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

An Autotuning Framework for Intel Xeon Phi Platforms.

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΕΛΕΥΘΕΡΙΟΣ - ΙΟΡΔΑΝΗΣ ΧΡΙΣΤΟΦΟΡΙΔΗΣ

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

Αθήνα, Ιούλιος 2016

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

An Autotuning Framework for Intel Xeon Phi Platforms.

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΕΛΕΥΘΕΡΙΟΣ - ΙΟΡΔΑΝΗΣ ΧΡΙΣΤΟΦΟΡΙΔΗΣ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21η Ιουλίου 2016.

.....	.....	.....
Δημήτριος Σούντρης Αν. Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Κιαμάλ Πεκμεστζή Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2016

Ελευθέριος - Ιορδάνης Χριστοφορίδης

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

Περίληψη

Έχουμε πλέον εισέλθει στην εποχή όπου ο όγκος των δεδομένων προς επεξεργασία είναι ασύλληπτα μεγάλος και τα συστήματα/εφαρμογές που καλούνται να χρησιμοποιήσουν τόσο όγκο πληροφορίας αδυνατούν, όταν στηρίζονται στις παραδοσιακές μεθόδους. Γι' αυτό το λόγο, έχουν δημιουργηθεί πολυπήρηνα συστήματα, σύγχρονες αρχιτεκτονικές υπολογιστών και μέθοδοι παράλληλης επεξεργασίας με στόχο να λύσουν το παραπάνω πρόβλημα με αποδοτικό και γρήγορο τρόπο. Ωστόσο, τα νέα υπολογιστικά συστήματα και οι τρόποι επεξεργασίας διαθέτουν ένα μεγάλο βαθμό πολυπλοκότητας στη λειτουργία τους, τον οποίο και μεταφέρουν στην προσαρμογή των εφαρμογών και προγραμμάτων που χειρίζονται μεγάλους όγκους δεδομένων. Ταυτόχρονα, έχουν δημιουργηθεί όρια απόδοσης και κατανάλωσης ενέργειας τα οποία είναι απαραίτητο να τηρούνται για την εξοικονόμηση πόρων και ενέργειας. Μέχρι στιγμής, αυτά τα δύο κομβικά σημεία καλείται να εκτελέσει ο ίδιος ο προγραμματιστής. Πρέπει να αναλύσει το κάθε πρόγραμμα ανεξάρτητα και να εξετάσει τον τρόπο εκτέλεσής του μέχρι να βρει την κατάλληλη μορφή που θα τηρεί τους περιορισμούς που έχουν τεθεί. Προφανώς, αυτό το έργο αποτελεί πολύ δύσκολη δουλειά και συνήθως η προσαρμογή προγραμμάτων από ανθρώπους δεν εξαντλεί όλα τα περιθώρια βελτίωσης. Συνεπώς, καθίσταται απαραίτητη η δημιουργία ενός εργαλείου που θα αυτοματοποιεί αυτό το έργο και θα παρέχει αποδοτικότερες μορφές των προγραμμάτων σε μικρό χρονικό διάστημα.

Η παρούσα διπλωματική εργασία παρουσιάζει τον *Autotuner*, ένα άμεσο και κλιμακωτό εργαλείο που αναπτύχθηκε ειδικά για την πλατφόρμα Intel Xeon Phi coprocessor και προτείνει, για κάθε εφαρμογή που δέχεται, περιβάλλοντα διαμόρφωσης για την αποδοτικότερη εκτέλεσή τους στην πλατφόρμα. Αντικαθιστά έτσι την χειρονακτική δουλεία που έπρεπε να κάνει ο προγραμματιστής καθώς καλούνταν να εξερευνήσει 2,880 διαφορετικά περιβάλλοντα εκτέλεσης. Αντί να αναλύει κάθε εφαρμογή πάνω σ' όλα τα περιβάλλοντα εκτέλεσης, χρησιμοποιεί πληροφορίες που το εργαλείο έχει αποθηκεύσει από προηγούμενες εφαρμογές. Η λειτουργία του βασίζεται σε μια collaborative filtering μέθοδο έτσι ώστε γρήγορα και με ακρίβεια να κατηγοροποιεί μια εφαρμογή σε σύνολα περιβάλλοντων εκτέλεσης βρίσκοντας ομοιότητες με προηγούμενες εφαρμογές που έχουν βελτιστοποιηθεί.

Ο *Autotuner* ελέγθηκε πάνω σε ένα σύνολο απαιτητικών και διαφορετικών εφαρμογών από δύο σύγχρονες σουίτες και οι μετρήσεις ήταν πολύ ενθαρρυντικές. Συγκεριμένα, σε λιγότερο από 8 λεπτά για κάθε εφαρμογή ο *Autotuner* πρότεινε ένα περιβάλλον διαμόρφωσης που η απόδοσή του ξεπερνούσε το 90% της καλύτερης εκτέλεσης.

Λέξεις κλειδιά

αυτόματη προσαρμογή, αυτόματη διαμόρφωση, μηχανική μάθηση, Intel Xeon Phi επεξεργαστής, πολυπύρηνα συστήματα, συμβουλευτικό σύστημα, μοντέλο μοιραζόμενης μνήμης, παρακολούθηση, μεγάλος όγκος δεδομένων.

Abstract

We have already entered the era where the size of the data that need processing is extremely large and the applications that use them face difficulties if they follow the traditional ways. For that reason, new approaches have been developed, multi- and many- core systems, modern computing architectures and parallel processing models that aim to provide a solution for that problem efficiently and in a timely manner. However, these new computing systems kai processing methods are characterized by a lot inner complexity and that complexity is transfered also to programs' and applications' tuning which analyze big data. Concurrently, there have beed set performance and power limitations that need to comply with. Until now, these two major tasks are tackled by the application developer himself. He has to analyze every application independently and examine the its execution in order to find the version that will fulfill the restraints that have been set. Obviously, this is an onerous task and usually hand tuning does not fully exploit the margins for improvement. Hence, it is crusial the developement of a tool that will automate program tuning and will provide efficient tuned programs in a small period of time.

This diploma thesis presents, the *Autotuner*, an online and scalable tool that was developed specifically for the Inte Xeon Phi coprocessor and suggests for every incoming application, a performance effective and energy-saving tuning configuration. It substitutes the hard work the application developer had to do, as he had to explore 2,880 different tuning configurations. Instead of analyzing every application against every tuning configuration, it uses previously cached information from already checked applications. The *Autotuner* is based on a collaborative filtering technique that quickly and with accuracy classifies an application with respect to sets of tuning configurations by identifying similarities to previously optimized applications.

The *Autotuner* was tested against a set of demanding and diverse applications from two modern benchmark suites and the evaluation looked very promising. Particularly, in less than 8 minutes for every appplication the *Autotuner* suggested a tuning environment in which the application achieved more than 90% of the best tuning configuration.

Key words

automatic tuning, machine learning, Intel Xeon Phi coprocessor, manycore systems, multicore systems, recommender system, shared memory model, monitoring, big data.

Ευχαριστίες

Η παρούσα διπλωματική εργασία σηματοδοτεί την ολοκλήρωση των σπουδών μου στη σχολή των Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Ε.Μ.Π. και κλείνει ένα ταξίδι στο χώρο των υπολογιστών που αν την πρώτη μέρα φαινόταν ενδιαφέρον σήμερα φαντάζει συναρπαστικό και μοναδικό. Οι γνώσεις και οι εμπειρίες που απέκτησα με εξέλιξαν σαν άνθρωπο και με βοήθησαν να αναπτύξω τη δική μου σκέψη ως μηχανικός.

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Δημήτριο Σούντρη για την εμπιστοσύνη που μου έδειξε, για το θερμό του καλοσώρισμα όταν πρωτοπήγα στο microlab και για την συνεχή του αμέριστη υποστήριξη. Με βοήθησε να εφαρμόσω την θεωρία στην πράξη και έτσι να έχω μια πιο πρακτική προσέγγιση αλλά ταυτόχρονα σωστά θεμελιωμένη. Επίσης θα ήθελα να εκφράσω την ευγνομωσύνη μου τον Δρ. Σωτήρη Ξύδη ο οποίος μου μετέφερε την ιδέα του και μαζί την αναπτύξαμε περισσότερο - θεωρώ προς το καλύτερο - με αποτέλεσμα αυτή την εργασία. Η καθοδήγησή του καθ' όλη τη διάρκεια της διπλωματικής μου και η προτροπή του για το κάτι παραπάνω συνέβαλαν μόνο θετικά στο σύνολο αλλά και σε μένα τον ίδιο.

Τέλος, οφείλω ένα μεγάλο ευχαριστώ στους γονείς μου που ήταν και είναι πάντα δίπλα μου και με στηρίζουν, καθώς και στα κοντινά μου πρόσωπα και φίλους, παλιούς και νέους, στην Ελλάδα και στο εξωτερικό, με τους οποίους η συναναστροφή και η ανταλλαγή ιδεών διέρυνε τους ορίζοντες μου και με έκανε καλύτερο. Η φιλία τους είναι αναντικατάστατη και καθένας και καθεμία έχει σημαδέψει τον χαρακτήρα μου.

Ελευθέριος - Ιορδάνης Χριστοφορίδης,

Αθήνα, 21η Ιουλίου 2016

Εκτεταμένη Περίληψη

Εισαγωγή

Είναι γνωστό πως ύστερα από περίπου πέντε δεκαετίες, ερχόμαστε στο τέλος του νόμου του Moore. Η κατασκευή μικρότερων τρανζίστορ δεν εγγυάται πια ότι θα είναι γρήγορα, ενεργειακά αποδοτικά, έμπιστα και φθηνότερα. Ωστόσο αύτο δεν σημαίνει ότι η πρόοδος της υπολογιστικής ικανότητας έχει φτάσει στα όρια της, αλλά ότι η φύση της έχει αλλάξει. Έτσι, παράλληλα με την πιο αργή βελτίωση της απόδοσης του υλικού (hardware), το μέλλον των υπολογιστών ορίζεται κυρίως από τρεις άλλες κατηγορίες.

Η πρώτη είναι το λογισμικό. Πολλά παραδείγματα ανάμεσα στα οποία και το AlphaGo[19], έχουν αποδείξει ότι μεγάλα κέρδη στην απόδοση είναι εφικτά μέσω νέων αλγορίθμων, διατηρώντας το υλικό. Η δεύτερη είναι το ”cloud”, το δίκτυο των κέντρων δεδομένων που διαθέτουν on-line υπηρεσίες. Εφόσον μοιράζονται τους πόρους τους, οι υπολογιστές μπορούν ομαδικά, να αυξήσουν κατά μεγάλες ποσότητες τις δυνατότητές τους. Τέλος, η τρίτη κατηγορία βρίσκεται στις νέες αρχιτεκτονικές υπολογιστών. Αξιοσημείωτα παραδείγματα είναι οι πολυπύρηνοι επεξεργαστές και οι επιταχυντές (GPGPUs, FPGAs). Αυτές οι αρχιτεκτονικές υποστηρίζουν επίσης τον παραλληλισμό εκτέλεσης εντολών και συντάσουν την High Performance Computing (HPC) area.

Σχετικά με την τελευταία κατηγορία που αναφέραμε, παρατηρούμε ότι τα παράλληλα υπολογιστικά συστήματα γίνονται συνεχώς πιο πολύπλοκα. Οι αιτίες βρίσκονται κυρίως στην εκθετική αύξηση των δεδομένων προς επεξεργασία και στις πιο απαιτητικές εφαρμογές (π.χ. επιστήμες, βελτιστοποίηση, προσομοιώσεις). Τα συστήματα του παρελθόντος είναι ανίκανα να επεξεργαστούν τόσα δεδομένα στο χρόνο που ο κόσμος πλέον αναμένει. Γι’ αυτούς τους λόγους, τα HPC συστήματα με μέγιστη απόδοση πολλών τετράκις εκατομμυρίων πράξεων κινητής υποδιαστολής, έχουν εκατοντάδες χιλιάδες πυρήνες οι οποίοι πρέπει να είναι ικανοί να δουλεύουν αποδοτικά ταυτόχρονα και να μοιράζονται έξυπνα τους υπολογιστικούς τους πόρους. Χρησιμοποιούν επίσης επιταχυντές υλικού με συγκεκριμένη λειτουργία και εξελιγμένες μεθόδους στην αποθήκευση δεδομένων, ισορροπία φορτίου και στην ενδοεπικοινωνία. Δυστυχώς, οι εφαρμογές όπως είχαν σχεδιαστεί για να εκτελούνται, δεν αποδίδουν το μεγίστο με αυτά τα συστήματα. Συνεπώς, υψηλές ποσότητες ενέργειας και χρημάτων χάνονται εξαιτίας της χαμηλής διεκπεραιωτικής ικανότητας κάθε επεξεργαστή. Για να αλλάξη αυτή η κατάσταση, οι προγραμματιστές πρέπει να μελετήσουν ατομικά την αρχιτεκτονική κάθε συστήματος και να προσαρμόσουν το προγραμμά τους με τέτοιο τρόπο ώστε να εκμεταλλέυονται κάθε μονάδα υλικού βέλτιστα. Μόνο τότε η εφαρμογή θα φτάνει τη μέγιστη δυνατή ρυθμαπόδοση του συστήματος.

