ορίζεται ως ένας $m \times n$ πίνακας, όπου κάθε i, j στοιχείο δίνεται από τον τύπο

$$d_{i,j} = \sum_{p=1}^{k} (a_{i,p} - b_{j,p})^2$$

.
Η συνολική πολυπλοκότητα του αντίστοιχου αλγορίθμου είναι $\mathcal{O}(mkn)$

8.3 Οι συναρτήσεις των scipy και numpy που θα επιταχυνθούν

Στο πλαίσιο των προγραμμάτων μηχανικής μάθησης της βιβλιοθήκης scikit-learn, ενδιαφερόμαστε για τον υπολογισμό της Ευκλείδιας τετραγωνικής απόστασης δύο πινάκων και στη συνέχεια την εφαρμογή της συνάρτησης $g(x) = \exp(-\frac{1}{2}x)$ σε κάθε στοιχείο του πίνακα με τις αποστάσεις. Αν $A \equiv B$, τότε σε Python κώδικα αυτό γίνεται ως εξής:

```
d = scipy.spatial.distance.pdist(A, metric='sqeuclidean')
d = numpy.exp(d)
D = scipy.spatial.distance.squareform(d)
numpy.fill_diagonal(D)
```

Αν ο A και ο B είναι διαφορετικοί πίνακες:

```
D = scipy.spatial.distance.cdist(A, B, metric='sqeuclidean')
D = numpy.exp(D)
```

Επίσης για το πρόβλημα του πολλαπλασιασμού πινάκων οι συναρτήσεις numpy.dot(A,B) ή cblas_dgemm καλούνται στην Pyhton.

8.4 Υλοποίση στο μοντέλο ροής δεδομένων της Maxeler

Η πρώτη προσέγγιση για τη λήση του προβλήματος του πολλαπλασιασμού πινάκων με το μοντέλο ροής δεδομένων θα μπορούσε να είναι η δημιουργία δύο ακολουθιών από δεδομένα, μία από τον πίνακα A και μία από τον B. Στη συνέχεια θα μπορούσαμε να περάσουμε τα δεδομένα αυτά από έναν απλό Kernel με έναν πολλαπλασιαστή και ένα συσσωρευτή αθροίσματος. Για παράδειγμα η A ακολουθία δεδομένων θα μπορούσε να ξεκινάει με την πρώτη γραμμή του A και η B ακολουθία με την πρώτη στήλη του B. Κάθε ζεύζος αριθμών τότε θα πολλαπλασιαζόταν και το γινόμενο θα αθροιζόταν με το ως τότε άθροισμα. Όταν το εσωτερικό γινόμενο θα ήταν έτοιμο το αποτέλεσμα θα έφευγε ως έξοδος από το Kernel.

Παρ'ολα αυτά η λύση αυτή θα απαιτούσε mkn χτυπήματα του Kernel (Kernel ticks) και μεγάλη μεταφορά δεδομένων, εφόσον ίδιες γραμμές από τον A και ίδιες στήλες από τον B θα μεταφέρονταν πολλές φορές από τον επεξεργαστή στο DFE. Επιπλέον δεν θα υπήρχε κανένας παραλληλισμός στις πράξεις εφόσον μόνο ένας πολλαπλασισμός και μια πρόσθεση θα εκτελούνταν σε κάθε Kernel tick.

Το υπολογιστικό μοντέλο ροής δεδομένων της Maxeler μας δίνει την δυνατότητα να δημιουργήσουμε πολλές παράλληλες σωληνώσεις εκτέλεσης όπου πολλές πράξεις εκτελούνται την ίδια στιγμή. Για να εκμεταλευτούμε τον παραλληλισμό που διαθέτει το πρόβλημα πολλαπλασιασμού πινάκων θα πρέπει να κρατήσουμε τα δεδομένα στην Fmem μνήμη και να μεγιστοποιήσουμε την επαναχρησιμοποίηση των δεδομένων. Όμως για μεγάλους πίνακες αυτό θα ήταν αδύνατο και επομένως χρειαζόμαστε μια λύση όπου οι πίνακες χωρίζονται σε κομμάτια (tiles), τα οποία χωράνε στην Fmem.

8.4.1 Πολλαπλασιασμός πινάκων σε tiles

Ας υποθέσουμε ότι ο $A_{m\times k}$ και ο $B_{k\times n}$ μπορούν να χωριστούν σε τετραγωνικά tiles, όπως φαίνεται στο σχήμα 8.1, και $mt = \frac{m}{TILE_SIZE}, kt = \frac{k}{TILE_SIZE}, nt = \frac{n}{TILE_SIZE}$

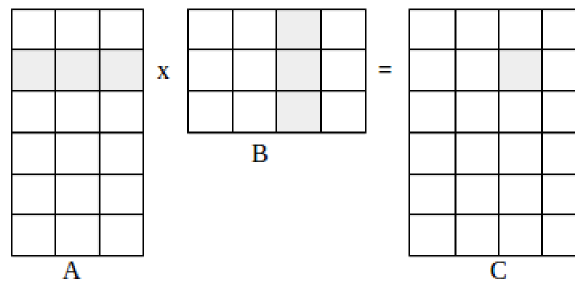


Figure 8.1: $C_{23} = A_{21}B_{13} + A_{22}B_{23} + A_{23}B_{33}$

Τότε το tile C_{ij} του τελικού πίνακα με τα εσωτερικά γινόμενα C δίνεται από:

$$C_{ij} = \sum_{p=1}^{kt} A_{jp}B_{pj} \quad i = 1, ..., mt,\ j = 1, ..., nt$$

όπου τα A_{jp} και B_{pj} είναι tiles του A και του B, και το $\sum$ σημβολίζει άθροισμα πινάκων (tiles).

Η λύση με το μοντέλο ροής δεδομένων αποτελείται από έναν Kernel που πολλαπλασιάζει δύο τετραγωνικά tiles των A και B, και έναν δεύτερο Kernel που δέχεται kt tiles το ένα μετά το άλλο και συσσωρεύει το άθροισμά τους. Ο επεξεργαστής στέλνει ζευγάρια από A, B tiles στο DFE και παίρνει πίσω C tiles. Η ίδια υλοποίηση μπορεί να χρησιμοποιηθεί για τον υπολογισμό της Ευκλείδιας τετραγωνικής απόστασης πινάκων, αν αντί για πολλαπλασιασμό δύο στοιχείων υπολογίζουμε το τετράγωνο της διαφοράς τους. Η λύση για το πρόβλημα πολλάπλασιασμού πινάκων με την χρήση των εργαλείων της Maxeler στηρίζεται στην max-power βιβλιοθήκη της Maxeler που παρέχει τον Kernel για τον πολλαπλασιασμό δύο tiles, οπότε δεν κρίνεται σκόπιμη η παρουσίαση των λεπτομεριών τη υλοποίησης.

8.5 Διάγραμμα αρχιτεκτονικής

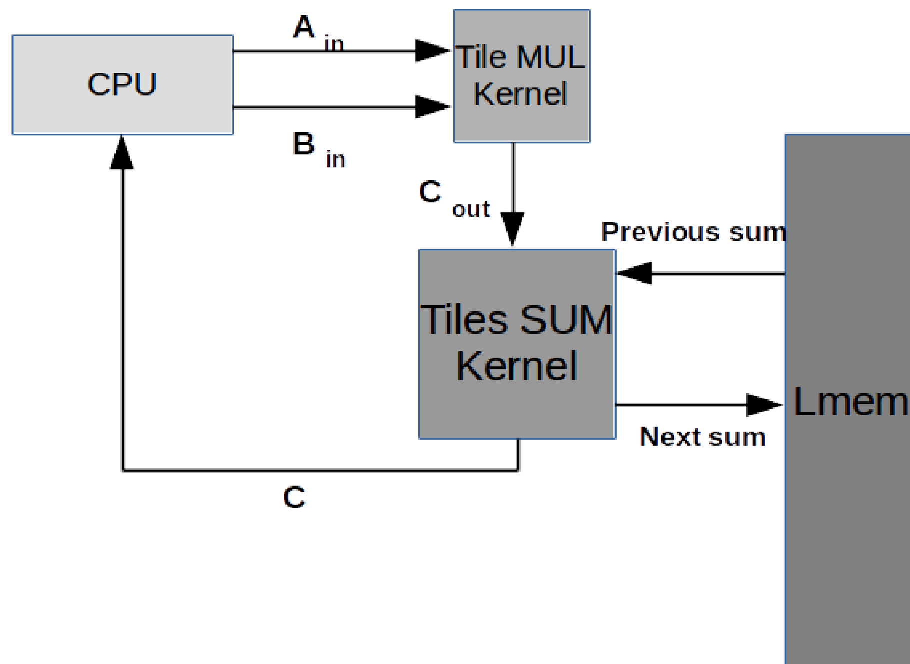


Figure 8.2: Διάγραμμα αρχιτεκτονικής

Κεφάλαιο 9

Επιτάχυνση της Cholesky παραγοντοποίησης με το μοντέλο ροής δεδομένων

Αν ο $A \in \mathbb{R}^{n \times n}$ είναι ένας συμμετρικός θετικά ορισμένος πίνακας, η Cholesky παραγοντοποίηση τον χωρίζει στο γινόμενο δύο πινάκων, ενός κάτω τριγωνικού και του αναστραμένου του.

$$A = LL^\top \tag{9.1}$$

Κάθε συμμετικός θετικά ορισμένος πίνακας έχει μια μοναδική Cholesky παραγοντοποίηση. Η Cholesky παραγοντοποίηση χρησιμοποιείται κυρίως για την λύση γραμμικών συστημάτων εξισώσεων της μορφής $AX = B$. Αν ο A είναι συμμετρικός και θετικά ορισμένος πίνακας, μπορούμε να λύσουμε το σύστημα $AX = B$, υπολογίζοντας πρώτα την Cholesky παραγοντοποίηση $A = LL^\top$ και λύνοντας στη συνέχεια το $LY = B$ με άγνωστο το Y με μπροστά αντικατάσταση και το $L^\top X = Y$ με άγνωστο το X με πίσω αντικατάσταση.