Πέρα από την καθαρή απόδοση, είναι αναγκαία και η μείωση της κατανάλωσης ενέργειας αυτών των συστημάτων. Η ενεργειακή κρίση έχει θέσει όρια στις ποσότητες ενέργειας που κάθε σύστημα καναλώνει. Προς αυτό τον στόχο, ο προγραμματιστής εφαρμόζει προχωρημένες τεχνικές τόσο υλικού όσο και λογισμικού που δημιουργούν μια κατάσταση που πρέπει να φτάσει σε ισορροπία μεταξύ απόδοσης και εξοικονόμησης ενέργειας, με αποτέλεσμα να αυξάνεται επιπλέον η διαδικασία προσαρμογής. Συγκεντρώνοντας όλα τις επιλογές και τις διαμορφώσεις που ένας προγραμματιστής έχει να ζυγίσει για να προσαρμόσει κατάλληλα την εφαρμογή του, ένα ομογενές σύστημα μεταμορφώνεται τελικά σε ένα ετερογενές, το οποίο περιπλέκει περισσότερο την κατάσταση.

Επιπλέον, οι προγραμματιστές αφιερώνουν σημαντικό χρόνο για να προσαρμόσουν την εφαρμογή τους σε συγκεκριμένα συστήματα. Αυτή είναι μια κυκλική διαδικασία συλλογής δεδομένων, αννγνώρισης περιοχών κώδικα που μπορούν να βελτιωθούν και πάλι προσαρμογή αυτών των περιοχών [24]. Επομένως, αυτή η διαδικασία είναι κοπιαστική, χρονοβόρα και μερικές φορές πρακτικά αδύνατη για να γίνει χειροκίνητα.

Ο στόχος είναι να αυτοματοποιηθεί αυτή η διαδικασία με καλύτερα αποτελέσματα απ' αυτά της συντηρητικής μεθόδου. Για να πετύχει αυτό η ερευνητική κοινότητα έχει διαιρέσει την διαδικασία προσαρμογής στα εργαλεία ανάλυσης απόδοσης και στα εργαλεία αυτόματης προσαρμογής. Και ο ακαδημαϊκός και ο εμπορικός κόσμος, έχουν αναπτύξει εργαλεία που υποστηρίζουν και εν μέρει αυτοματοποιούν την αναγνώριση και την παρακολούθηση εφαρμογών [13, 43]. Επίσης, ειδική αναφορά λαμβάνει το Roofline model [65] το οποίο αποκαλύπτει τα όρια απόδοσης μιας αρχιτεκτονικής μαζί με τη θέση μιας εφαρμογής που εξετάζεται, δίνοντας μια πιο πρακτική οπτική στη διαδικασία της προσαρμογής.

Τα εργαλεία, μέχρι τώρα, δεν παράγουν αυτόματα ένα βέλτιστο εκτελέσιμο, αντιθέτως περιορίζουν την λειτουργία τους στον χαρακτηρισμό περιοχών κώδικα και σε απλές συμβουλές προς τον προγραμματιστή, ο οποίος χειροκίνητα κάνει κάθε αλλαγή στον κώδικα και προχωράει με δοκίμες και λάθη. Συνεπώς, τα τελευταία χρόνια πολύ έρευνα έχει γίνει πάνω στην αυτοματοποίηση της διαδικασίας προσαρμογής. Στρατηγικές που έχουν χρησιμοποιηθεί, έχουν σαν κρίσιμο σημείο την αυτόματη και αποδοτική αναζήτηση του καλύτερου συνδυασμού παραμέτρων του περιβάλλοντος εκτέλεσης για κάθε εφαρμογή σε κάθε αρχιτεκτονική. Οι ιδέες που έχουν ανακαλυφθεί στον τομέα της αυτόματης προσαρμογής έχουν δώσει ενθαρρυντικά αποτελέσματα και έχουν οφελήσει τους προγραμματιστές [52, 60, 38].

Ένα πρωτοπόρο παράδειγμα πολυπύρηνων συστημάτων που χρησιμοποιείται σήμερα είναι η Intel Many Integrated Core Architecture (Intel MIC)[4], ένας συνεπεξεργαστής που αναπτύχθηκε από την Intel. Το πρωτότυπο κυκλοφόρησε το 2010 με το κωδικό όνομα *Knights Ferry*. Ένα χρόνο αργότερα το *Knights Corner* ανακοινώθηκε και από τον Ιούνιο του 2013 ο συνεπεξεργαστής είναι στην δεύτερη γενιά του *Knights Landing*. Πολύ σύντομα, μαζί με τα Intel Xeon processors-based συστήματα, έγιναν τα κύρια συστατικά των υπερ-υπολογιστών. Αυτή τη στιγμή, ο Tianhe-2(MilkyWay-2) ο υπερ-υπολογιστής στο Εθνικό Κέντρο υπερ-υπολογιστών Guangzhou, κατέχει την πρώτη θέση στη περίφημη λίστα Top500 list[15] και περιλαμβάνει 32,000 Intel Xeon E5-2692 12C στα 2.200 GHz and 48,000 Xeon Phi 31S1P, φτάνοντας τα 33,862.7 TFLOP/S. Ωστόσο, πολλές εφαρμογές δεν έχουν προσαρμοσθεί ακόμα για να εκμεταλλεύονται το μέγιστο μέγεθος παραλληλισμού, τους υψηλούς ρυθμούς ενδοεπικοινωνίας και μεταφοράς δεδομένων καθώς επίσης και τις διανυσματικές δυνατότητες του Intel Xeon Phi. Για να φτάσουμε υψηλές αποδόσεις μ' αυτή την πλατφόρμα χρειαζόμαστε πολύ προσπάθεια στην παραλληλοποίηση, στην ανάλυση και στην βελτιστοποίηση στρατηγικών [33].

Άρα, αναζητώντας και αξιολογώντας την διαδικασία προσαρμογής σε ένα σύστημα τελευταίας τεχνολογίας όπως ο Intel Xeon Phi, είναι σίγουρα μια πολύτιμη και ευοίωνη συνεισφορά στους τομείς του HPC και της αυτοματοποιημένης προσαρμογής.

Συνεισφορά

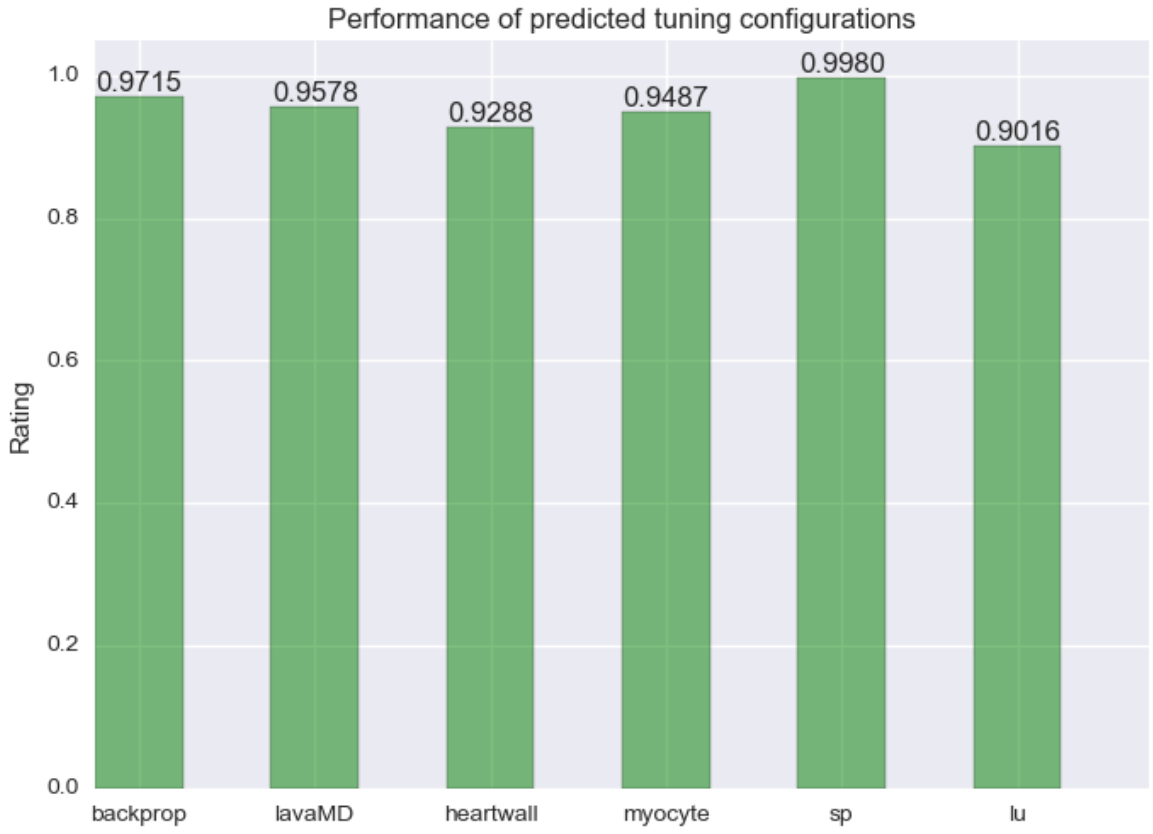
Σ' αυτή την διπλωματική, αναπτύξαμε ένα αυτοματοποιημένο εργαλείο για τον Intel Xeon Phi συνεπεξεργαστή βασισμένο σε αναλυτικές μεθόδους. Ο στόχος του είναι να απαλάξει τον προγραμματιστή από την χειροκίνητη προσαρμογή του μεταγλωτιστή και του περιβάλλοντος εκτέλεσης βρίσκοντας αποδοτικά και βέλτιστα τη λύση με το καλύτερο αποτέλεσμα λαμβάνοντας υπ' όψη και την καθαρή απόδοση και την ενέργεια.

Συνοπτικά, ο *Autotuner* έχει μια offline βάση δεδομένων με δεδομένα απόδοσης από ένα σύνολο διαφορετικών εφαρμογών εκτελεσμένων σε ένα σύνολο από παραμέτρους. Αυτά τα δεδομένα συλλέχθη-

σαν χρησιμοποιώντας το LIKWID[62], ένα ελαφρύ εργαλείο για x86 πολυπύρηνα περιβάλλοντα. Το εργαλείο μας χρησιμοποιεί αυτά τα δεδομένα για να βρει συσχετισμούς μεταξύ εφαρμογών και παραμέτρων προσαρμογής που εξετάζονται. Για να το πετύχει αυτό βασίζεται σε μια τεχνική collaborative filtering technique[48, 35] και στην ιδέα της Singular Value Decomposition (SVD)[50]. Έτσι, οι εφαρμογές και οι παράμετροι προσαρμογής προβάλλονται στο χώρο χαρακτηριστικών. Αυτός είναι ένας σύνολο γνωρισμάτων τα οποία αποτελούνται από τιμές παραμέτρων και τη βαθμωτή σχέση των εφαρμογών και των παραμέτρων μ' αυτά τα γνωρίσματα. Στη συνέχεια, κάθε εφαρμογή που καταφθάνει, εκτελείται για μερικά διανύσματα παραμέτρων και προβάλλεται στον κατασκευασμένο χώρο χαρακτηριστικών με βάση τις δικές της βαθμολογίες. Οι συσχετίσεις με κάθε χαρακτηριστικό παράγονται με αποτέλεσμα, οι άγνωστες βαθμολογίες να μπορούν να υπολογιστούν. Στο τέλος, έχουμε ένα γεμάτο διάνυσμα με τις προγνώσεις για όλες τις παραμέτρους, από το οποίο μπορούμε να διαλέξουμε την καλύτερη που αντιστοιχεί σε συγκεκριμένο διάνυσμα παραμέτρων.

Επίσης, το εργαλείο που αναπτύξαμε στηρίζει την εφαρμογή της μηχανικής μάθησης και των δυνατοτήτων της στο πεδίο της αυτόματης προσαρμογής και συμβάλλει σημαντικά σ' αυτό. Πέρα από τις γρήγορες προγνώσεις και την καλή απόδοση, η μέθοδος singular value decomposition μειώνει επίσης το χώρο που χρειάζεται για τον χαρακτηρισμό των εφαρμογών έναντι των παραμέτρων, και έτσι αποθηκεύει τεράστια πληροφορία σε μικρό χώρο.

Μια ματιά στα αποτελέσματα μας λέει ότι, ο *Autotuner* καταφέρνει να δίνει συνεχώς ένα διάνυσμα προσαρμογής που καταλαμβάνει περισσότερο του 90% της εκτέλεσης που αντιστοιχεί στην καλύτερη προσαρμογή. Επίσης, αυτό συμβαίνει σε λιγότερο από 8 λεπτά, που είναι ο χρόνος για τον μερικό χαρακτηρισμό της εφαρμογής. Η εικόνα 0.1 δείχνει την απόδοση που επιτυγχάνει από τις προγνώσεις για 6 εφαρμογές.



Σχήμα 0.1: Απόδοση από της προβλέψεις σε χρόνο λιγότερο από 8 λεπτά μερικού χαρακτηρισμού.

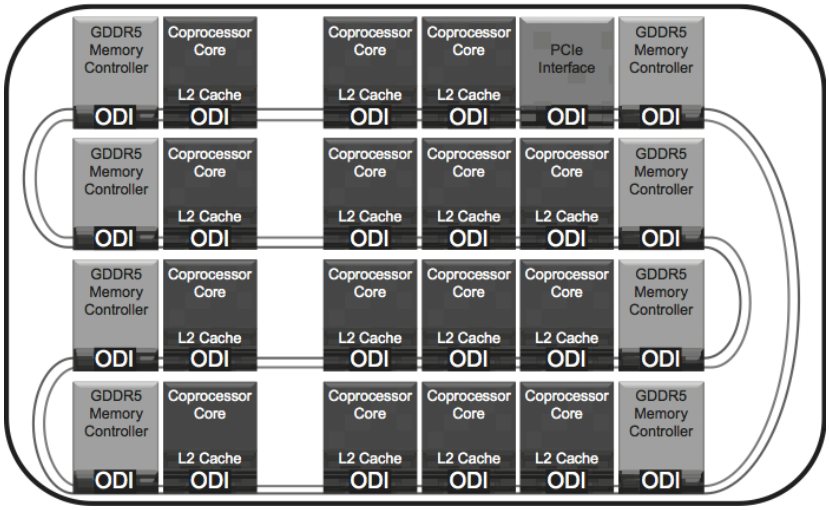
Πειραματική Πλατφόρμα και Περιβάλλον

Αρχιτεκτονική

Σ' αυτή την εργασία χρησιμοποιήσαμε τον συνεπεξεργαστή της Intel, Intel®Xeon Phi™της σειράς 3100 με κωδικό όνομα *Knights Corner*. Τα χαρακτηριστικά του φαίνονται παρακάτω:

- 22nm μέγεθος επεξεργαστή
- Intel Many Integrated Core (MIC) αρχιτεκτονική
- Πολυεπεξεργαστής μοιραζόμενης Μνήμης και τρέχει Λίνουξ
- 57, in-order, dual issue, x86 πυρήνες στα 1.1GHz με 4 νήματα υλικού ο καθένας
- 6 GB GDDR5 κύρια μνήμη στα 240 GB/δευτερόλεπτο
- 32KB L1 (εντολών & δεδομένων) kai 512 KB L2 για κάθε φυσικό πυρήνα.

Όλοι η πυρήνες και οι επιτηρητές μνήμης συνδέονται πάνω σε ένα αμφίδρομο δακτυλίδι (ODI) όπως φαίνεται στο Σχήμα 0.2.



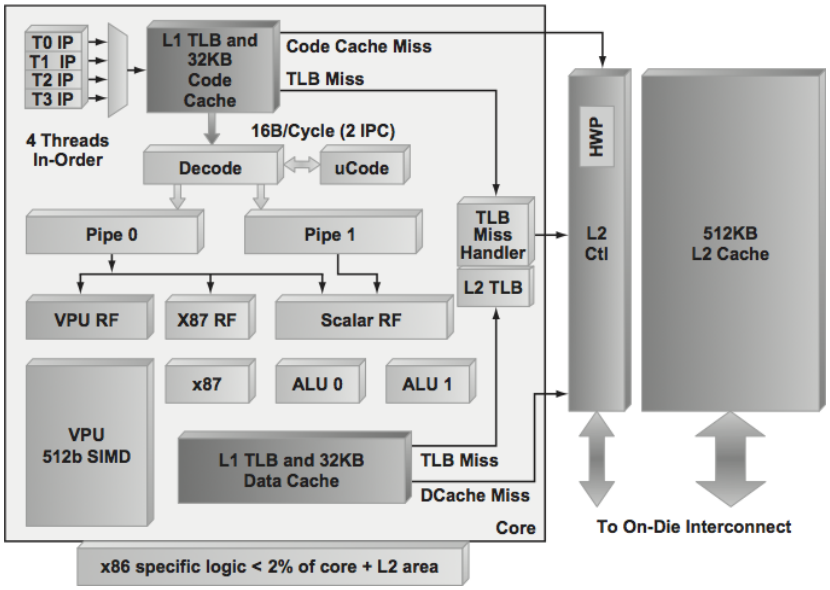
Σχήμα 0.2: Ο συνεπεξεργαστής σε επίπεδο πυριτίου[46].

Επίσης το μικροαρχιτεκτονικό διάγραμμα του επεξεργαστή φαίνεται στο παρακάτω Σχήμα 0.3

Παράμετροι Προσαρμογής

Οι παράμετροι προσαρμογής που χρησιμοποιήθηκαν για την δημιουργία του χώρου αναζήτησης πάρθηκαν από επιλογές κατά την μεταγώττιση αλλά και από παραμέτρους εκτέλεσης. Στον παρακάτω πίνακα 0.1 φαίνονται συγκετρωτικά:

Συνολικά, αποτελούν 2,880 συνδυασμούς.



Σχήμα 0.3: Μικρο-αρχιτεκτονική ενός φυσικού πυρήνα του συνεπεξεργαστή[46].

Flag	Arguments
-O[=n]	n=2,3
-opt-prefetch[=n]	n=0,2,3,4
-opt-streaming-stores [keyword]	keyword=never,always
-opt-streaming-cache-evict[=n]	m=0,1,2,3
-unroll	enabled/disabled
huge pages	enabled/disabled
affinity [type]	type=scatter,balanced
cores	19,38,57
threads per core	2,3,4

Πίνακας 0.1: Παράμετροι προσαρμογής.

Εφαρμογές Αναφοράς

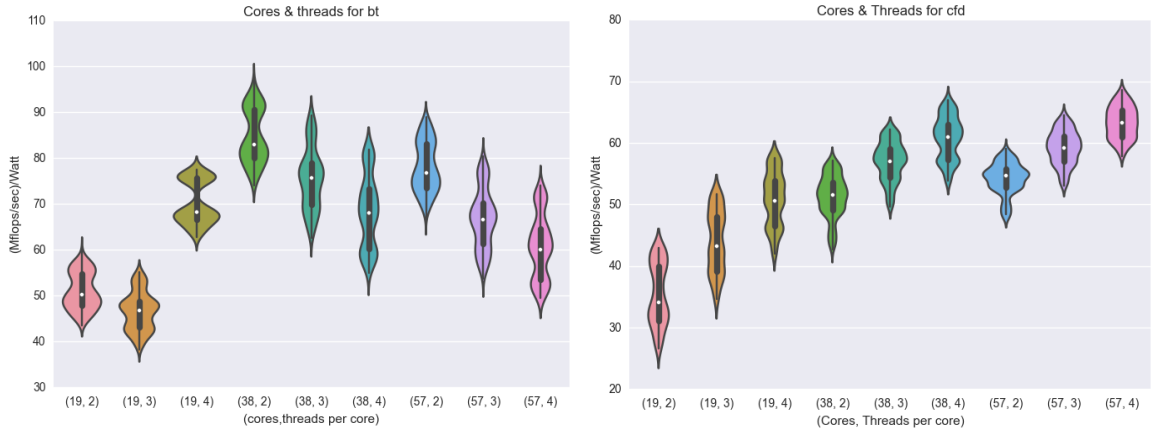
Οι εφαρμογές που επιλέχθησαν για τον έλεγχο και την αξιολόγηση του συστήματος παρατηρούνται σε πολλές σημερινές πειραματικές εφαρμογές των ημερών μας. Προέρχονται από τις σουίτες *Rodinia* και *NAS Parallel Benchmarks*.

Παρακάτω φαίνονται τα χαρακτηριστικά τους συγκεντρωτικά:

Χαρακτηρισμός των Εφαρμογών ως προς το Χώρο Αναζήτησης

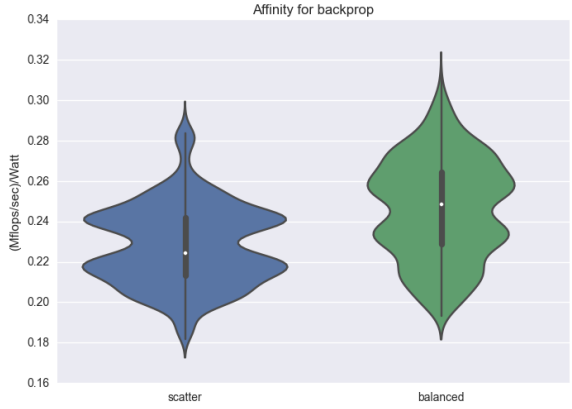
Προκειμένου να δείξουμε ότι οι εφαρμογές μας επηρεάζονται από την διακύμανση του χώρου αναζήτησης των παραμέτρψν προσαρμογής, τις αξιολογούμε ως προς *MFlops/δευτ.* και βλέπουμε πως κάθε μια παράμετρος επηρεάζει την κατανομή της απόδοσής της.

Παρατηρούμε ότι, οι εφαρμογές έχουν μεγάλη διακύμανση στην απόδοσή τους ως προς κάθε παράμετρο. Συνεπώς, ο χώρος αναζήτησης είναι ικανοποιητικός και πλήρης, δηλαδή κάθε μια παράμετρος μπορεί να επηρεάζει αρνητικά ή θετικά την εκτέλεση μιας εφαρμογής.