9.1 Οι αλγόριθμοι Cholesky-Banachiewicz και Cholesky-Crout

Αν γράψουμε την εξίσωση $A = LL^\top$,

$$\begin{aligned} A = LL^\top &= \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix} \\ &= \begin{pmatrix} l_{11}^2 & & (symmetric) \\ l_{21} \cdot l_{11} & l_{21}^2 + l_{22}^2 & \\ l_{31} \cdot l_{11} & l_{31} \cdot l_{21} + l_{32} \cdot l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix} \end{aligned} \tag{9.2}$$

παίρνουμε τους παρακάτω τύπους για τα στοιχεία του L:

$$\begin{aligned} l_{jj} &= \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2} \\ l_{ij} &= \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} \cdot l_{jk} \right), \quad \text{for}\, i > j. \end{aligned} \tag{9.3}$$

Σημειώνουμε ότι εφόσον οι παλαιότερες τιμές του a_{ij} δεν χρειάζονται για τον υπολογισμό νέων στοιχείων, μπορούν να γραφτούν από πάνω οι τιμές l_{ij}, επομένως ο αλγόριθμος μπορεί να εκτελεστεί χρησιμοποιώντας την ίδια μνήμη για του πίνακες A και L.

Επομένως, μπορούμε να υπολογίσουμε το (i, j) στοιχείο του L, αν γνωρίζουμε τα στοιχεία αριστερά και πάνω. Ο υπολογιμός γίνεται με έναν από τους δύο τρόπους:

- Ο αλγόριθμος Cholesky-Banachiewicz (Row-Cholesky) ξεκινά από την επάνω αριστερή γωνία του πίνακα L και συνεχίζει υπολογίζοντας τον πίνακα γραμμή-γραμμή.
- Ο αλγόριθμος Cholesky-Crout (Column-Cholesky) ξεκινά από την επάνω αριστερή γωνία του πίνακα L και συνεχίζει υπολογίζοντας τον πίνακα στήλη-στήλη.

Ο αλγόριθμος Column-Cholesky παρουσιάζεται παρακάτω, όπου ο L γράφεται πάνω στον A.

Algorithm 9.1 Αλγόριθμος Column-Cholesky

1: **procedure** Column-Cholesky(A)
2:     **for** $j \leftarrow 1, n$ **do**
3:         **for** $i \leftarrow j, n$ **do**
4:             **for** $k \leftarrow 1, j - 1$ **do**
5:                 $A_{ij} \leftarrow A_{ij} - A_{ik} \cdot A_{jk}$
6:             **if** $i = j$ **then**
7:                 $A_{ij} \leftarrow \sqrt{A_{ij}}$
8:             **else**
9:                 $A_{ij} \leftarrow A_{ij} / A_{jj}$
10: **return** A (έχει γίνει ο πίνακας L)

Αλγόριθμος 9.1: Ο αλγόριθμος βασίζεται στην εξίσωση 9.3 για τον υπολογισμό κάθε στοιχείου του L, ο οποίος χρειάζεται $\mathcal{O}(n)$ γινόμενα. Η Cholesky παραγοντοποίηση έχει πολυπλοκότητα $\mathcal{O}(n^3)$, εφόσον χρειάζεται συνολικά $\frac{1}{6}n^3$ πράξεις. Ομως ο αριθμός τον στοιχείων που παίρνουν μέρος στον υπολογισμό είναι $\mathcal{O}(n^2)$ και αυτός είναι ένας σημαντικός παράγοντας που καθηστά την Cholesky παραγοντοποίηση έναν καλό υποψήφιο πρόβλημα για επιτάχυνση με το μοντέλο ροης δεδομένων της Maxeler.

Η εξάρτηση των δεδομένων φαίνεται στο σχήμα 9.1.

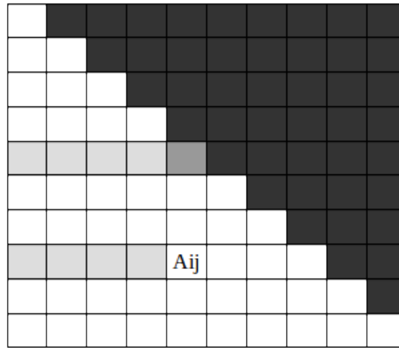


Figure 9.1: Ο υπολογισμός ενός $A_{ij}^{new} \equiv L_{ij}$ στοιχείου απαιτεί δύο γραμμές στοιχείων που έχουν υπολογιστεί προηγουμένως και τα οποία φαίνονται με ανοιχτόχρωμο γκρι. Η πρώτη είναι η γραμμή που ανήκει το στοιχείο A_{ij}^{new} και η δεύτερη είναι η γραμμή που ανήκει το διαγώνιο στοιχείο της στήλης του A_{ij}^{new}. Τέλος, χρειάζεται και το διαγώνιο στοιχείο, που φαίνεται με σκούρο γκρι.

Το μοτίβο πρόσβασης στην μνήμη φαίνεται στο σχήμα 9.2.

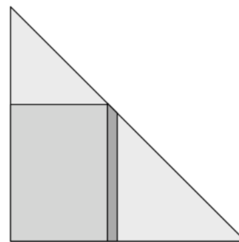


Figure 9.2: Για τον υπολογισμό μιας ολόκληρης στήλης του L (σε σκούρο γκρι), η τετραγωνική περιοχή με ανοιχτό γκρι πρέπει να διαβαστεί από τη μνήμη. Επιπλέον, η ίδια η στήλη πρέπει να διαβαστεί και να γραφτεί στη συνέχεια στη μνήμη.

9.2 Οι συναρτήσεις των scipy και numpy που θα επιταχυνθούν

Στην Python ο υπολογισμός της Cholesky παραγοντοποίησης ενός πίνακα υπολογίζεται με τη συνάρτηση

scipy.linalg.cholesky(a, lower=False, overwrite_a=False, check_finite=True)

Όλα τα προγράμματα της βιβλιοθήκης scikit-learn που ενδιαφερόμαστε να επιταχύνουμε καλούν την συνάρτηση scipy.linalg.cholesky() με lower=True, ενώ οι άλλοι παράμετροι τίθονται στην προκαθορισμένη τους τιμή.

9.3 Υλοποίση στο μοντέλο ροής δεδομένων της Maxeler

Το πρώτο θέμα που έχουμε να λάβουμε υπόψιν μας για την επιτάχυνση της Cholesky παραγοντοποίησης είναι πως ο πίνακας A θα μεταφερθεί στο DFE και πως θα αποθηκεύεται σε αυτό. Λαμβάνοντας υπόψιν την εξάρτηση των δεδομένων που φαίνονται στο σχήμα 9.2, για τον υπολογισμό μιας στήλης του L,πρέπει να έχουμε πρόσβαση σε ένα πολύ μεγάλο μέρος από τιμές του L που υπολογίστηκαν προηγουμένως.

Όπως είναι γνωστό, τα DFEs διαθέτουν δύο είδη μνήμης, την Fmem και την Lmem. Το κλειδί για μια αποδοτική υλοποίηση της Cholesky παραγοντοποίησης είναι η επαναχρησιμοποίηση των δεδομένων και η όσο το δυνατόν μικρότερη μεταφορά δεδομένων μέσα και έξω από το DFE. Επομένως για τον υπολογισμό των στοιχείων του L, όσο το δυνατόν ταχύτερα επιλέγουμε να κρατήσουμε τον πίνακα L στην Fmem μνήμη πανω στο chip.

Παρ'ολα αυτά, η Fmem δεν μπορεί να αποθηκεύσει ολόκληρο τον πίνακα L για μεγάλες τιμές του n. Για τον λόγο αυτό πρέπει να χωρίσουμε τον A σε κομμάτια (tiles), ώστε τα αντίστοιχα L tiles να χωρούν στην Fmem.

Αρχικά ο A $(n \times n)$ χωρίζεται σε $n_{tiles} \times n_{tiles}$ tiles μεγέθους $n_B \times n_B$, όπου $n = n_{tiles} \cdot n_B$,

$$A = \begin{bmatrix} A_{1,1} & A_{2,1}^\top & A_{3,1}^\top & \dots & A_{n_{tiles},1}^\top \\ A_{2,1} & A_{2,2} & A_{3,2}^\top & \dots & A_{n_{tiles},2}^\top \\ A_{3,1} & A_{3,2} & A_{3,3} & \dots & A_{n_{tiles},3}^\top \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n_{tiles},1} & A_{n_{tiles},2} & A_{n_{tiles},3} & \dots & A_{n_{tiles},n_{tiles}} \end{bmatrix} \tag{9.4}$$

Με αυτόν τον τρόπο στο πρώτο στάδιο μπορούμε να υπολογίσουμε το $L_{1,1}$ περνώντας το $A_{1,1}$ tile στο DFE: $L_{1,1}$ = cholesky_DFE($A_{1,1}$). Με τον ίδιο τρόπο μπορούμε να υπολογίσουμε τα $L_{2,1}$, $L_{3,1}$... .'Ομως αυτά τα tiles χρειάζονται δεδομένα από το $L_{1,1}$, όπως φαίνεται στο σχήμα 9.4. Για τον υπολογισμό του στοιχείου με τον αστερίσκο $(*)$ του $L_{2,1}$, η γραμμή με ανοικτό γκρι του tile $L_{1,1}$ είναι απαραίτητη. Επομένως, $L_{2,1}$ = cholesky_DFE($A_{2,1}$, $L_{1,1}$), $L_{3,1}$ = cholesky_DFE($A_{3,1}$, $L_{1,1}$) και ούτω καθεξής.

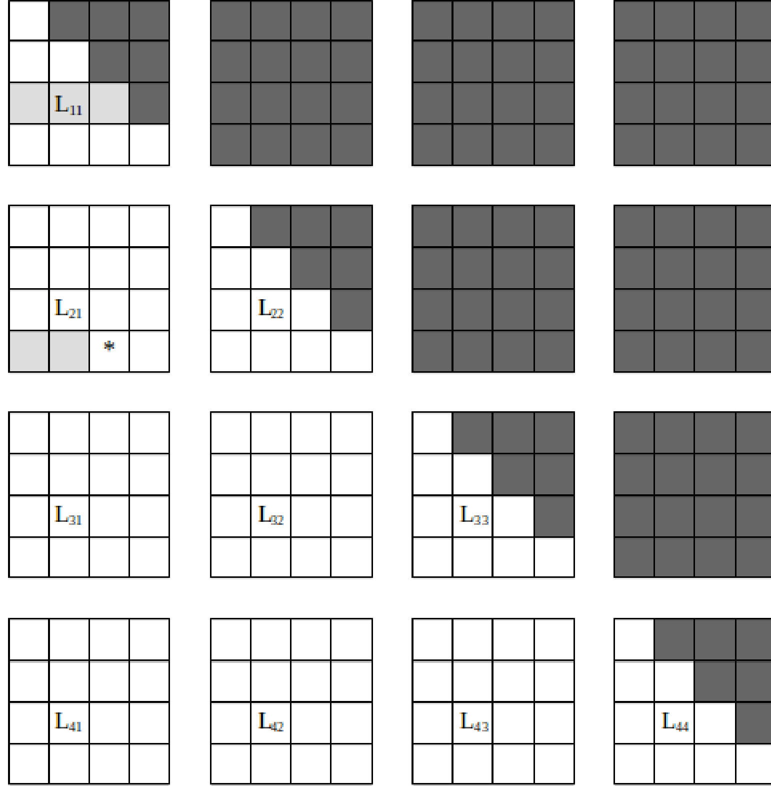


Figure 9.3: Ο πίνακας L χωρίζεται σε tiles. Η εξάρτηση δεδομένων για το στοιχείο $*$.