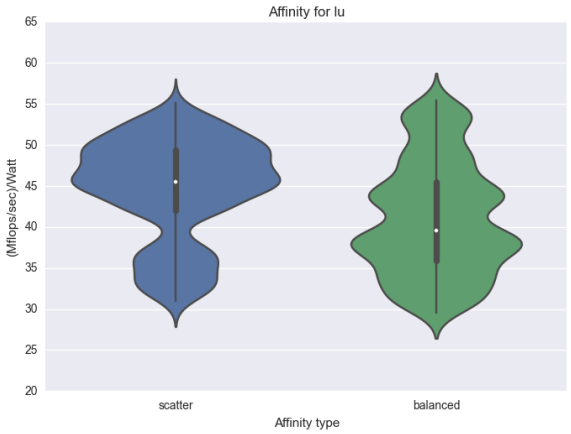


(α□)

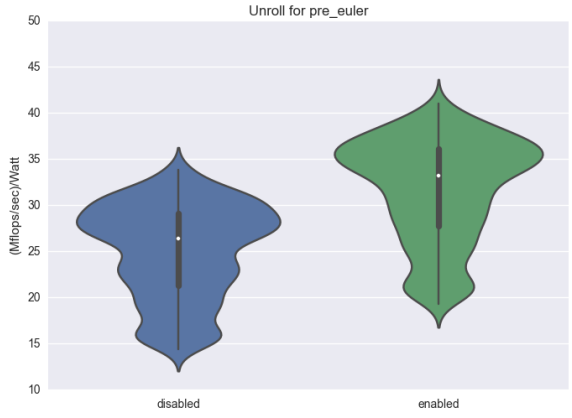
(β□)



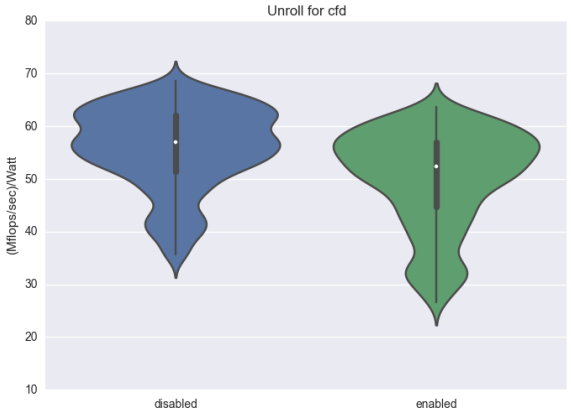
(γ□)



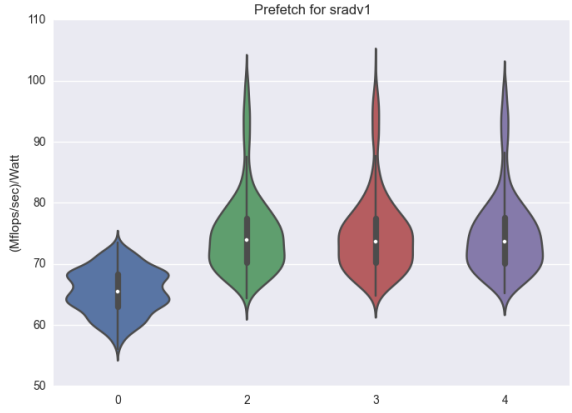
(δ□)



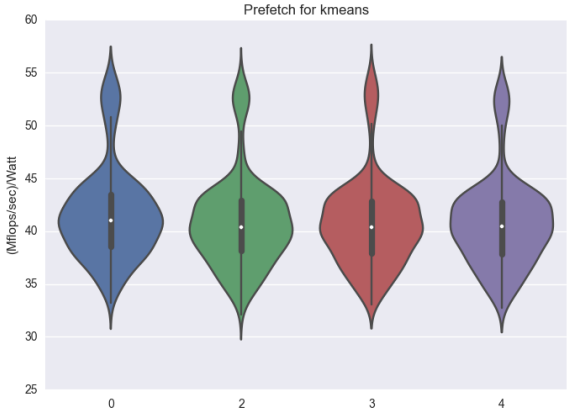
(ε□)



(στ□)



(ζ□)



(η□)

Σχήμα 0.4: Διαγράμματα βιολιά για τις παραμέτρους τους χώρου προσαρμογής.

Application	Domain	MFlops
LUD	Linear Algebra	350,950.0
Hotspot	Physics Simulation	3,144.5
Hotspot3D	Physics Simulation	3,770.0
Streamcluster	Data Mining	1,716.0
K-means	Data Mining	63,492.0
LavaMD	Molecular Dynamics	14,720.0
Heartwall	Medical Imaging	175.9
Myocyte	Biological Simulation	2331.2
srad_v1	Image Processing	103,462.0
srad_v2	Image Processing	151,200.0
Back Propagation	Pattern Recognition	469.8
NN	Data Mining	182.4
CFD	Fluid Dynamics	157,347.4
pre-CFD	Fluid Dynamics	168,371.0

Πίνακας 0.2: Εφαρμογές Αναφοράς από τη σουίτα Rodinia.

Benchmark	Class	MFlops
BT	A	168,300.0
SP	A	85,000.0
LU	A	119,280.0
FT	B	92,050.0
MG	C	155,700.0
CG	B	54,700.0

Πίνακας 0.3: Εφαρμογές Αναφοράς από τη σουίτα NAS Parallel Benchmarks.

Ο Autotuner: Υπόβαθρο & Υλοποίηση

Ο *Autotuner* έχει δύο στάδια, το offline και το online, τα οποία και επεξηγούμε παρακάτω.

Offline Στάδιο

Κατά το offline στάδιο χτίζεται η βάση μάθησης που θα χρησιμοποιηθεί. Οι εφαρμογές που θα την αποτελούν τρέχουν στον Intel Xeon Phi για κάθε δυνατό διάνυσμα παραμέτρων προσαρμογής. Από την εκτέλεση αυτή, με τη χρήση του Likwid εργαλείου, συλλέγουμε τις τιμές των μετρητών απόδοσης από τους οποίους κατασκευάζουμε μετρικές. Αυτές είναι: IPC, MFlops/sec, Bandwidth, Time, Vectorization, Power. Με αυτές λοιπόν συντάσουμε ένα csv αρχείο για κάθε εφαρμογή. Το σύνολο όλων των csv αρχείων αποτελεί τη βάση μάθησής μας.

Online Στάδιο

Κατά το online στάδιο γίνεται η πρόβλεψη του βέλτιστου διανύσματος προσαρμογής για κάθε εισερχόμενη εφαρμογή. Όπως και στο offline στάδιο, η εφαρμογή που έρχεται εκτελείται στον συνεπεξεργαστή και χαρακτηρίζεται άλλα για πολύ λιγότερα διανύσματα παραμέτρων προσαρμογής από το σύνολο, τάξη 1%. Στη συνέχεια μαζί με την βάση μάθησης από το offline στάδιο, μέσω μιας κρίσιμης

συνάστησης που ορίζει το σχέδιο βαθμολογία μας, φτιάχνουν το σύνολο εκπαίδευσης του μοντέλου για την πρόβλεψη των βαθμολογιών.

Το μοντέλο αυτό ανήκει στην οικογένεια των συστημάτων σύστησης (recommendation systems) και συγκεκριμένα στην κατηγορία collaborative filtering (συνεργατικό φιλτράρισμα). Το μοντέλο αυτό προβάλλει τους χρήστες-εφαρμογές και τα αντικείμενα-διανύσματα προσαρμογής στον ίδιο χώρο όπου μπορούν να συγκριθούν. Ο χώρος αυτός ονομάζεται latent factors.

Η μορφή του μοντέλου μας είναι:

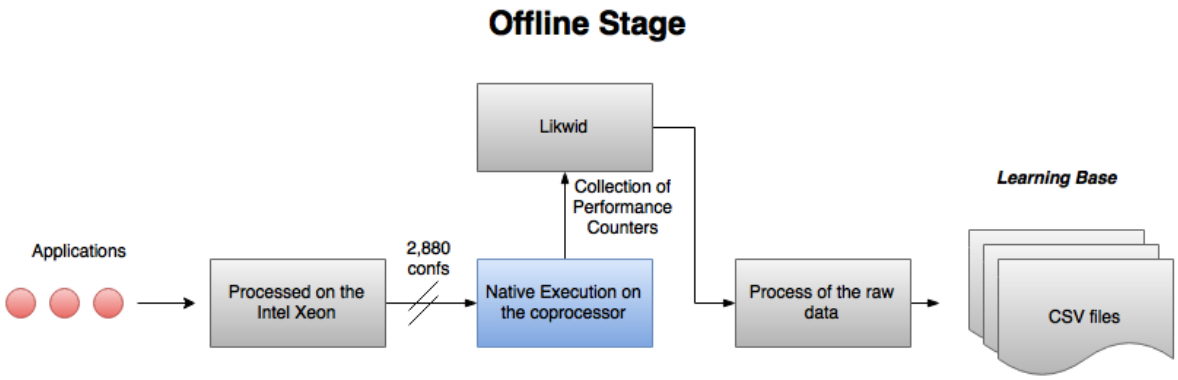
$$\hat{r}_{ui} = \mu + b_u + b_i + p_u q_i^T$$

Το μοντέλο προσαρμόζεται μηδενίζοντας το τετραγωνικό λάθος με κάθε παρατήρηση που του δίνουμε:

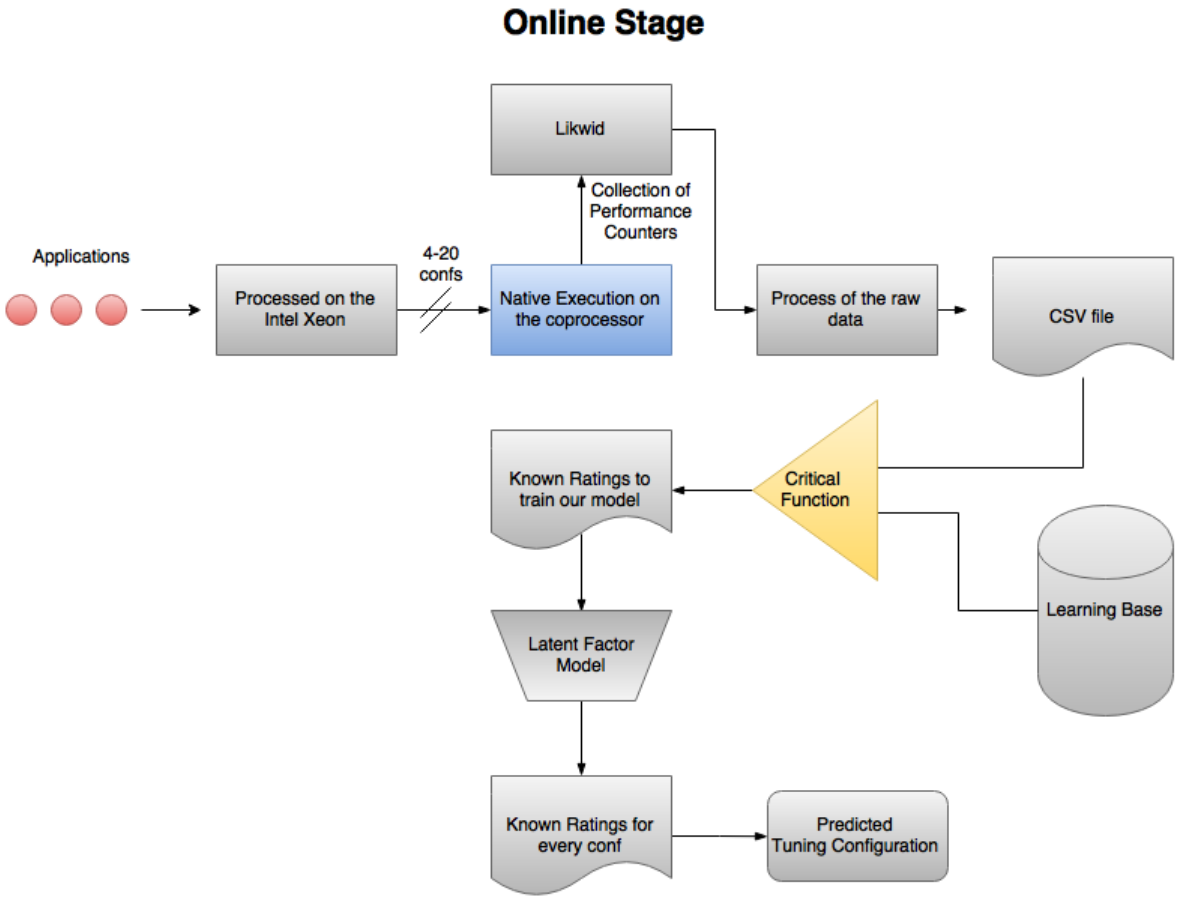
$$\min_{p^*,q^*,b^*} L = \sum_{u,i \in \mathcal{K}} (r_{ui} - \mu - b_u - b_i - p_u q_i^T)^2 + \lambda_1 \sum_u \|p_u\|^2 + \lambda_2 \sum_i \|q_i\|^2 + \lambda_3 \sum_u b_u^2 + \lambda_4 \sum_i b_i^2$$

Η παραπάνω συνάρτηση μηδενίζεται με την χρήση της μεθόδου *Stochastic Gradient Descent* (προσέγγιση με στοχαστική κλίση) και δεν έχει αναλυτική λύση.

Γραφικά, τα δύο στάδια φαίνονται στα παρακάτω σχήματα:



Σχήμα 0.5: Offline στάδιο.