Στο σημείο αυτο, η πρώτη στήλη από tiles του L έχει υπολογιστεί. Πριν προχωρήσουμε στην επεξεργασία της δεύτερης στήλης από tiles, πρέπει να υπολογίσουμε τον πίνακα

$$\begin{aligned} S^{(1)} &= \begin{bmatrix} L_{2,1} \\ L_{3,1} \\ \vdots \\ L_{n_{tiles},1} \end{bmatrix} \cdot \begin{bmatrix} L_{2,1}^{\top} & L_{3,1}^{\top} & \dots & L_{n_{tiles},1}^{\top} \end{bmatrix} \\ &= \begin{bmatrix} L_{2,1}L_{2,1}^{\top} & L_{2,1}L_{3,1}^{\top} & \dots & L_{2,1}L_{n_{tiles},1}^{\top} \\ L_{3,1}L_{2,1}^{\top} & L_{3,1}L_{3,1}^{\top} & \dots & L_{3,1}L_{n_{tiles},1}^{\top} \\ \vdots & \vdots & \ddots & \vdots \\ L_{n_{tiles},1}L_{2,1}^{\top} & L_{n_{tiles},1}L_{3,1}^{\top} & \dots & L_{n_{tiles},1}L_{n_{tiles},1}^{\top} \end{bmatrix} \end{aligned} \tag{9.5}$$

ο οποίος είναι συμμετρικός. Ο πίνακας αυτός είναι το γινόμενο δύο πινάκων και επομένως για τον υπολογισμό του μπορούμε να χρησιμοποιήσουμε την Maxeler εφαρμογή πολλαπλασιασμού πινάκων της προηγούμενης ενότητας. Τώρα χρησιμοποιώντας τον $S^{(1)}$ μπορούμε να συνεχίσουμε με τον υπολογισμό της δεύτερης στήλης από tiles του L. Για να υπολογίσουμε το $L_{2,2}$, από το σχήμα 9.5 βλέπουμε ότι το στοιχείο $*$ χρειάζεται όλα τα προηγουμένως υπολογισμένα στοιχεία σε ανοικτό γκρι. Συγκεκριμένα το γινόμενο των δύο γκρι γραμμών του $L_{2,1}$ πρέπει να αφαιρεθεί από το a_*. Το γινόμενο αυτό μπορούμε να το πάρουμε απευθείας από το $S_{1,1}^{(1)}$, που περιέχει όλα τα γινόμενα γραμμής με γραμμή που χρειάζονται για τον υπολογισμό του $L_{2,2}$. Με τον ίδιο τρόπο το $S_{2,1}^{(1)}$ περιέχει όλα τα γινόμενα γραμμής με γραμμή για τον υπολογισμό

του $L_{3,2}$, το $S_{3,1}$ για το $L_{4,2}$ και ούτω καθεξής. Επιπλέον, όπως προηγουμένως τα $L_{3,2}$, $L_{4,2}$, ..., χρειάζονται δεδομένα από το $L_{2,2}$.

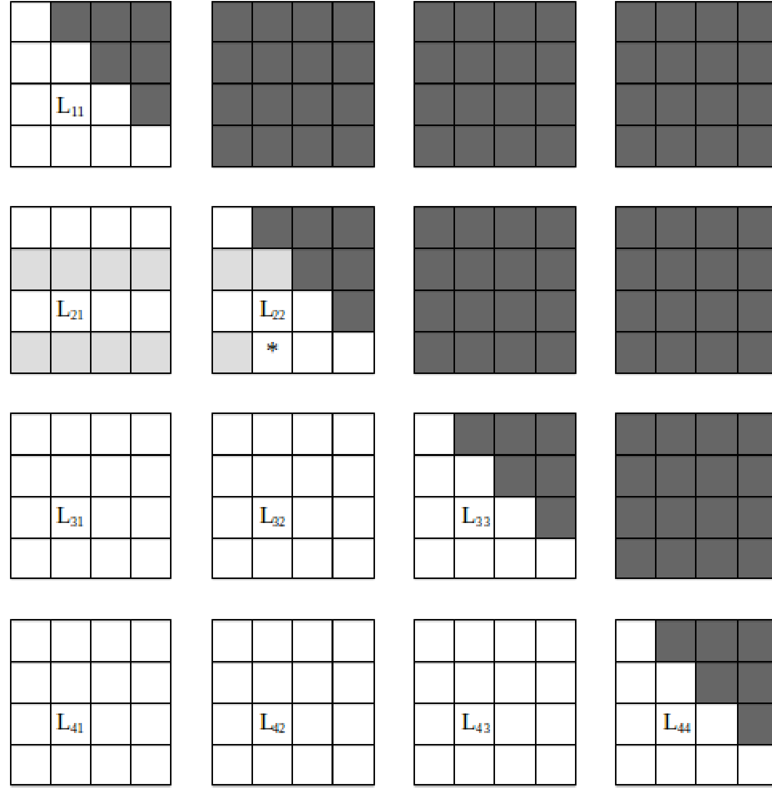


Figure 9.4: L matrix partitioned into tiles. Data dependency for $*$ element.

Στο τέλος αποθηκεύουμε έναν υποπίκανα του $S^{(1)}$.

$$S_{acc}^{(1)} = \begin{bmatrix} L_{3,1}L_{3,1}^{\top} & \dots & L_{3,1}L_{n_{tiles},1}^{\top} \\ \vdots & \ddots & \vdots \\ L_{n_{tiles},1}L_{3,1}^{\top} & \dots & L_{n_{tiles},1}L_{n_{tiles},1}^{\top} \end{bmatrix} \tag{9.6}$$

Στο δεύτερο στάδιο, παίρνουμε $L_{2,2} =$cholesky_DFE($A_{2,2}, S_{1,1}^{(1)}$) και $L_{3,2} =$cholesky_DFE($A_{3,2}, L_{2,2}, S_{2,1}^{(1)}$), $L_{4,2} =$cholesky_DFE($A_{4,2}, L_{2,2}, S_{3,1}^{(1)}$), ... Στο σημείο αυτό, έχει υπολογιστεί η δεύτερη στήλη από tiles του L. Πρωτού συνεχίσουμε τη διαδικασία για την τρίτη στήλη από tiles, πρέπει να υπολογίσουμε τον πίνακα $S^{(2)}$ με τον ίδιο τρόπο όπως κάναμε για τον $S^{(1)}$ προηγουμένως, αλλά επίσης να προσθέσουμε και τον $S_{acc}^{(1)}$, έτσι ώστε

$$\begin{aligned} S^{(2)} &= \begin{bmatrix} L_{3,2} \\ \vdots \\ L_{n_{tiles},2} \end{bmatrix} \cdot \begin{bmatrix} L_{3,2}^{\top} & \dots & L_{n_{tiles},2}^{\top} \end{bmatrix} + S_{acc}^{(1)} \\ &= \begin{bmatrix} L_{3,2}L_{3,2}^{\top} & \dots & L_{3,2}L_{n_{tiles},2}^{\top} \\ \vdots & \ddots & \vdots \\ L_{n_{tiles},2}L_{3,2}^{\top} & \dots & L_{n_{tiles},2}L_{n_{tiles},2}^{\top} \end{bmatrix} + S_{acc}^{(1)} \end{aligned} \tag{9.7}$$

Χρησιμοποιώντας τον $S^{(2)}$ μπορούμε να συνεχίσουμε για την τρίτη στήλη από tiles του L, επαναλαμβάνοντας τη διαδικασία που περιγράψαμε προηγουμένως, και τελικά να υπολογίσουμε ολόκληρο τον L.

9.3.1 Kernel

O Kernel CholeskyTileKernel είναι ο πυρήνας του υπολογισμού της Cholesky παραγοντοποίησης με το μοντέλο ροής δεδομένων, ο οποίος δέχεται ένα tile του A και στέλνει πίσω το αντίστοιχο L tile. Διαθέτει τέσσερις ακολουθίες εισόδου (input streams):

- το aIn είναι το tile του A.
- το aUp είναι το διαγώνιο tile του L, το οποίο ανήκει στην ίδια στήλη από tiles με το aIn και έχει υπολογιστεί προηγουμένως.
- το partialSum έχει το tile του S πίνακα, με τα εσωτερικά γινόμενα των αριστερότερων tiles του L, τα οποία πρέπει να αφαιρεθούν από τα στοιχεία του A tile που είναι προς υπολογισμό.
- το ctrl, το οποίο είναι ένα stream ελέγχου και είναι χρήσιμο για να αντιμετωπιστεί η εξάρτηση δεδομένων κάθε στοιχείου σε μια στήλη από το διαγώνιο στοιχείο της στήλης αυτής. Αυτό σημαίνει ότι κατά τη διάρκεια του υπολογισμού ενός διαγώνιου στοιχείου ο Kernel σταματά (stall) μέχρι να ολοκληρωθεί ο υπολογισμός.

```
1  class CholeskyTileKernel extends Kernel {
2    CholeskyTileKernel(KernelParameters parameters, DFEType type, int B, int laten) {
3      super(parameters);
4
5      //Scalar Input
6      DFEVar mode = io.scalarInput(”mode”, dfeUInt(32));
7
8      //Enable Input and Signals
9      DFEVar ctrl = io.input(”ctrl”, dfeUInt(32));
10     DFEVar EnableInOut = ctrl===1 | ctrl===4;
11     DFEVar EnableCount = ctrl===3 | ctrl===4;
12
13     //Inputs
14     DFEVar aIn = io.input(”aIn”, type, EnableInOut);
15     DFEVar partialSum = io.input(”partialSum”, type, EnableInOut);
16     DFEVar aUp = io.input(”aUp”, type, EnableInOut);
```

Οι advanced counters μας επιτρέπουν να παρατηρούμε τη θέση του στοιχείου που υπολογίζεται σε κάθε Kernel tick. Σημειώνουμε ότι λόγω της εξάρτησης δεδομένων τα στοιχεία υπολογίζονται στήλη-στήλη και το i που είναι ο μετρητής γραμμών αυξάνει σε κάθε tick. Ο μετρητής στηλών j αυξάνει όταν το i μηδενίζεται.

```
17     //Advanced counters for flow control−−Column cholesky
18     Count.Params paramsi = control.count.makeParams(32)
19     .withEnable(EnableCount)
20     .withMax(B)
21     .withInc(1);
22     Counter counteri = control.count.makeCounter(paramsi);
23     Count.Params paramsj= control.count.makeParams(32)
24     .withEnable(counteri.getWrap())
25     .withMax(B)
26     .withInc(1);
27     Counter counterj = control.count.makeCounter(paramsj);
```

```
28      DFEVar i = counteri.getCount();
29      DFEVar j = counterj.getCount();
30      DFEVar jj = j.cast(dfeInt(32));
31      DFEVar ii = i.cast(dfeInt(32));
32
33      //Down triangle computation only or whole area
34      DFEVar computeArea = (mode === 1) ? (jj <= ii) : (ii >= 0);
```

Στόχος μας είναι να υπολογίζεται ένα στοιχείο του L σε κάθε Kernel tick. Για τον λόγο αυτό μέχρι και B πολλάπλασιασμοί θα πρέπει να πραγματοποιούνται παράλληλα, εφόσον B είναι το μέγεθος του tile. Χρησιμοποιούμε την τεχνική loop unrolling και δημιουργούμε B πολλαπλασιαστές. Επίσης, όπως φαίνεται και από το σχήμα 1, δύο γραμμές από προηγουμένως υπολογισμένα στοιχεία πρέπει να πολλαπλασιαστούν μεταξύ τους και πρέπει να έχουμε πρόσβαση σε αυτά σε ένα tick. Επομένως θα χρησιμοποιήσουμε B μνήμες Fmem μεγέθους B η κάθε μια για, ώστε να αποθηκεύουμε στοιχεία του L.

```
35      //Arrays of streams to hold rows for dot product
36      DFEVar[] products = new DFEVar[B];
37      DFEVar[] upperrowElement = new DFEVar[B];
38      DFEVar[] myrowElement = new DFEVar[B];
39      DFEType addrType = dfeUInt(MathUtils.bitsToAddress(B));
40      ArrayList<Memory<DFEVar>> bBuf = new ArrayList<Memory<DFEVar>>();
41      for (int b = 0; b < B; ++b) {
42         bBuf.add(mem.alloc(aIn.getType(), B));
43      }
44
45      //Loop unrolling for parallel computation of products
46      for (int b = 0; b < B; ++b) {
47         DFEVar iread = (b < jj);
48         Memory<DFEVar> table = bBuf.get(b);
49         DFEVar rdAddrup = (jj).cast(addrType);
50         DFEVar rdAddrmy = (ii).cast(addrType);
51         DFEVar a = Reductions.streamHold(stream.offset(aUp, b−1), (ii === 0));
52         upperrowElement[b] = iread & computeArea ? ( mode ===1 ? table.read(rdAddrup) : a) : 0;
53         myrowElement[b] = iread & computeArea ? table.read(rdAddrmy) : 0;
54         optimization.pushPipeliningFactor(0.1);
55         products[b] = upperrowElement[b] * myrowElement[b];
56         optimization.popPipeliningFactor();
57      }
58
59      DFEVar upperrow = type.newInstance(this);
60      DFEVar myrow = type.newInstance(this);
61
62      products[0] = upperrow * myrow;
```

Στο επόμενο βήμα το άθροισμα των γινομένων υπολογίζεται.

```
63      //streams to hold dot product and diagonal element
64      DFEVar sum = type.newInstance(this);
65
66      //add products to produce sum
67      if (aIn.getType() instanceof DFEFloat) {
68         sum = FloatingPointMultiAdder.add(products);
69      } else {
70         sum = TreeReduce.reduce(new Add<DFEVar>(), products);
71      }
```

Στη συνέχεια, το άθροισμα αυτό και το partialSum από τα αριστερότερα tiles αφαιρείται από το aIn, και η τιμή που προκύπτει διαιρείται από το διαγώνιο στοιχείο της στήλης, ή περνά από την συνάρτηση τετραγωνικής ρίζας. Στο τέλος το αποτέλεσμα οδηγείται στην έξοδο του Kernel (output stream).

```
72      DFEVar diag = type.newInstance(this);
```

```
73|        DFEVar EnableComputation = EnableInOut & computeArea;
74|        DFEVar aNew = aIn – sum – partialSum;
75|        DFEVar aOut = EnableComputation ? ((ii === jj & mode===1) ? KernelMath.sqrt(aNew) : aNew /
  |        diag) : 0;
76|
77|        DFEVar a = Reductions.streamHold(stream.offset(aUp, jj,0,B–1), (ii === 0));
78|        diag <== (mode===1) ? stream.offset(aOut, jj – ii –laten, –B +1 –laten, –laten) : a;
79|
80|        io.output(”aOut”, aOut, type, EnableInOut);
```

Το τελευταίο βήμα είναι να γράψουμε το αποτέλεσμα στην Fmem. Όμως το στοιχείο που υπολογίζεται στο παρόν tick δεν μπορεί να αποθηκευτεί απευθείας, μιας και χρειάζεται αρκετά ticks για να είναι έτοιμο, άρα σε κάθε tick αποθηκεύουμε το αποτέλεσμα από B ticks στο παρελθόν.

```
82|        DFEVar aOutB = stream.offset(aOut, –B);
83|        DFEVar aOutBl = stream.offset(aOut, –B–laten);
84|
85|        DFEVar updif = jj – ii + (–B–laten);
86|        int updifmin = –2*B + 1 –laten;
87|        int updifmax = –B–laten;
88|        DFEVar aprev = Reductions.streamHold(stream.offset(aUp, jj–1,0,B–1), (ii === 0));
89|        upperrow <== (mode===1) ? ((ii === jj) ? aOutB : stream.offset(aOut, updif, updifmin, updifmax)) :
  |        aprev;
90|        myrow <== (ii === jj) ? aOutB : aOutBl;
91|
92|        DFEVar wrAddr = (ii).cast(addrType);
93|        DFEVar writeValue = (ii === jj ) ? aOutB : aOutBl;
94|
95|        for (int b = 0; b < B; ++b) {
96|            bBuf.get(b).write(wrAddr, writeValue, (EnableInOut & jj===b));
97|        }
98|    }
99| }
```

9.3.2 Διάγραμμα αρχιτεκτονικής

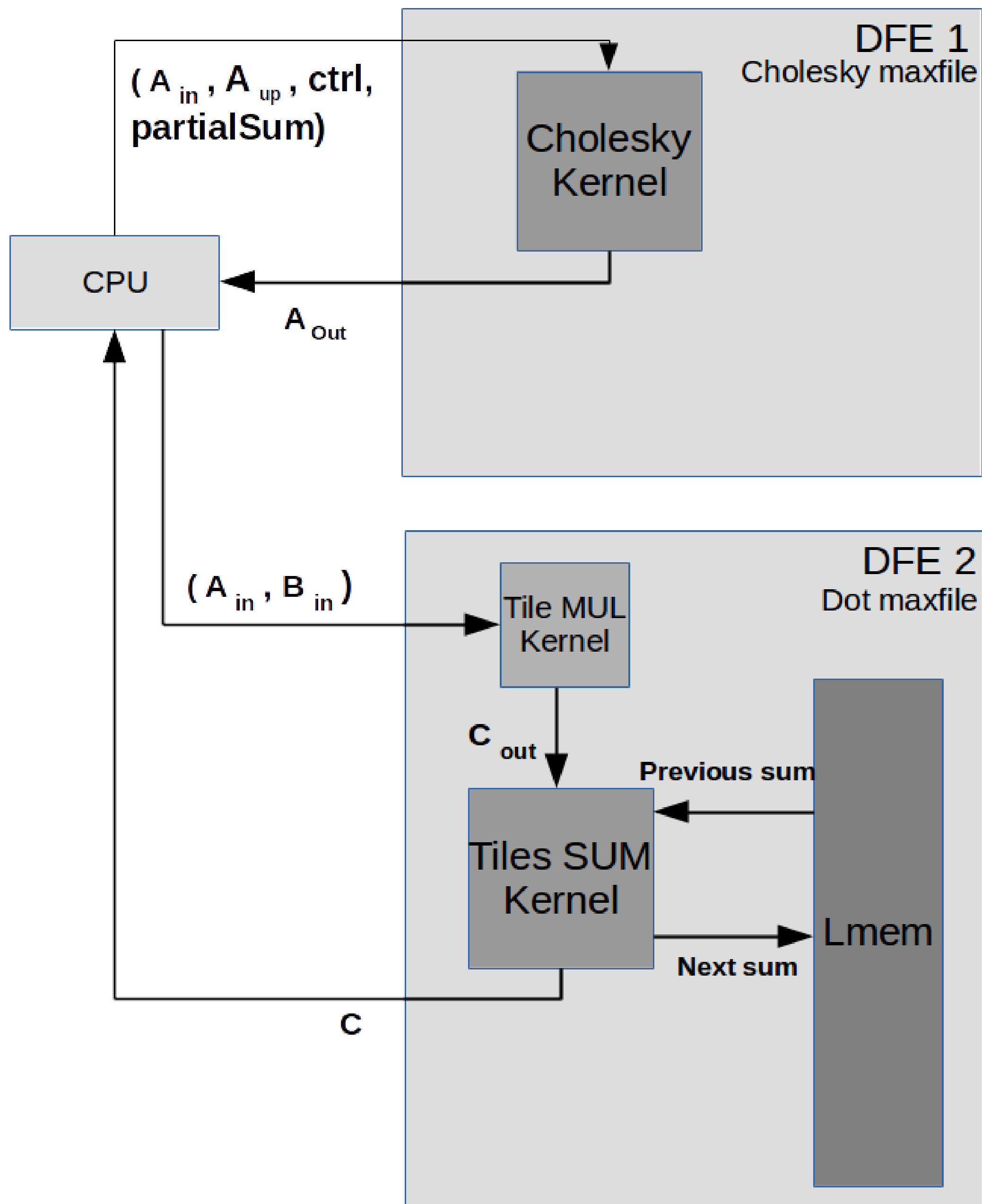


Figure 9.5: Για την Cholesky παραγοντοποίηση χρειάζονται δύο υλοποιήσεις προγραμμάτων της Maxeler. Μία για τον υπολογισμό του L και μια για τον πολλαπλασιασμό πινάκων.