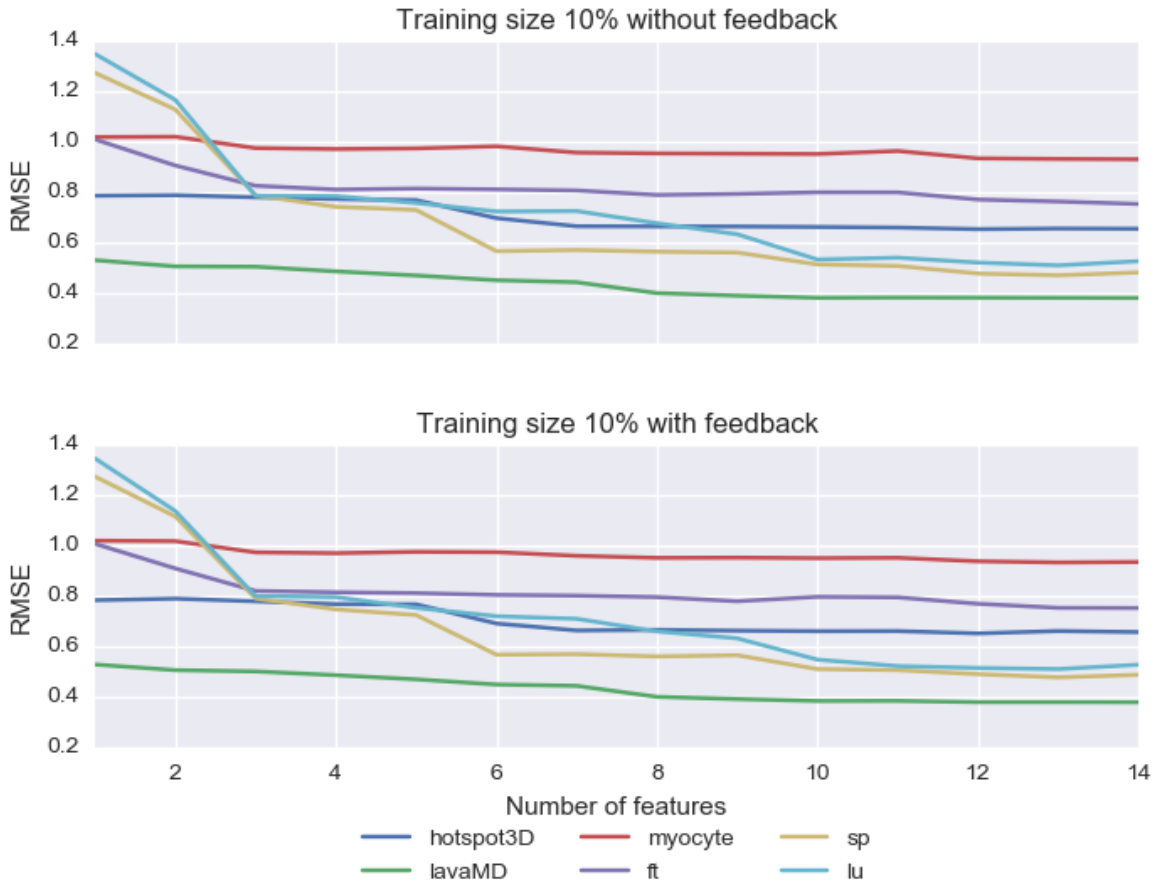


Σχήμα 0.6: Online στάδιο.

Αποτελέσματα

Παρακάτω παρουσιάζουμε τα αποτελέσματα για την κρίσιμη συνάρτηση *MFlops/sec/Watt*. Για το λάθος της πρόβλεψης χρησιμοποιούμε το ριζικό μέσο τετραγωνικό λαθος (RMSE).

Αρχικά πρέπει να βρούμε το μέγεθος του latent factor χώρου ο οποίος χαρακτηρίζει επαρκώς της εφαρμογές και τα διανύσματα προσαρμογής. Στο Σχήμα φαίνεται ότι ο μέγεθος είναι 12, καθώς από εκεί και έπειτα δεν έχουμε κάποια μείωσει του RMSE.



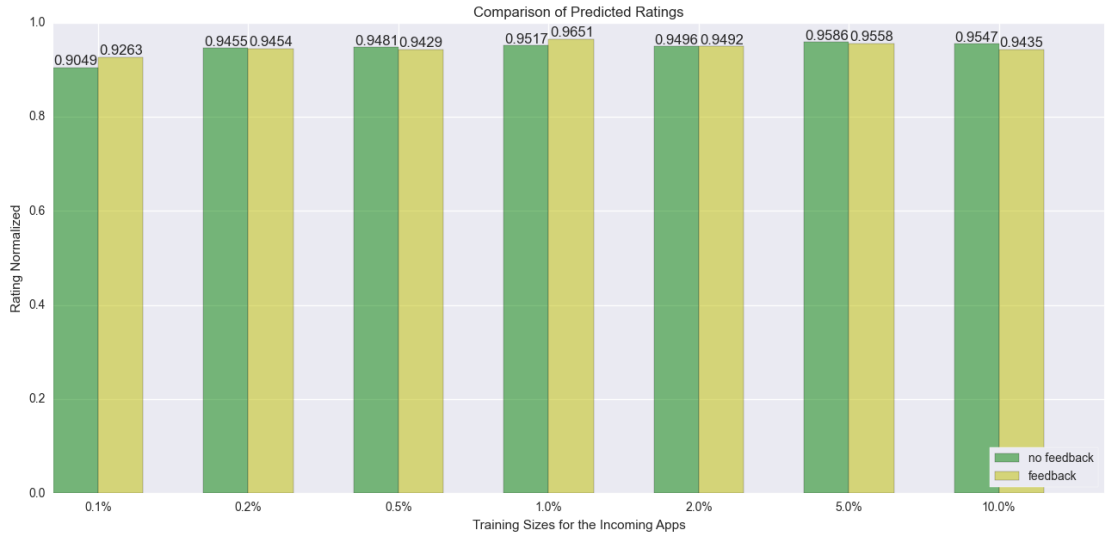
Σχήμα 0.7: RMSE για μεταβλητό αριθμό χαρακτηριστικών με ή χωρις επανατροφοδότηση.

Στη συνέχεια πρέπει να δούμε πως συμπεριφέρεται το μοντέλο ανάλογα με το μέγεθος του μερικού χαρακτηρισμού κάθε εισερχόμενης εφαρμογής. Έτσι έχουμε το παρακάτω σχήμα:

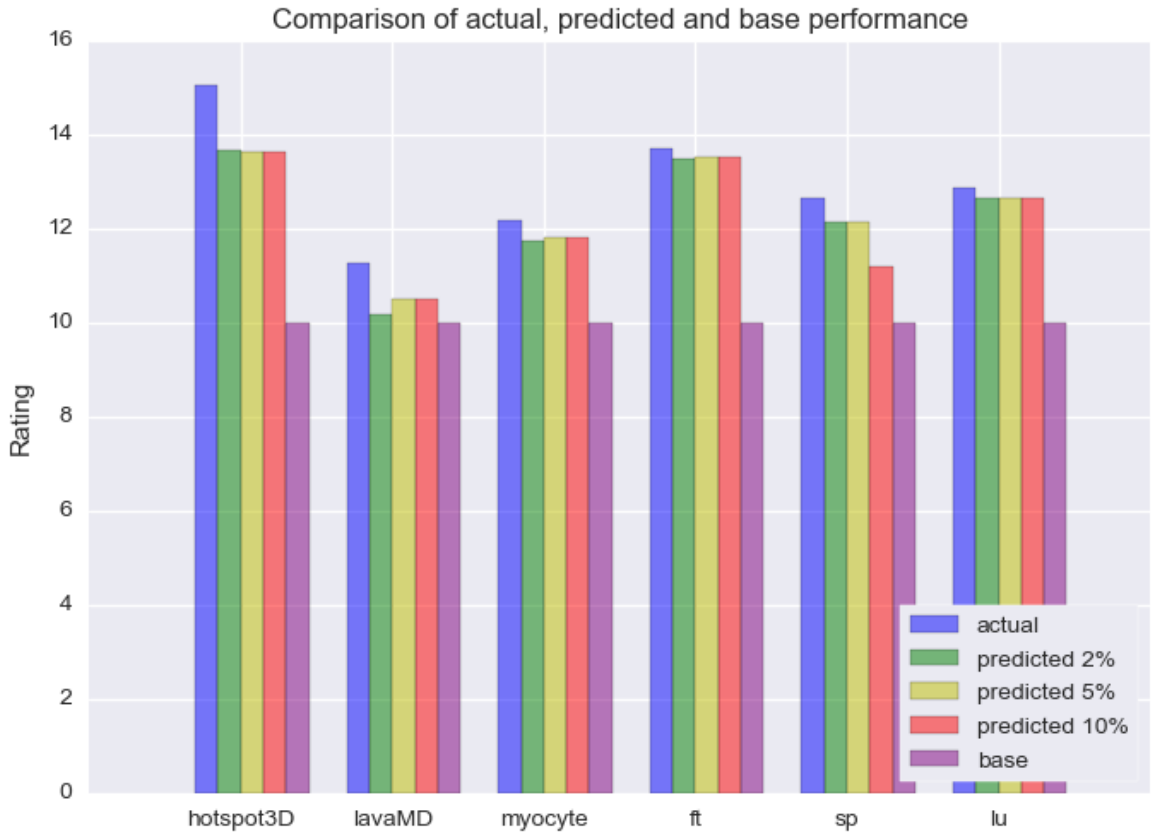
Παρατηρούμε ότι για κάθε μέγεθος έχουμε παραπάνω από 90% απόδοση ως πρός την καλύτερη εκτέλεση. Πιο συγκεκριμένα μπορούμε να δούμε τις αποδόσεις συγκριτικά με την καλύτερη εκτέλεση και την βασική.

Συζήτηση

Ο *Autotuner* αποδεικνύεται ότι είναι αποτελεσματικός σε πολλά περιβάλλοντα. Σε ποικίλες βάσεις από τις οποίες μαθαίνει, σε διαφορετικές συναρτήσεις βαθμολογίας και μέγεθος μάθησης. Πάντα όμως θα πρέπει να είμαστε προσεκτικοί με την ανάθεση τιμών στη στοχαστική προσέγγιση (SGD), δηλαδή στους παράγοντες μάθησης και ρύθμισης ώστε να αποφεύγεται το overfitting των δεδομένων



Σχήμα 0.8: Μέσο RMSE για 12 χαρακτηριστικά και μεταβλητό μέγεθος εκπαίδευσης με ή χωρις επανατροφοδότηση.



Σχήμα 0.9: Σύγκριση απόδοσης για 12 χαρακτηριστικά, 2%, 5% and 10% γνωστές βαθμολογίες, με ή χωρις επανατροφοδότηση.

μάθησης και ταυτόχρονα να συγκλίνει σε αποδεκτές τιμές. Τα προβλεπόμενα διανύσματα παραμέτρων ξεπερνούν την απόδοση της βασικής ρύθμισης των εφαρμογών και δείχνουν λεπτομερή προσαρμογή για κάθε εφαρμογή. Αυτό είναι πολύ σημαντικό χαρακτηριστικό καθώς είναι πολύ δύσκολο για έναν προγραμματιστή να προσαρμώσει με λεπτομέρεια μια εφαρμογή. Χρειάζεται να εκτελέσει

ανάλυση απόδοσης για κάθε μορφή του προγράμματος ώστε να βρει τα κρίσιμα σημεία στον κώδικα που χρειάζονται βελτίωση. Αυξάνει την απόδοση σταδιακά και όχι άμεσα.

Επίσης, ο *Autotuner* είναι σχετικά γρήγορος μόλις γίνει online. Το offline μέρος του είναι το πιο χρονοβόρο αλλά το πλεονέκτημα έγκειται ότι χρειάζεται να γίνει μόνο μια φορά. Αν η βάση μάθησης περιλαμβάνει 14 εφαρμογές, η καθεμία με μέσο χρόνο εκτέλεσης 15 δευτερόλεπτα, τότε για 2,880 διανύσματα παραμέτρων και 4 εκτελέσεις το καθένα παίρνουμε ένα άθροισμα 2,419,200 δευτερολέπτων ή 672 ωρών. Από εκεί και έπειτα, για κάθε εισερχόμενη εφαρμογή χρειαζόμαστε μερικό χαρακτηρισμό πάνω στο 0.1%-1% του συνόλου των διανυσμάτων παραμέτρων. Αυτό το βήμα χρειάζεται 4-40 λεπτά το πολύ και ο *Autotuner* επιστρέφει σε λιγότερο από 30 δευτερόλεπτα την πρόγνωση για το διάνυσμα παραμέτρων. Ανάλογα με τα επίπεδα απόδοσης που ψάχνουμε, το μέγεθος μάθησης ποικίλει και συνεπώς και ο χρόνος που χρειάζεται το online τμήμα. Πάντα ωστόσο, ξεπερνάμε το 90% της εκτέλεσης που αντιστοιχεί στο καλύτερο διάνυσμα παραμέτρων. Επίσης, μπορούμε να ζητήσουμε και την προβλεπόμενη απόδοση για οποιαδήποτε διάνυσμα παραμέτρων.