Κεφάλαιο 10

Επιτάχυνση του Cholesky solve και Lower triangular solve με το μοντέλο ροής δεδομένων

10.1 Lower triangular solve

Υποθέτουμε ότι το L είναι ένας κάτω τριγωνικός πίνακας μεγέθους $n \times n$ και ο B είναι ένας πίνακας μεγέθους $n \times m$. Τότε το γραμμικό σύστημα $LX = B$ μπορεί να λυθεί με μπροστά αντικατάσταση. Το σύστημα μπορεί να γραφτεί ως

$$\begin{bmatrix} l_{1,1} & 0 & \dots & 0 \\ l_{2,1} & l_{2,2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \dots & l_{n,n} \end{bmatrix} \cdot \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,m} \\ x_{2,1} & x_{2,2} & \dots & x_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \dots & x_{n,m} \end{bmatrix} = \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,m} \\ b_{2,1} & b_{2,2} & \dots & b_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \dots & b_{n,m} \end{bmatrix} \tag{10.1}$$

ή

$$\begin{array}{lllll} l_{1,1} \cdot x_{1,j} & & & & = b_{1,j} \\ l_{2,1} \cdot x_{1,j} & + & l_{2,2} \cdot x_{2,j} & & = b_{2,j} \\ \quad \vdots & \vdots & \ddots & & \vdots \\ l_{n,1} \cdot x_{1,j} & + & l_{n,2} \cdot x_{2,j} & + \cdots + \; l_{n,n} \cdot x_{n,j} & = b_{n,j} \end{array} \qquad j = 1, ..., m. \tag{10.2}$$

Παρατηρούμε ότι η πρώτη εξίσωση $l_{1,1} \cdot x_{1,j} = b_{1,j}$ περιέχει μόνο το $x_{1,j}$ και επομένως μπορεί να λυθεί απευθείας. Η δεύτερη εξίσωση περιλαμβάνει τα $x_{1,j}$ και $x_{2,j}$, μπορεί επομένως να λυθεί αν αντικαταστήσουμε την τιμή του $x_{1,j}$ που βρέθηκε προηγουμένως. Συνεχίζοντας με αυτόν τον τρόπο, η k-οστή εξίσωση περιλαμβάνει τα $x_{1,j}$ ... $x_{k,j}$ και μπορούμε να την λύσουμε ως προς $x_{k,j}$, χρησιμοποιώντας τις τιμές των $x_{1,j}$ ... $x_{k-1,j}$.

Έτσι προκύπτουν οι τύποι:

$$
\begin{aligned}
x_{1,j} &= \frac{b_{1,j}}{l_{1,1}}, \\
x_{2,j} &= \frac{b_{2,j} - l_{2,1} \cdot x_{1,j}}{l_{2,2}}, \\
&\vdots \\
x_{n,j} &= \frac{b_{n,j} - \sum_{k=1}^{n-1} l_{n,k} \cdot x_{k,j}}{l_{n,n}}.
\end{aligned}
\qquad j = 1, ..., m. \tag{10.3}
$$

Algorithm 10.1 Αλγόριθμος μπροστά αντικατάστασης

1: **procedure** forward-solve(L, B)
2:     **for** $j \leftarrow 1, m$ **do**
3:         **for** $i \leftarrow 1, n$ **do**
4:             $X_{ij} \leftarrow B_{ij}$
5:             **for** $k \leftarrow 1, i-1$ **do**
6:                 $X_{ij} \leftarrow X_{ij} - L_{ik} \cdot X_{kj}$
7:             $X_{ij} \leftarrow X_{ij} / L_{ii}$
8: **return** X

Αλγόριθμος 10.1: Η συνολική πολυπλοκότητα του Αλγορίθμου είναι $\mathcal{O}(mn^2)$.

10.2 Cholesky solve

Έστω A ένας συμμετρικός θετικά ορισμένος πίνακας μεγέθους $n \times n$ και B ένας πίνακας μεγέθους $n \times m$. Στην περίπτωση αυτή, το σύστημα $AX = B$ έχει μια μοναδική λύση, εφόσον το A μπορεί να παραγοντοποιηθεί χρησιμοποιώντας την παραγοντοποίηση Cholesky σε ένα γινόμενο $LL^\top$. Το γραμμικό σύστημα μπορεί να γραφτεί ως

$$LL^\top X = B \tag{10.4}$$

και θέτοντας $L^\top X = Y$ το σύστημα μπορεί να λυθεί με μπροστά αντικατάσταση στο κάτω τριγωνικό σύστημα $LY = B$ και στη συνέχεια με μια πίσω αντικατάσταση στο επάνω τριγωνικό σύστημα $L^\top X = Y$.

Ο πίνακας $L^\top$ είναι επάνω τριγωνικός και το σύστημα $L^\top X = Y$ λύνεται με ανάλογο τρόπο με το κάτω τριγωνικό σύστημα.

Οι τύποι που προκύπτουν είναι ανάλογοι με τους τύπους στην 10.3:

$$\left.\begin{aligned} x_{n,j} &= \frac{y_{n,j}}{l_{n,n}}, \\ x_{n-1,j} &= \frac{y_{n-1,j} - l_{n,n-1} \cdot x_{n-1,j}}{l_{n-1,n-1}}, \\ &\vdots \\ x_{1,j} &= \frac{y_{1,j} - \sum_{k=2}^{n} l_{k,1} \cdot x_{k,j}}{l_{1,1}}. \end{aligned}\right. \qquad j = 1, ..., m. \tag{10.5}$$

Οπότε η πίσω αντικατάσταση είναι ουσιαστικά ίδια με την μπροστά αντικατάσταση, με την διαφορά ότι δουλεύει ανάποδα.

10.3 Οι συναρτήσεις των scipy και numpy που θα επιταχυνθούν

- scipy.linalg.solve(L, B)
  όπου το L είναι ένας κάτω τριγωνικός πίνακας μεγέθους $n \times n$ και ο B είναι ένας πίνακας μεγέθους $n \times m$.
- scipy.linalg.cho_solve(L, B)
  όπου το L είναι ο πίνακας που προκύπτεί από την Cholesky παραγοντοποίηση του A και έχει μέγεθος $n \times n$, ενώ ο B πίνακας έχει μέγεθος $n \times m$.

10.4 Υλοποίση στο μοντέλο ροής δεδομένων της Maxeler

10.4.1 Μπροστά αντικατάσταση με tiles

Το πρώτο βήμα είναι να χωρίσουμε τον πίνακα $L_{n\times n}$ και $B_{n\times m}$ σε κομμάτια (tiles)

$$L = \begin{bmatrix} L_{1,1} & \mathbf{0} & \dots & \mathbf{0} \\ L_{2,1} & L_{2,2} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ L_{nt,1} & L_{nt,2} & \dots & L_{nt,nt} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} & \dots & B_{1,mt} \\ B_{2,1} & B_{2,2} & \dots & B_{2,mt} \\ \vdots & \vdots & \ddots & \vdots \\ B_{nt,1} & B_{nt,2} & \dots & B_{nt,mt} \end{bmatrix} \tag{10.6}$$

όπου $nt = \dfrac{n}{TILE_SIZE}$, $mt = \dfrac{m}{TILE_SIZE}$ και οι πίνακες (tiles) $L_{i,i}$ είναι κάτω τριγωνικοί.

Για ευκολία ας υποθέσουμε ότι $nt = 4$ και $mt = 2$, αν και ο αλγόριθμος δουλεύει για κάθε μέγεθος των nt και mt.

Οι πίνακες χωρισμένοι σε tiles φαίνονται στο σχήμα 10.1.

L_{11}	**0**	**0**	**0**
L_{21}	L_{22}	**0**	**0**
L_{31}	L_{32}	L_{33}	**0**
L_{41}	L_{42}	L_{43}	L_{44}

B_{11}	B_{12}
B_{21}	B_{22}
B_{31}	B_{32}
B_{41}	B_{42}

Figure 10.1: Τα tiles με ανοιχτόχρωμο γκρι στέλνονται στο DFE.

Το επόμενο βήμα είναι η αποστολή της πρώτης στήλης από tiles του L και του B στο DFE για μια εκτέλεση. Το αποτέλεσμα που επιστρέφει από το DFE είναι μια στήλη από tiles $X_1^{(1)}$ η οποία γράφεται πάνω στον B, όπως φαίνεται στο σχήμα 10.2.

L_{11}	**0**	**0**	**0**
L_{21}	L_{22}	**0**	**0**
L_{31}	L_{32}	L_{33}	**0**
L_{41}	L_{42}	L_{43}	L_{44}

$X_{11}^{(1)}$	B_{12}
$X_{21}^{(1)}$	B_{22}
$X_{31}^{(1)}$	B_{32}
$X_{41}^{(1)}$	B_{42}

Figure 10.2: Τα tiles $X_{11}^{(1)}$, $X_{21}^{(1)}$,$X_{31}^{(1)}$,$X_{41}^{(1)}$ υπολογίζονται στο DFE. Στο σημείο αυτό το πρώτο tile σε σκούρο γκρι έχει υπολογιστεί εντελώς.

Ο Kernel του DFE υλοποιεί τον παρακάτω αλγόριθμο:

Algorithm 10.2 DFE Kernel

1: **procedure** DFE Kernel(L, B (τα οποία είναι στήλες από tiles))
2: **for** All tiles in column **do**
3: **if** first tile **then**
4: **for** $i \leftarrow 1$, TILE_SIZE **do**
5: **for** $j \leftarrow 1$, TILE_SIZE **do**
6: $X_{ij}^{firsttile} \leftarrow B_{ij}^{firsttile} - \sum_{k=1}^{i-1} L_{ik}^{firsttile} X_{kj}^{firsttile}$
7: $X_{ij}^{firsttile} \leftarrow X_{ij}^{firsttile} / L_{ii}^{firsttile}$
8: **else**
9: **for** $i \leftarrow 1$, TILE_SIZE **do**
10: **for** $j \leftarrow 1$, TILE_SIZE **do**
11: $X_{ij}^{othertile} \leftarrow B_{ij}^{othertile} - \sum_{k=1}^{TILE_SIZE} L_{ik}^{othertile} X_{kj}^{firsttile}$
12: **return** X

Στο δεύτερο βήμα γίνεται μια ακόμα εκτέλεση στο DFE με εισόδους τα tiles με ανοικτό γκρι, όπως φαίνεται στο σχήμα 10.3.