Τέλος, αν χρησιμοποιήσουμε επανατροφοδότηση περιμένουμε να πάρουμε βελτιωμένες προβλέψεις καθώς το μοντέλο μαθαίνει περισσότερο από κάθε εισερχόμενη εφαρμογή. Στα πειράματά μας, δεν προσέξαμε κάποια μεγάλη πρόοδο καθώς το πλήθος των εφαρμογών που ελέγξαμε ήταν μικρό και η επίδραση δεν πρόλαβε να διαδοθεί. Θεωρούμε όμως ότι είναι ένα λογικό επιχείρημα αυτό και θα πρέπει να θεωρείται αποδεκτό.

Ο πίνακας 0.4 δείχνει τα μέσα ποσοστά απόδοσης των προβλέψεων που ελέγθησαν στα πειράματά μας.

Training size	Best Rating	Base Rating
0.1%	90.06%	131.42%
0.2%	93.39%	136.28%
0.5%	95.06%	138.71%
1%	94.53%	137.94%
2%	95.43%	139.25%
5%	95.47%	139.31%
10%	95.85%	139.87%

Πίνακας 0.4: Μέσα ποσοστά απόδοσης των προβλεπόμενων διανυσμάτων παραμέτρων ως προς την βέλτιστη και την βασική βαθμολογία.

Για να συγκρίνουμε με άλλα εργαλεία, ο *Autotuner* παρουσιάζει πολλά πλεονεκτήματα. Αρχικά, οι οδηγοί για τη χειροκίνητη προσαρμογή δεν είναι ικανοί να οδηγήσουν στη βέλτιστη διαμόρφωση για κάθε συγκεκριμένη εφαρμογή. Προτείνουν γενικές αλλαγές και περιβάλλοντα εκτέλεσης για την πλειοψηφία των εφαρμογών. Δεύτερον, ο *Autotuner* είναι ικανός να επιστρέψει το καλύτερο προβλεπόμενο διάνυσμα παραμέτρων ανάμεσα σε ένα πολύ μεγάλο χώρο αναζήτησης σε σύντομο χρονικό διάστημα, ξεπερνώντας σε απόδοση τα επαναληπτικά εργαλεία αναζήτησης. Αν και τα τελεταία εγγυόνται ότι θα επιστρέψουν το καλύτερο διάνυσμα, ο *Autotuner* επιστρέφει διανύσματα που φτάνουν σε μεγαλύτερα επίπεδα της 90% της απόδοσης του καλύτερου διανύσματος προσαρμογής.

Ως εργαλείο, ο *Autotuner* αναπτύχθηκε για την προσαρμογή και διαμόρφωση εφαρμογών που εκτελούνται φυσικά πάνω στον Intel Xeon Phi Coprocessor. Όμως, η πλατφόρμα δεν αποτελεί όριο και αν προσαρμόσουμε την εκτέλεση και τον μερικό χαρακτηρισμό για μια διαφορετική αρχιτεκτονική το εργαλείο μας μπορεί πολύ εύκολα να προσαρμοστεί και να δουλέψει αποδοτικά για το νέο σύστημα.

Προτάσεις για Έρευνα

Ο συγκεκριμένος *Autotuner* μπορεί να εξελιχθεί σε πολλές κατευθύνσεις. Πρώτα, μπορεί να προστεθεί μηχανισμός ώστε να αξιολογεί επίσης και παραπάνω της μιας εκτέλεσης πάνω στον συνεπεξεργαστή. Αυτό σημαίνει ότι θα είναι ικανός να μετράει την παρέμβαση μεταξύ των εφαρμογών που εκτελούνται και να τις αναθέσει σε ξένα μεταξύ τους σύνολα πυρήνων, κάτω από περιορισμούς απόδοσης και ενέργειας. Επίσης, ο host, Intel Xeon Processor, μπορεί επίσης θα γίνει μέρος της εξίσωσης και να αλλάξει η προσέγγιση την εκτέλεσης. Το κύριο εκτελέσιμο περιβάλλον αλλάζει σ' αυτό του host και ο συνεπεξεργαστής χρησιμοποιείται για offloading υπολογιστικά απαιτητικών περιοχών κώδικα. Έτσι, ο *Autotuner* πρέπει να παρακολουθεί την εκτέλεση της εφαρμογής και στον επεξεργαστή και στον συνεπεξεργαστή και να εξερευνήσει τις παραμέτρους προσαρμογής με στόχο να ωφεληθεί από την αρχιτεκτονική του. Οπότε, ο *Autotuner* μπορεί να αποκτήσει ένα πιο γενικό χαρακτήρα υποστηρίζοντας διαφορετικές αρχιτεκτονικές και μεθόδους εκτέλεσης, επεκτείνοντας σε GPUs και σε άλλες πολυπύρηνους επεξεργαστές.

Η χρησιμοποίηση της μηχανικής μάθησης στην αυτόματη προσαρμογή είναι μια πολύ ευοίωνη προσέγγιση και αποδείξαμε ότι είναι εφικτό να επιτύχουμε σπουδαία αποτελέσματα.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is evident that after almost five decades, the end of Moore's law is in sight. Making transistors smaller no longer guarantees that they will be faster, power efficient, reliable and cheaper. However, this does not mean progress in computing has reached its limit, but the nature of that progress is altered. So, in parallel with the now slower improvement of raw hardware performance, the future of computing will be primarily defined by improvements in three other areas.

The first one is software. Many examples and very recently AlphaGo[19], have demonstrated that huge performance gains can be achieved through new algorithms, persevering the current hardware. The second is the "cloud", the networks of data centers that deliver on-line services. By sharing their resources, existing computers can greatly add to their capabilities. Lastly, the third area lies in new computing architectures. Examples are multi- and many-core processors, accelerators(GPGPUs,FPGAs). These new architectures support also parallelism and they form the High Performance Computing(HPC) area.

Focusing on the latter, we note that parallel computer systems are getting increasingly complex. The reasons mainly lie in the exponential upsurge of data to interpret and the far more demanding applications (e.g. sciences, optimization, simulations). Systems of the past are incapable of processing that data quickly as the world demands. For these reasons, HPC systems today with a peak performance of several petaflops have hundreds of thousands of cores that have to be able to work together and use their resources efficiently. They consist of hardware units that accelerate a specific operation and they use evolved ideas in data storage, load balancing and intercommunication. Unfortunately, applications as they were used to run, do not deliver the maximum performance when they are ported on these systems. As a consequence, high amount of energy and money are being lost because of the low processor utilization. To reverse that state, programmers need to understood their machine's unique architecture and tune their program in a way that exploits every hardware unit optimally. Only then their program will get the maximum capable throughput.

Besides performance, there is a need to reduce power consumption of those systems. The energy crisis sets limitations to the amounts of power every system consumes. Towards that goal, a programmer applies advanced hardware and software techniques (e.g. GPUs, Dynamic Voltage and Frequency Scaling) which create a trade-off between performance and power saving, thus increasing the difficulty of the tuning process. By assembling all the choices and the configurations that a programmer has to weight in order to tune effectively his application, a homogeneous system transforms eventually into a heterogeneous one, which complicates programming tasks.

However, the tuning process is not at all easy. Application developers are investing significant time to tune their codes for the current and emerging systems. This tuning can be described as a cyclic process of gathering data, identifying code regions that can be improved, and tuning those code regions[24]. Alas, this task is toilsome, time-consuming and sometimes practically forbidden to be carried out manually.

The goal is to automate that task and outperform the human tuning. To accomplish that the research

community has divided the tuning process into two blocks, performance analysis tools and performance autotuning tools. Both the academic and the commercial world have developed tools that support and partially automate the identification and monitoring of applications[13, 43]. Moreover, special reference receives the Roofline model[65] which exposes the performance ceilings of an architecture along with the position of the application under evaluation, thus providing more accurate guidance in the tuning procedure.

The tools, until now, do not produce automatically an optimized executable, on the contrary they limit their operation to characterizations of code regions and simple suggestions to the developer, who manually makes any changes and repeats the tuning phase. As a result much research has been dedicated during the latest years to automate the tuning phase. Strategies that are employed, have as critical point to automatically and efficiently search for the best combination of parameter configurations of the execution environment for each particular application on a specific architecture. The ideas that have emerged in the area of automatic tuning have provided encouraging results and have highly benefited application programmers [52, 60, 38].

A leading example of manycore systems that is used today is Intel Many Integrated Core Architecture (Intel MIC)[4], a coprocessor developed by Intel. The first prototype was launched in 2010 with the code name *Knights Ferry*. A year later the *Knights Corner* product was announced and from June 2013 the coprocessor is in its second generation *Knights Landing*. Very soon, along with Intel Xeon processors-based systems, they became the primary components of supercomputers. Currently, Tianhe-2(MilkyWay-2) the supercomputer at National Supercomputer Center in Guangzhou, which holds the first place in the Top500 list[15], is composed of 32,000 Intel Xeon E5-2692 12C at 2.200 GHz and 48,000 Xeon Phi 31S1P, reaching 33,862.7 TFLOP/S. However, many applications have not yet been structured to take advantage of the full magnitude of parallelism, the high interconnection and bandwidth rates and the vectorization capabilities of the Intel Xeon Phi. Achieving high performance with that platform still needs lot of effort on parallelization, analysis and optimization strategies[33].

Hence, investigating and evaluating tuning on a state of the art system, such as Intel Xeon Phi, is certainly a very valuable and a promising contribution in the areas of HPC and auto-tuning.

1.1 Contribution

In this thesis, we develop an auto-tuning framework for Intel Xeon Phi co-processors based on analytical methods. Its purpose is to relieve the application developer from configuring the compiler and the execution environment by efficiently and optimally finding the solution that delivers the best outcome in respect of performance and power.

Shortly, the *Autotuner* has an offline database of performance data from a set of diverse applications executed on a set of configurations. These data were collected using LIKWID[62], a lightweight performance-oriented tool suite for x86 multicore environments. The framework uses these data to find correlations between the applications and the configurations that are being examined. To achieve this it uses a collaborative filtering technique[48, 35] that exploits the idea behind Singular Value Decomposition (SVD)[50]. Hence, applications and configurations are mapped to a feature space. That is a set of attributes, which consists of some configurations, and the scalar relation of the applications and the configurations to those attributes. Then each new application that arrives, is minimally profiled to a couple configurations and then it is projected to the constructed feature space, based on its ratings for the known configurations. Correlations with each feature are produced and consequently, its unknown ratings can be calculated. In the end, we have a fully populated vector with predicted ratings for all the configurations, from which we are able to choose the best predicted rating that corresponds to a specific configuration.

In addition, the auto-tuning framework we developed substantiates the employment of machine learning techniques and the utilization of their capabilities in the scarce field of autotuners and contributes significantly to it. Besides the fast predictions and the good performance, singular value decomposition also reduces the space needed for the characterization of the applications against the configurations, thus storing huge info in small space.

To have a glance at our results, the *Autotuner* manages to constantly report a tuning configuration that achieves more than 90% of the performance that corresponds to the best execution. In addition, that happens in less than 8 minutes, which is the time for the partial profiling of the application over a couple tuning configurations. Figure 1.1 shows the performance achieved from the predicted configurations for 6 applications.