L_{11}	**0**	**0**	**0**
L_{21}	L_{22}	**0**	**0**
L_{31}	L_{32}	L_{33}	**0**
L_{41}	L_{42}	L_{43}	L_{44}

$X_{11}^{(1)}$	B_{12}
$X_{21}^{(1)}$	B_{22}
$X_{31}^{(1)}$	B_{32}
$X_{41}^{(1)}$	B_{42}

Figure 10.3: Τα tiles με ανοιχτόχρωμο γκρι στέλνονται στο DFE. Στο σημείο αυτό το πρώτο tile σε σκούρο γκρι έχει υπολογιστεί εντελώς.

Η εκτέλεση αυτή θα επιστρέψει έναν πίνακα $X_1^{(2)}$, ο οποίος γράφεται πάνω στα ανοικτά γκρι tiles του $X_1^{(1)}$, όπως φαίνεται στο σχήμα 10.4.

L_{11}	**0**	**0**	**0**
L_{21}	L_{22}	**0**	**0**
L_{31}	L_{32}	L_{33}	**0**
L_{41}	L_{42}	L_{43}	L_{44}

$X_{11}^{(1)}$	B_{12}
$X_{21}^{(2)}$	B_{22}
$X_{31}^{(2)}$	B_{32}
$X_{41}^{(2)}$	B_{42}

Figure 10.4: Τα tiles $X_{21}^{(2)}$,$X_{31}^{(2)}$,$X_{41}^{(2)}$ υπολογίζονται στο DFE. Τώρα το δεύτερο σκούρο γκρι tile έχει υπολογιστεί εντελώς.

Η διαδικασία αυτή συνεχίζεται μέχρι να πάρουμε και το $X_{41}^{(4)}$, και στο σημείο αυτό η πρώτη στήλη από tiles του X είναι εντελώς υπολογισμένη. Η ίδια ακριβώς διαδικάσία επαναλαμβάνεται για τη δεύτερη στήλη από tiles του B, με αποτέλεσμα τον υπολογισμό ολόκληρου του πίνακα X.

10.4.2 Cholesky Solve με tiles

Το γραμμικό σύστημα

$$LL^{\top}X = B \tag{10.7}$$

θέτοντας $L^{\top}X = Y$, μπορεί να λυθεί με μπροστά αντικατάσταση στο κάτω τριγωνικό σύστημα $LY = B$, ακολουθούμενη από μια πίσω αντικατάσταση στο επάνω τριγωνικό σύστημα $L^{\top}X = Y$.

Το βήμα της μπροστά αντικατάστασης παρουσιάστηκε προηγουμένως, οπότε εστιάζουμε το ενδιαφέρον μας στην πίσω αντικατάσταση στο επάνω τριγωνικό σύστημα $L^{\top}X = Y$. Προκειμένου να λύσουμε το πρόβλημα αυτό μπορούμε να μετατρέψουμε την πίσω αντικατάσταση σε μπροστά και να χρησιμοποιήσουμε την λύση της μπροστά αντικατάστασης με tiles.

Αρχικά παίρνουμε τον νέο πίνακα L' από τον $L^{\top}$:

Algorithm 10.3 Μεταροπή του $L^{\top}$ στον L', ο οποίος είναι ένας κάτω τριγωνικός πίνακας.

1: **procedure** tranform-$L^{\top}(L^{\top})$
2:     **for** $i \leftarrow 1, n$ **do**
3:         **for** $j \leftarrow 1, n$ **do**
4:             $L'_{i,j} \leftarrow L_{n+1-i,n+1-j}$
    return L'
5:

Έπειτα ο νέος πίνακας Y' προκύπτει από τον Y:

Algorithm 10.4

```
1: procedure upsidedown-Y(Y)
2:     for i ← 1, n do
3:         for j ← 1, n do
4:             Y'_{i,j} ← L_{n+1−i,j}
       return Y'
5:
```

Στο σημείο αυτό μπορούμε να λύσουμε το σύστημα $L'X' = Y'$ χρησιμοποιώντας την λύση της μπροστά αντικατάστασης με tiles και να πάρουμε τον X'. Τέλος, ο επιθυμητός πίνακας X που είναι η λύση του αρχικού συστήματος προκύπτει χρησιμοποιώντας τον παραπάνω αλγόριθμο στο X'.

10.4.3 Kernel

Η υλοποίηση στο μοντέλο ροής δεδομένων του Αλγορίθμου 6.5 περιέχεται στον παρακάτω Kernel. Μια στήλη από tiles του L μεταφέρεται στον Kernel μέσω της ακολοθίας εισόδου (input stream) aIn, και μια στήλη από tiles του B μεταφέρεται στον Kernel μέσω του input stream bIn. Μετρητές και σήματα ελέγχου δημιουργούνται στον Kernel για τη σωστή επεξεργασία των δεδομένων.

```
1  class CholeskySolveTileKernel extends Kernel {
2    CholeskySolveTileKernel(KernelParameters parameters, DFEType type, int B) {
3      super(parameters);
4
5      //Inputs
6      DFEVar aIn = io.input(”aIn”, type);
7      DFEVar bIn = io.input(”bIn”, type);
8
9      //Counters
10     CounterChain cc = control.count.makeCounterChain();
11     DFEVar i = cc.addCounter(B, 1);
12     DFEVar j = cc.addCounter(B, 1);
13     DFEVar ii = i.cast(dfeUInt(32));
14     DFEVar jj = j.cast(dfeUInt(32));
15     Count.Params paramsC = control.count.makeParams(32)
16     .withMax(B*(B+1))
17     .withInc(1)
18     .withWrapMode(WrapMode.STOP_AT_MAX);
19     Counter counter = control.count.makeCounter(paramsC);
20     DFEVar count = counter.getCount();
21     DFEVar c = count.cast(dfeUInt(32));
22
23     //Enable signals
24     DFEVar allFromMem = (c === B*(B+1));
25     DFEVar secondTile = (c>B*B−1);
26     DFEVar secondTileFirstRow = secondTile & ii===0;
27     DFEVar enableWrite = c < B*(B+1);
```

Η υψηλή απόδοση επιτυγχάνεται χρησιμοποιώντας την τεχνική loop unrolling. Τα γινόμενα μέσα στο άθροισμα των γραμμών 6 και 11 του Αλγορίθμου 10.2 υπολογίζονται παράλληλα. Για τον λόγο αυτόν, το πρώτο tile με τα αποτελέσματα X αποθηκεύεται στην Fmem, ώστε να μπορούμε να έχουμε γρήγορη πρόσβαση στα στοιχεία που συμμετέχουν στους πολλαπλασιασμούς.

```
28	//Arrays of streams to hold rows for dot product
29	DFEVar[] products = new DFEVar[B+1];
30	DFEVar[] a = new DFEVar[B+1];
31	DFEVar[] bprev = new DFEVar[B+1];
32
33	DFEType addrType = dfeUInt(MathUtils.bitsToAddress(B));
34	ArrayList<Memory<DFEVar>> bBuf = new ArrayList<Memory<DFEVar>>();
35	for (int b = 0; b < B+1; ++b) {
36	   bBuf.add(mem.alloc(bIn.getType(), B));
37	}
38
39	//Loop unrolling for parallel computation of products
40	for (int b = 0; b < B+1; ++b) {
41	   DFEVar iread = ((b < ii) | allFromMem | (secondTileFirstRow & b!=B));
42	   a[b] = Reductions.streamHold(stream.offset(aIn, b−1), (j === 0));
43	   Memory<DFEVar> table = bBuf.get(b);
44	   DFEVar rdAddr = (jj).cast(addrType);
45	   bprev[b] = table.read(rdAddr);
46	   optimization.pushPipeliningFactor(0.1);
47	   products[b] = iread ? a[b] * bprev[b] : 0;
48	   optimization.popPipeliningFactor();
49	}
50
51	// Product of exactly previous BOut with a
52	DFEVar bprevv = type.newInstance(this);
53	DFEVar afst = Reductions.streamHold(stream.offset(aIn, ii−1, 0, B−2), (j === 0));
54	DFEVar asec = Reductions.streamHold(stream.offset(aIn, B−1), (j === 0));
55	DFEVar aa = secondTile ? asec : afst;
56	products[0] = ((ii > 0 | secondTile) & (allFromMem===0)) ? aa * bprevv : 0;
57
58	// sum of products
59	DFEVar sum = type.newInstance(this);
60	if (aIn.getType() instanceof DFEFloat) {
61	   sum = FloatingPointMultiAdder.add(products);
62	} else {
63	   sum = TreeReduce.reduce(new Add<DFEVar>(), products);
64	}
```

Όταν υπολογιστεί και το άθροισμα των γινομένων, πρέπει στη συνέχεια να αφαιρεθεί από το αντίστοιχο στοιχείο του B που βρίσκεται στο stream bIn. Το αποτέλεσμα αυτό στη συνέχεια διαιρείται από το αντίστοιχο διαγώνιο στοιχείο του L, αν βρισκόμαστε στην επεξεργασία του πρώτου tile, αλλιώς περνάει απευθείας στη έξοδο του Kernel. Τέλος αν το στοιχείο που υπολογίστηκε ανήκει στο πρώτο tile του X, αποθηκεύεται στην Fmem.

```
65	DFEVar bNew = bIn − sum;
66	DFEVar adiag = Reductions.streamHold(stream.offset(aIn, ii,0,B−1), (jj === 0));
67	DFEVar bOut = (secondTile | adiag===0) ? bNew : bNew / adiag;
68	io.output(”bOut”, bOut, type);
69
70	bprevv <== stream.offset(bOut, −B);
71
72	// Write Bout to memory
73	DFEVar wrAddr = (jj).cast(addrType);
74	DFEVar writeValue = stream.offset(bOut, −B);
75
76	for (int b = 0; b < B+1; ++b) {
77	   bBuf.get(b).write(wrAddr, writeValue, enableWrite & ((ii===b) | secondTile & b==B));
78	}
79
80	 }
81	}
```

Κεφάλαιο 11

Σύγκριση των υλοποιήσεων σε ροή δεδομένων με τις numpy και scipy συναρτήσεις

11.1 Χρόνος εκτέλεσης και επιτάχυνση

Στην ενότητα αυτή συγκρίνουμε τον χρόνο εκτέλεσης των προγραμμάτων στο μοντέλο ροής δεδομένων της Maxeler με τον αντίστοιχο χρόνο εκτέλεσης των αρχικών προγραμμάτων των βιβλιοθηκών scipy και numpy. Συγκεκριμένα ενδιαφερόμαστε για την επιτάχυνση (speedup) που παρέχουν οι νέες υλοποιήσεις καθώς το μέγεθος των πινάκων εισόδου των προγραμμάτων αυξάνει.