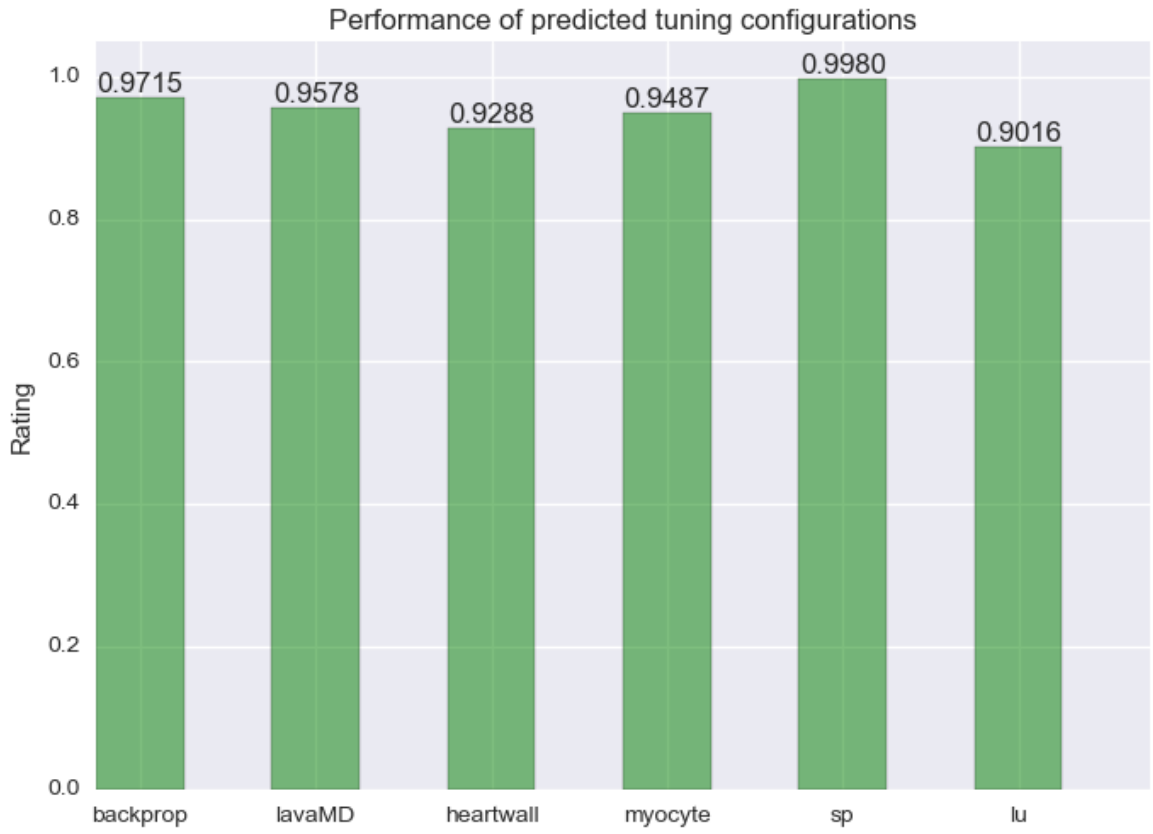


Figure 1.1: Performance achieved from predicted configurations in less than 8 minutes profiling.

1.2 Thesis Structure

The thesis is organized as follows:

Chapter 2

We describe performance analysis and tuning tools along with worth noting examples. In addition some related auto-tuners and their field of application. Lastly, we point out the main differences between our auto-tuning framework and the rest of the bibliography.

Chapter 3

We describe the system we used, Intel Xeon Phi and the application programming interface (API). Furthermore we present the Roofline model [**reference**] for the Intel Xeon Phi. Lastly, we define our tuning exploration space and we describe the applications used in the evaluation.

Chapter 4

We present meticulously the strategy and the theory behind the *Autotuner* and its building blocks. How we extract information while executing an application and how we apply collaborative filtering for our recommendation system.

Chapter 5

We demonstrate the experimental results and the efficiency of the *Autotuner*. We perform an scrupulous evaluation with many varying parameters.

Chapter 6

We briefly conclude and refer to future work.

Appendix A

User manual and source code for the set up of the *Autotuner* framework.

Chapter 2

Related Work

2.1 Performance Analysis

Performance analysis tools support the programmer in gathering execution data of an application and identifying code regions that can be improved. Overall, they monitor a running application. Performance data are both summarized and stored as profile data or all details are stored in trace files.

State of the art performance analysis tools fall into two major classes depending on their monitoring approach:

- profiling tools
- and tracing tools

Profiling tools summarize performance data for the overall execution and provide information such as the execution time for code regions, number of cache misses, time spent in MPI routines, and synchronization overhead for OpenMP synchronization constructs.

Tracing tools provide information about individual events, generate typically huge trace files and provide means to visually analyze those data to identify bottlenecks in the execution.

Representatives for these two classes are Gprof[3], OmpP[36], Vampir[17], PAPI[26], Likwid[62] and Intel® VTune™Amplifier[43].

2.1.1 Gprof

Gprof is the GNU Profiler tool. It provides a flat profile and a call graph profile for the program's functions. Instrumentation code is automatically inserted into the code during compilation, to gather caller-function data. The flat profile shows how much time the program spent in each function and how many times that function was called. The call graph shows for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. Lastly there is the annotated source listing which is a copy of the program's source code, labeled with the number of times each line of the program was executed. Yet, Gprof cannot measure time spent in kernel mode (syscalls, waiting for CPU or I/O waiting) and it is not thread-safe. Typically it only profiles the main thread.

2.1.2 OmpP

OmpP is a profiling tool specifically for OpenMP developed at TUM and the University of Tennessee. It is based on instrumentation with Opari, while it supports the measurement of hardware performance counters using PAPI. It is capable to expose productivity features such as overhead analysis and detection of common inefficiency situations and determines certain overhead categories of parallel regions.

2.1.3 Vampir

Vampir is a commercial trace-based performance analysis tool for MPI, from Technische Universität Dresden. It provides a powerful visualization of traces and scales to thousands of processors based on a parallel visualization server. It relies on the MPI profiling interface that allows the interception and replacement of MPI routines by simply re-linking the user-application with the tracing or profiling library The tool is well-proven and widely used in the high performance computing community for many years.

2.1.4 PAPI

The Performance API (PAPI) project specifies a standard application programming interface for accessing hardware performance counters available on most modern microprocessors. Developed at the University of Tennessee, it provides two interfaces to the underlying counter hardware; a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level ,interface directed towards users with more sophisticated needs. In addition, it provides portability across different platforms. It can be used both as a standalone tool and as lower layer of 3rd party tools (ompP, Vampir etc.)

2.1.5 Intel® VTune™Amplifier

Intel® VTune™Amplifier is the commercial performance analysis tool of Intel. It provides insight into CPU and GPU performance, threading performance and scalability, bandwidth, caching, hardware event sampling etc. In addition, it provides detailed data for each OpenMP region highlights tuning opportunities.

2.1.6 Likwid

Likwid (Like I knew What I Am Doing) developed at University of Erlangen-Nuremberg, is a set of command-line utilities that addresses four key problems: Probing the thread and cache topology of a shared-memory node, enforcing thread-core affinity on a program, measuring performance counter metrics, and toggling hardware prefetchers. An API for using the performance counting features from user code is also included.

To the previous list have been added lately PA tools that automate the analysis and improve the scalability of the tools. In addition, automation of the analysis facilitates a lot the application developer's task.

These tools are based on the APART Specification Language, a formalization of the performance problems and the data required to detect them, with aim of supporting automatic performance analysis for a variety of programming paradigms and architectures.

Some are:

2.1.7 Paradyn

Paradyn[11] is a performance measurement tool for parallel and distributed programs from University of Wisconsin, and it was the first automatic online analysis tool. It is based on a dynamic notion of performance instrumentation and measurement. Unmodified executable files are placed into execution and then performance instrumentation is inserted into the application program and modified during execution. The instrumentation is controlled by the Performance Consultant module, that automatically directs the placement of instrumentation. The Performance Consultant has a well-defined notion of performance bottlenecks and program structure, so that it can associate bottlenecks with specific causes and specific parts of a program.

2.1.8 SCALASCA

SCALASCA[13] is an automatic performance analysis tool developed at the German Research School on Simulation Sciences, the Technische Universität Darmstadt and Forschungszentrum Jülich. It is based on performance profiles as well as on traces. It supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior. The analysis identifies potential performance bottlenecks - in particular those concerning communication and synchronization - and offers guidance in exploring their causes.

2.1.9 Periscope

Periscope[39] is an automatic performance analysis tool for highly parallel applications written in MPI and/or OpenMP developed at Technische Universität München. Unique to Periscope is that it is an online tool and it works in a distributed fashion. This means that the analysis is done while the application is executing (online) and by a set of analysis agents, each searching for performance problems in a subset of the application's processes (distributed). The properties found by Periscope point to code regions that might benefit from further tuning.

Many more tools have been developed over the years. We mentioned some examples with particular interest.

2.2 Performance Autotuning

The core of the tuning process is the search for the optimal combination of code transformations and parameter settings of the execution environment that satisfy a specific goal. This creates an enormous search space which further complicates the tuning task. Thus, automation of that step is more than essential. Much research has been conducted on that matter and as a result many different ideas have been published. These can be grouped into four categories:

- self-tuning libraries for linear algebra and signal processing like ATLAS, FFTW, OSKI, FEniCS and SPIRAL;
- tools that automatically analyze alternative compiler optimizations and search for their optimal combination;
- autotuners that search a space of application-level parameters that are believed to impact the performance of an application;
- frameworks that try to combine ideas from all the other groups.

2.2.1 Self-Tuning libraries

The Automatically Tuned Linear Algebra Software[1] (ATLAS) supports the developers in applying empirical techniques in order to provide portable performance to numerical programs. It automatically generates and optimizes the popular Basic Linear Algebra Subroutines (BLAS) kernels for the currently used architecture.

Similarly, FFTW[34] is a library for producing efficient signal processing kernels on different architectures without modification.

OSKI[9] is a collection of low-level C primitives that provide automatically tuned computational kernels on sparse matrices, for use in solver libraries and applications.

The FEniCS Project[20] is a collaborative project for the development of innovative concepts and tools for automated scientific computing, with a particular focus on the solution of differential equations by finite element methods.

Divergent from the previous, SPIRAL[56] is a program generation system (software that generates other software) for linear transforms and an increasing list of other mathematical functions. The goal of Spiral is to automate the development and porting of performance libraries producing high-performance tuned signal processing kernels.

2.2.2 Compiler optimizations search

This approach is based on the need for more general and application independent auto-tuning. Hence, the goal is to define the right compiler optimization parameters on any platform. Such tools are divided into two categories, depending on their strategy.

Iterative search tools iteratively enable certain optimization parameters and run the compiled program while monitoring its execution. Following, based on the outcome, they decide on the new tuning combination. Due to the huge search space, they are relatively slow. In order to tackle that drawback some algorithms have been built that prune the search space.

Triantafyllis et al[63] as well as Haneda et al[41] enhance that idea by employing heuristics and statistic methods achieving remarkable results.

Machine Learning tools use knowledge about the program's behavior and machine learning techniques (e.g. linear regression, support vector machines) to select the optimal combination of optimization parameters. This approach is based on an automatically build per-system model which maps performance counters to good optimization options. This model can then be used with different applications to guide their tuning. Current research work is also targeting the creation of a self-optimizing compiler that automatically learns the best optimization heuristics based on the behavior of the underlying platform, as the work of Fursin et al[37] indicates. In general, machine learning tools explore a much larger space and faster comparing with iterative search tools.

Ganapathi et al[38], Bergstra et al[25], Leather et al[49] and Cavazos et al[27] are some who have experimented with machine learning techniques in auto-tuning with auspicious results.

2.2.3 Application parameters search

Somehow more specific, this approach evaluates application's behavior by exploring its parameters and implementation. By parameters we refer to global loop transformations (i.e. blocking factor, tiling,

loop unroll, etc) and by implementation we refer to which algorithms are being used. Thus, this technique requires in advance some info regarding the application and which parameters should be tuned, although it is able to get some generality regarding applications with common functions such as matrix multiplications.

Tools in this category are also divided into two groups. 1. Iterative search tools 2. and Machine learning tools.

The Intel Software Autotuning tool (ISAT)[51] is an example of iterative search tool which explores an application's parameter space which is defined by the user. Yet, it is a time consuming task.

The Active Harmony system[61] is a runtime parameter optimization tool that helps focus on the application-dependent parameters that are performance critical. The system tries to improve performance during a single execution based on the observed historical performance data. It can be used to tune parameters such as the size of a read-ahead buffer or what algorithm is being used (e.g., heap sort vs. quick sort).

Focusing on the algorithmic autotuning, Ansel et al[22] developed PetaBricks a new implicitly parallel programming language for high performance computing. Programs written in PetaBricks can naturally describe multiple algorithms for solving a problem and how they can be fit together. This information is used by the PetaBricks compiler and runtime to create and autotune an optimized hybrid algorithm. The PetaBricks system also optimizes and autotunes parameters relating to data distribution, parallelization, iteration, and accuracy. The knowledge of algorithmic choice allows the PetaBricks compiler to automatically parallelize programs using the algorithms with the most parallelism.

A different approach followed by Nelson et al[55], interacts with the programmer to get high-level models of the impact of parameter values. These models are then used by the system to guide the search for optimization parameters. This approach is called model-guided empirical optimization where models and empirical techniques are used in a hybrid approach.