Τα αρχικά προγράμματα σε Python (CPU) έτρεξαν σε ένα πυρήνα ενός Intel Core i5-3230M @ 2.60GHz επεξεργαστή, ενώ τα προγράμματα ροής δεδομένων σε προσομοίωση του Max-Compiler, στον ίδιο επεξεργαστη και ο χρόνος εκτέλεσης στα DFEs προσεγγίστηκε με τύπο.

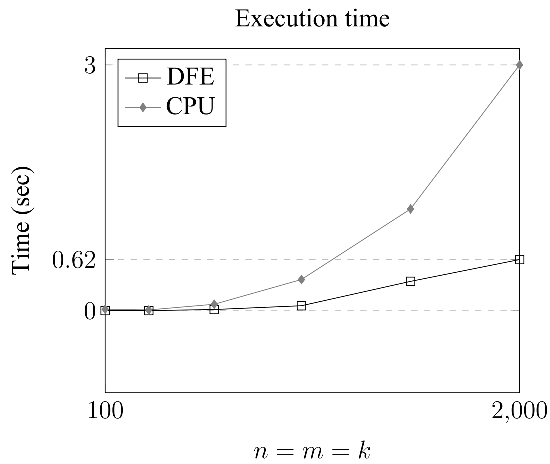
11.1.1 Πολλαπλασιασμός πινάκων

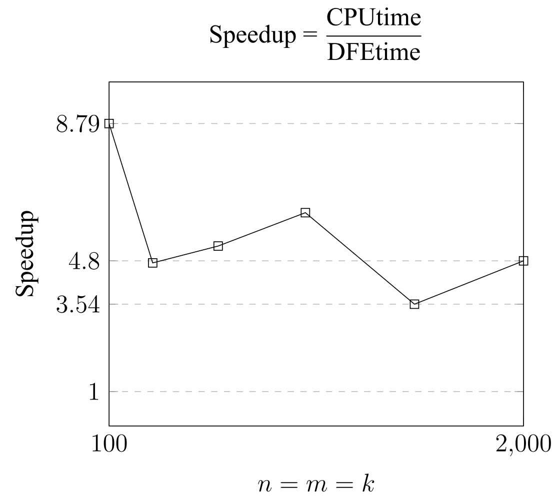
numpy συνάρτηση (CPU):

```
C = numpy.dot(A, B)
```

όπου ο A είναι ένας $m \times k$ πίνακας και ο B ένας $k \times n$ πίνακας.

Για να πετύχουμε την καλύτερη δυνατή απόδοση, θέλουμε να εκτελούμε όσο περισσότερους παράλληλους υπολογισμούς γίνεται. Όμως ο μέγιστος αριθμός των παράλληλων υπολογισμών είναι ίσος με το μέγεθος της διάστασης του tile, άρα χρειαζόμαστε μεγάλα tiles. Όμως το μέγεθος του tile περιορίζεται από τον μέγεθος της Fmem, των DSPs για πολλαπλασιαστές, των LUTs και των FFs για προσθέσεις. Επιλέξαμε tile μεγέθους 336×336 και συχνότητα ρολογιού 200MHz.





Όπως φαίνεται από το πρώτο διάγραμμα, ο χρόνος εκτέλεσης του προγράμματος ροής δεδομένων αυξάνει με μικρότερο ρυθμό από τον χρόνο εκτέλεσης του numpy dot. Επίσης το speedup είναι κοντά στο $\times 4.8$ για τα περισσότερα μεγέθη των A και B πινάκων.

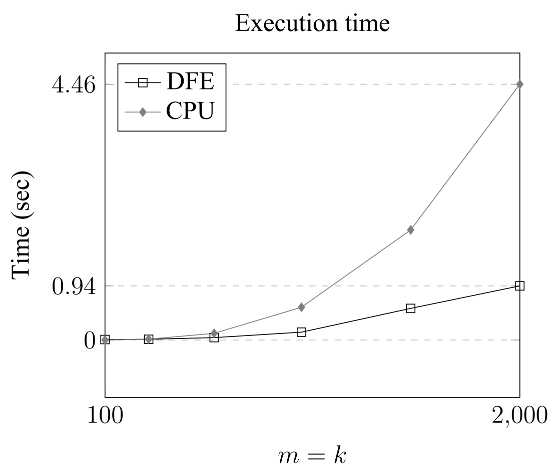
11.1.2 Απόσταση πινάκων

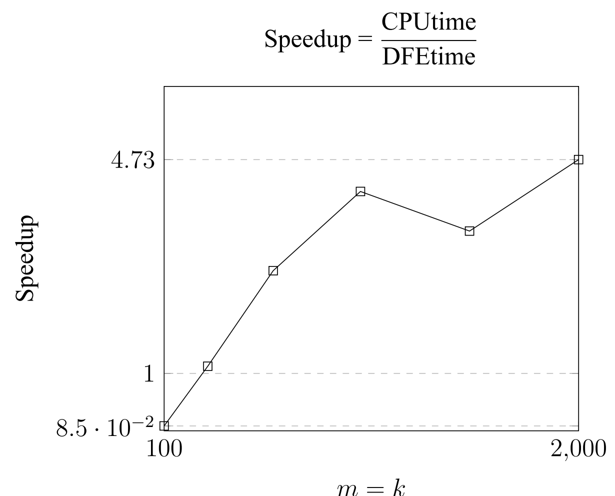
Pdist

scipy και numpy συναρτήσεις (CPU):

```
dists = scipy.spatial.distance.pdist(A, metric='sqeuclidean')
K = numpy.exp(-.5 * dists)
K = scipy.spatial.distance.squareform(K)
numpy.fill_diagonal(K, 1)
```

όπου το A είναι ένας $m \times k$ πίνακας εισόδου, και ο K είναι ένας $m \times m$ πίνακας εξόδου με τις αποστάσεις περασμένες από την συνάρτηση exp(). Επιλέξαμε tile μεγέθους 192×192 ώστε η εφαρμογή να ”χωράει” σε ένα Maia DFE. Η συχνότητα ρολογιού του DFE τέθηκε στα 200MHz.



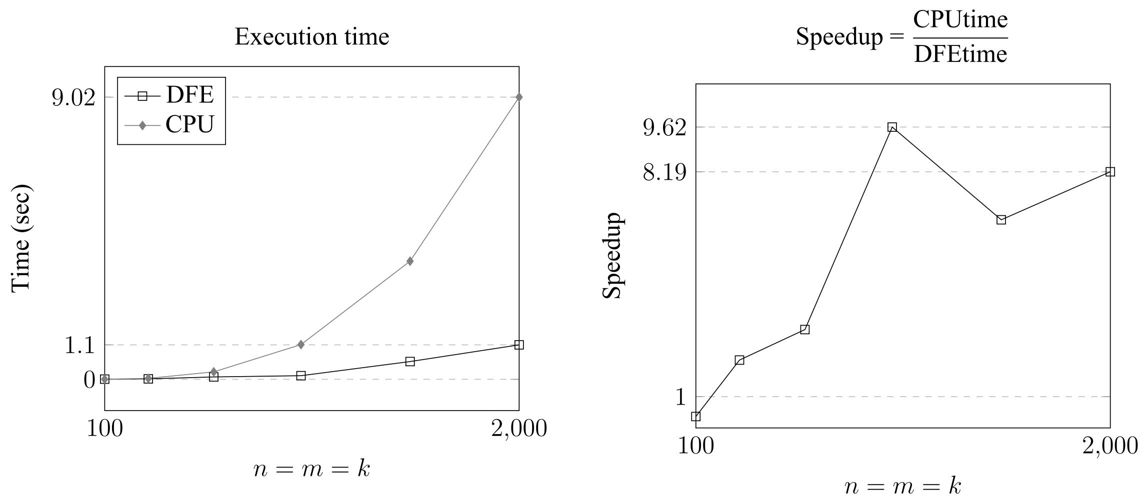


Cdist

scipy και numpy συναρτήσεις (CPU):

```
dists = scipy.spatial.distance.cdist(A, B, metric='sqeuclidean')
K = numpy.exp(-.5 * dists)
```

όπου τα A, B είναι πίνακες μεγέθους $n \times k$ και $m \times k$ αντίστοιχα και το K ο πίνακας εξόδου μεγέθους $n \times m$. Το πρόγραμμα αυτό χρησιμοποιεί το ίδιο maxfile με την εφαρμογή ροής δεδομένων για το Pdist.



Για μεγάλουε πίνακες το speedup είναι κοντά στο $\times 8$ και φαίνεται να αυξάνει. Παρ'ολα αυτά το sppedup για μικρότερους πίνακες δεν είναι σημαντικό.

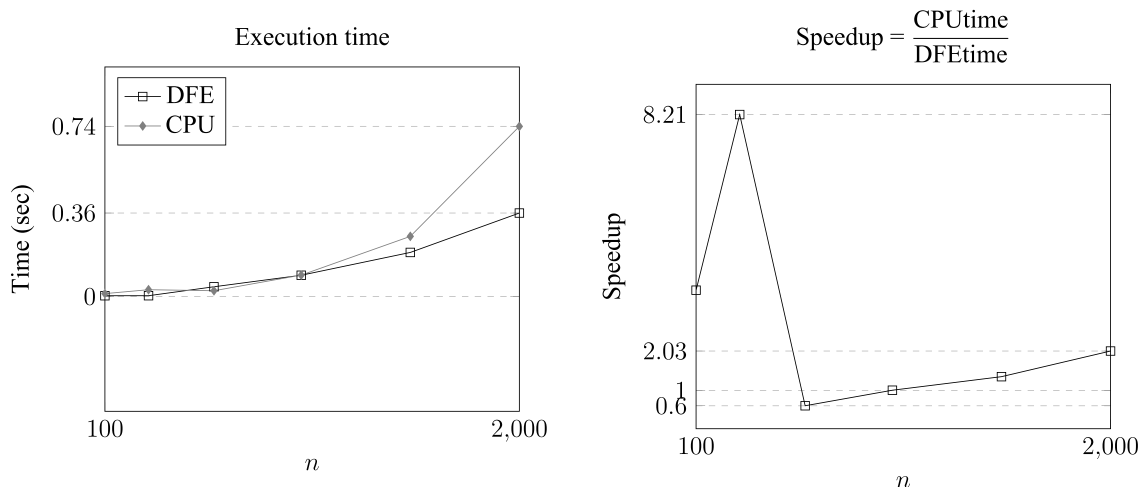
11.1.3 Cholesky παραγοντοποίηση

scipy συνάρτηση (CPU):

```
L = scipy.linalg.cholesky(A, lower=True)
```

όπου ο A είναι ένας συμμετρικός θετικά ορισμένος πίνακας μεγέθους $n \times n$. Η Cholesky παραγοντοποίηση στο μοντέλο ροής δεδομένων της Maxeler χρησιμοποιεί δύο maxfiles. Το πρώτο, το οποίο υπολογίζει την παραγοντοποίηση ενός tile του πίνακα A, επιλέχθηκε να έχει tile μεγέθους 336×336 και συχνότητα ρολογιού 200 MHz. Το δεύτερο maxfile είναι αυτό που χρησιμοποιείται για τον πολλαπλασιασμό δύο tiles και διαθέτει ίδιο μέγεθος tile και ίδια συχνότητα ρολογιού.

Ο χρόνος εκτέλεσης και το speedup φαίνονται παρακάτω:

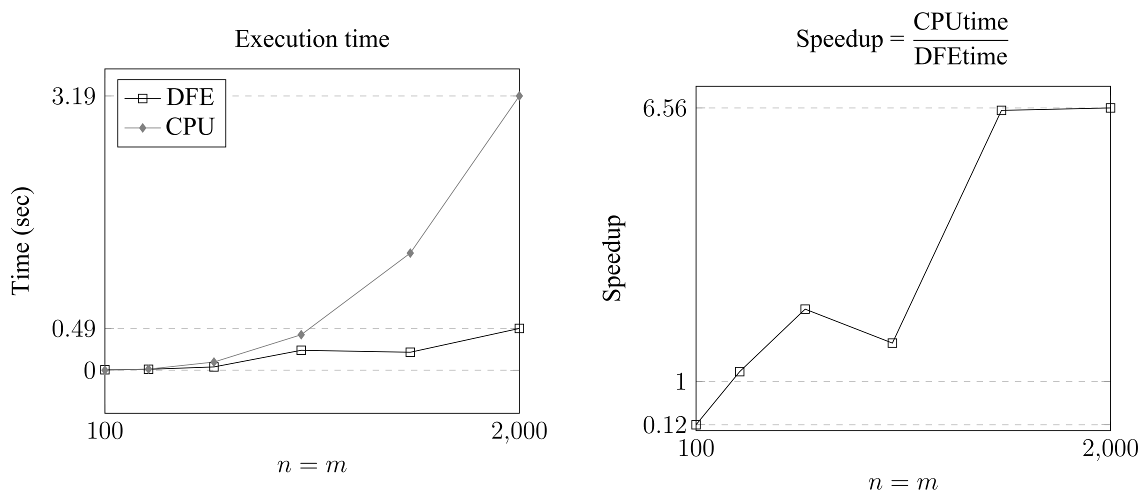


11.1.4 Cholesky Solve

scipy συνάρτηση (CPU):

```
X = scipy.linalg.cho_solve((L, True), B)
```

Ο πίνακας L είναι ο κάτω τριγωνικός πίνακας μεγέθους $n \times n$ που προκύπτει από την Cholesky παραγοντοποίηση ενός πίνακα A. Επιπλέον ο πίνακας X είναι η λύση της εξίσωσης $AX = B$ και έχει μέγεθος $n \times m$, όπως και ο B. Η υλοποίηση στο μοντέλο ροής δεδομένων διαθέτει μέγεθος tile 416×416, ενώ η συχνότητα του ρολογιού είναι ρυθμισμένη στα 200 MHz. Όπως και για τα άλλα maxfiles, το μέγεθος του tile επηρεάζει την απόδοση, όμως δεν μπορεί να αυξηθεί περισσότερο λόγω των περιορισμών που θέτει το υλικό.



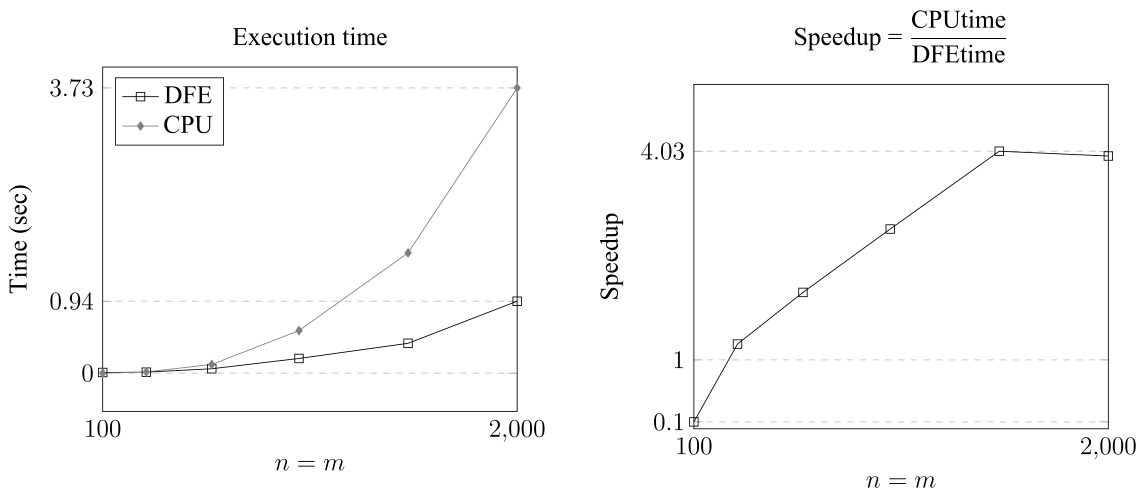
Όπως μπορούμε να δούμε από το δεξί διάγραμμα, το speedup είναι περίπου $\times 6.5$ για μεγάλους πίνακες.

11.1.5 Solve για συμμετρικό θετικά ορισμένο πίνακα

scipy function (CPU):

```
X = scipy.linalg.solve(A, B, sym_pos=True)
```

Ο A είναι πίνακας μεγέθους $n \times n$ και ο B πίνακας μεγέθους $n \times m$. Το πρόγραμμα στο μοντέλο ροής δεδομένων χρησιμοποιεί πρώτα την παραγοντοποίηση Cholesky σε DFE και τη συνέχεια την υλοποίηση Cholesky Solve στο μοντέλο ροής δεδομένων για να λύσει το σύστημα. Για τον λόγο αυτό και τα τρία maxfiles χρησιμοποιούνται με μέγεθος tile και συχνότητα ρολογιού όπως περιγράψαμε προηγουμένως.

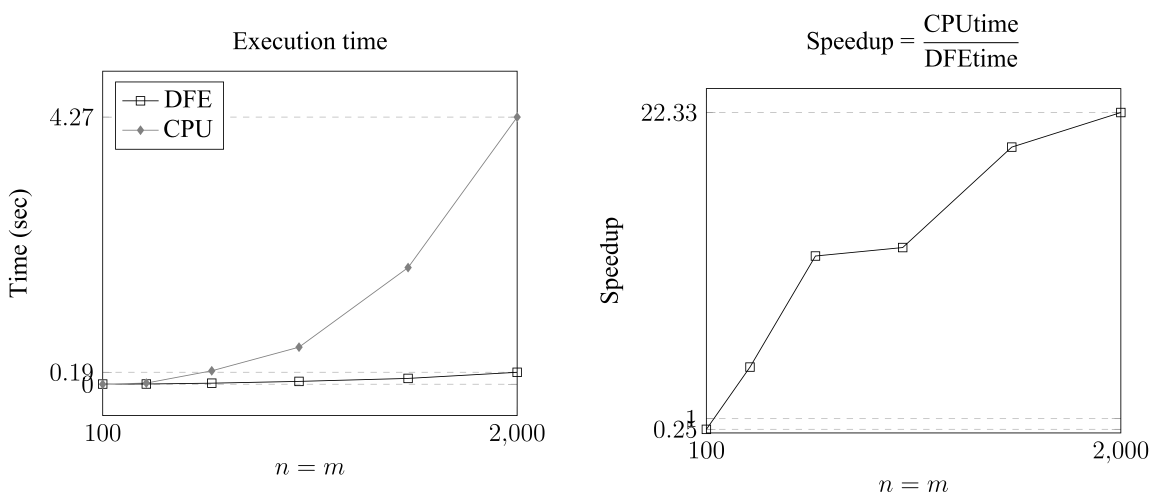


11.1.6 Solve για κάτω τριγωνικό πίνακα

scipy function (CPU):

```
X = scipy.linalg.solve(L, B)
```

όπου ο L είναι ένας κάτω τριγωνικός πίνακας μεγέθους $n \times n$ και ο B πίνακας μεγέθους $n \times m$. Χρησιμοποιείται το ίδιο maxfile με το πρόγραμμα Cholesky Solve, με μέγεθος tile 416×416 και συχνότητα ρολογιού 200 MHz.



Βιβλιογραφία

[1] Kevin P. Murphy. ”Machine Learning: a Probabilistic Perspective”, Massachusetts, The MIT Press, 2012, (pp. 1-8, 225-230, 488-493)

[2] C. E. Rasmussen & C. K. I. Williams. ”Gaussian Processes for Machine Learning”, the MIT Press, 2006, (pp. 1-48)

[3] S.N. Lophaven, H.B. Nielsen, J. Søndergaard. Aspects of the Matlab Toolbox DACE. Report IMM-REP-2002-13, Informatics and Mathematical Modelling, DTU, 2002. Available as http://www.imm.dtu.dk/∼hbn/publ/TR0213.ps

[4] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, R. C. Whaley. ”The design and implementation of the SCALAPACK LU, QR, and Cholesky factorization routines”, Department of Computer Science University of Tennessee at Knoxville, Mathematical Sciences Section Oak Ridge National Laboratory, 1994, (pp. 10-12)

[5] http://www.scikit-learn.org

[6] Maxeler Technologies. ”Multiscale Dataflow Programming”, Version 2014.2, 2015.

[7] Maxeler Technologies. ”Acceleration Tutorial Loops and Pipelining”, Version 2015.2, 2016.

[8] http://www.maxeler.com

[9] https://www.github.com/maxeler/Dense-Matrix-Multiplication

[10] https://www.github.com/maxeler/maxpower

[11] http://www.archive.ics.uci.edu/ml/