Using a totally different method from everything else MATE (Monitoring, Analysis and Tuning Environment)[53] is an online tuning environment for MPI parallel applications developed by the Universidad Autònoma de Barcelona. The fundamental idea is that dynamic analysis and online modifications adapt the application behavior to changing conditions in the execution environment or in the application itself. MATE automatically instruments at runtime the running application in order to gather information about the applications behavior. The analysis phase receives events, searches for bottlenecks by applying a performance model and determines solutions for overcoming such performance bottlenecks. Finally, the application is dynamically tuned by setting appropriate runtime parameters. All these steps are performed automatically and continuously during application execution by using the technique called dynamic instrumentation provided by the Dyninst library. MATE was designed and tested for cluster and grid environments.

2.2.4 Compiler optimizations & Application parameters search

The last category mixes ideas and strategies from both the last two, achieving very positive results. Some solutions are problem targeted, meaning that they are implemented for specific applications and some others are more general as they tackle a bigger and more diverse set of applications. Proportionally, their complexity is increasing.

Many autotuning methods have been developed focusing on signal processing applications, matrix vector multiplication and stencil computations. They take into account both application's and system's environment parameters. Contributions to this approach come from S. Williams[66] who implements autotuners for two important scientific kernels, Lattice Boltzmann Magnetohydrodynamics (LBMHD) and sparse matrix-vector multiplication (SpMV). In an automated fashion, these autotuners

explore the optimization space for the particular computational kernels on an extremely wide range of architectures. In doing so, it is determined the best combination of algorithm, implementation, and data structure for the combination of architecture and input data.

As stencil computations are difficult to be assembled into a library because they have a large variety and diverse areas of applications in the heart of many structured grid codes, some autotuning approaches[31, 47, 30] have been proposed that substantiate the enormous promise for architectural efficiency, programmer productivity, performance portability, and algorithmic adaptability on existing and emerging multicore systems.

General autotuners need more information about each application they examine, and for that reason a performance tool is also needed to reckon the bottlenecks and the critical areas that will deliver more performance by optimization. Popular examples are Parallel Active Harmony, the Autopilot framework and the AutoTune project.

The Parallel Active Harmony[60] is a combination of the Harmony system and the CHiLL[29] compiler framework. It is an autotuner for scientific codes that applies a search-based autotuning approach. While monitoring the program performance, the system investigates multiple dynamically generated versions of the detected hot loop nests. The performance of these code segments is then evaluated in parallel on the target architecture and the results are processed by a parallel search algorithm. The best candidate is integrated into the application.

The Autopilot[57] is an integrated toolkit for performance monitoring and dynamical tuning of heterogeneous computational grids based on closed loop control. It uses distributed sensors to extract qualitative and quantitative performance data from the executing applications. This data is processed by distributed actuators and the preliminary performance benchmark is reported to the application developer.

AutoTune project[52] extends Periscope with plugins for performance and energy efficiency tuning, and constitutes a part of the Periscope Tuning Framework (PTF)[39]. PTF supports tuning of applications at design time. The most important novelty of PTF is the close integration of performance analysis and tuning. It enables the plugins to gather detailed performance information during the evaluation of tuning scenarios to shrink the search space and to increase the efficiency of the tuning plugins. The performance analysis determines information about the execution of an application in the form of performance properties. The HPC tuning plugins that implemet PTF are: Compliler Flags Selection Tuning, MPI Tuning, Energy Tuning, Tuning Master Worker Application and Tuning Pipeline Applications.

An ongoing tuning project is X-TUNE[18] which evaluates ideas to refine the search space and search approach for autotuning. Its goal is to seamlessly integrate programmer-directed and compiler-directed auto-tuning, so that a programmer and the compiler system can work collaboratively to tune a code, unlike previous systems that place the entire tuning burden on either programmer or compiler.

Readex project[12] is another current project which aims to develop a tools-aided methodology for dynamic autotuning for performance and energy efficiency. The project brings together experts from two ends of the compute spectrum: the system scenario methodology[40] from the embedded systems domain as well as the High Performance Computing community with the Periscope Tuning Framework (PTF).

2.3 Autotuners tested on Intel Xeon Phi

Many researchers have been experimenting on the coprocessor developed by Intel to establish a working and useful autotuner. Intel Xeon Phi coprocessor is interesting among the HPC community because

of its simple programming model and its highly parallel architecture. Hence, there is a trend to derive its maximum computational power through fine automatic tuning.

Wai Teng Tang et al[59] implemented sparse matrix vector multiplication (SpMV), a popular kernel among many HPC applications that use scale-free sparse matrices (e.g. fluid dynamics, social network analysis and data mining), on the Intel Xeon Phi Architecture and optimized its performance. Their kernel makes use of a vector format that is designed for efficient vector processing and load balancing. Furthermore, they employed a 2D jagged partitioning method together with tiling in order to improve the cache locality and reduce the overhead of expensive gather and scatter operations. They also employed efficient prefix sum computations using SIMD and masked operations that are specially supported by the Xeon Phi hardware. The optimal panel number in the 2D jagged partitioning method varies for different matrices due to their differences in non-zero distribution, hence a tuning tool was developed. Their experiments indicated that the SpMV implementation achieves an average 3x speedup over Intel MKL for scale-free matrices, and the performance tuning method achieves within 10 % of the optimal configuration.

Williams et al[64] explored the optimization of geometric multigrid (MG) - one of the most important algorithms for computational scientists - on a variety of leading multi- and manycore architectural designs, including Intel Xeon Phi. They optimized and analyzed all the required components within an entire multigrid V-cycle using a variable coefficient, Red-Black, Gauss-Seidel (GSRB) relaxation on these advanced platforms. They also implemented a number of effective optimizations geared toward bandwidth-constrained, wide-SIMD, manycore architectures including the application of wavefront to variable-coefficient, Gauss-Seidel, Red-Black (GSRB), SIMDization within the GSRB relaxation, and intelligent communication-avoiding techniques that reduce DRAM traffic. They also explored message aggregation, residual restriction fusion, nested parallelism, as well as CPUand KNC-specific tuning strategies. Overall results showed a significant performance improvement of up to 3.8x on the Intel Xeon Phi compared with the parallel reference implementation, by combining autotuned threading, wavefront, hand-tuned prefetching, SIMD vectorization, array padding and the use of 2MB pages.

Heirman et al[42] extent ClusteR - aware Undersubscribed Scheduling of Threads (CRUST), a variation on dynamic concurrency throttling (DCT) specialized for clustered last-level cache architectures, to incorporate the effects of simultaneous multithreading, which in addition to competition for cache capacity, exhibits additional effects incurred by core resource sharing. They implemented this improved version of CRUST inside the Intel OpenMP runtime library and explored its performance when running on Xeon Phi hardware. Finally, CRUST can be integrated easily into the OpenMP runtime library; by combining application phase behavior and leveraging hardware performance counter information it is able to reach the best static thread count for most applications and can even outperform static tuning on more complex applications where the optimum thread count varies throughout the application.

Sclocco et al[58] designed and developed a many-core dedispersion algorithm, and implemented it using the Open Computing Language (OpenCL). Because of its low arithmetic intensity, they designed the algorithm in a way that exposes the parameters controlling the amount of parallelism and possible data-reuse. They showed how, by auto-tuning these user-controlled parameters, it is possible to achieve high performance on different many-core accelerators, including one AMD GPU (HD7970), three NVIDIA GPUs (GTX 680, K20 and GTX Titan) and the Intel Xeon Phi. they not only auto-tuned the algorithm for different accelerators, but also used auto-tuning to adapt the algorithm to different observational configurations.

ppOpen-HPC[54] is an open source infrastructure for development and execution of large-scale scientific applications on post-peta-scale (pp) supercomputers with automatic tuning (AT). ppOpen-HPC focuses on parallel computers based on many-core architectures and consists of various types of libraries covering general procedures for scientific computations. The source code, developed on a PC

with a single processor, is linked with these libraries, and the parallel code generated is optimized for post-peta-scale systems. Specifically on the Intel Xeon Phi coprocessor, the performance of a parallel 3D finite-difference method (FDM) simulation of seismic wave propagation was evaluated by using a standard parallel finite-difference method (FDM) library (ppOpen-APPL/FDM) as part of the ppOpen-HPC project.

2.4 How our work is different from the bibliography?

The autotuning framework we developed is based on data mining. The autotuner derives its suggestions from an already known set of profiled applications against the full set of configuration space. Thus, it is sensitive on the choice of those applications that constitute its initial knowledge. It belongs in the category of machine learning autotuners like Ganapathi[38]. It explores mainly compiler and execution environment parameters because the configuration space of the coprocessor is large enough. There is not a specified target group of applications to autotune. It performs well independently of the current testing application that is why it is a general autotuner.

From our experience with this framework and the employment of data mining techniques, we conclude that valuable knowledge and fine tuning can be derived from their use and at the same time in timely fashion with high accuracy. We know the optimal tuning of many applications, we need only to project them to newly machine architectures in a way to benefit from their capabilities and specifications. Then it is able to find correlations between them and unoptimized applications in order to suggest the optimal tuning.

The idea to use data mining techniques in autotuning for a heterogeneous - because of its vary configurations - coprocessor came from the work of Delimitrou and Kozyrakis[32] who developed an heterogeneity- and interference-aware scheduler, Paragon, for large-scale datacenters. Paragon is an online and scalable scheduler based on collaborative filtering techniques to quickly and accurately classify an unknown incoming workload with respect to heterogeneity and interference in multiple shared resources.

Chapter 3

Experimental Testbed & Environment

3.1 Intel Xeon Phi

3.1.1 Architecture

In this work we used an Intel®Xeon Phi™coprocessor of the 3100 Series with code name Knights Corner. It is Intel's first many-cores commercial product made at a 22nm process size that uses Intel's Many Integrated Core (MIC) architecture. A coprocessor needs to be connected to a Host CPU, via the PCI Express bus and in that way they share access to main memory with other processors.

The coprocessor is a symmetric multiprocessor (SMP) on-a-chip running Linux. It consists of 57 cores who are in-order dual issue x86 processor cores, they support 64-bit execution environment-based on Intel64 Architecture and are clocked at 1.1GHz. Each one has four hardware threads, resulting in 228 available hardware threads. They are used mainly to hide latencies implicit to the in-order microarchitecture. In addition to the cores, the coprocessor has six memory controllers supporting two GDDR5 (high speed) memory channels each at 5GT/sec. Each memory transaction to the total 6GB GDDR5 memory is 4 bytes of data resulting in 5GT/s x 4 or 20GB/s per channel. 12 total channels provide maximum transfer rate of 240GB/s. Then it consists of other device interfaces including the PCI Express system interface.

All the cores, the memory controllers, and PCI Express system I/O logic are interconnected with a high speed ring-based bidirectional on-die interconnect (ODI), as shown in Figure 3.1. Communication over the ODI is transparent to the running code with transactions managed solely by the hardware.

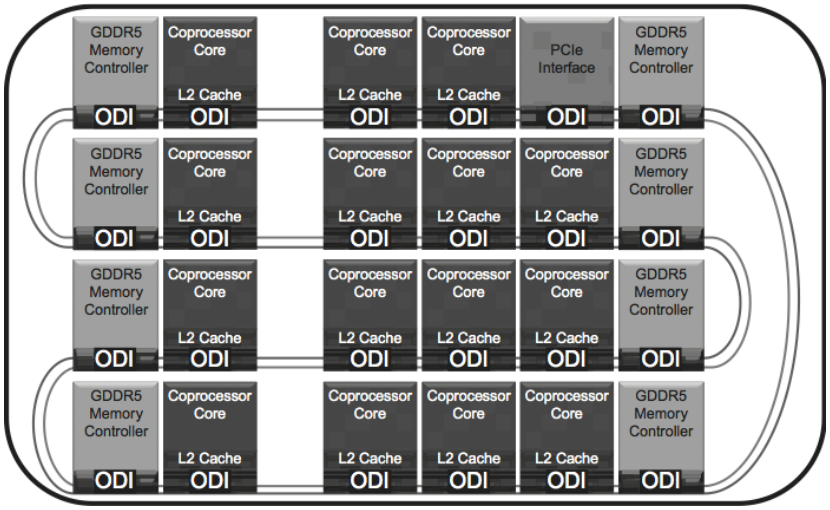


Figure 3.1: Overview of the coprocessor silicon and the On-Die Interconnect (ODI)[46].

At the core level, exclusives 32-KB L1 instruction cache and L1 data cache as well as a 512-KB Level 2 (L2) are assigned to provide high speed, reusable data access. In Table 3.1 we are summarized the main properties of the L1 and L2 caches.

Parameter	L1	L2
Coherence	MESI	MESI
Size	32 KB + 32 KB	512 KB
Associativity	8-way	8-way
Line size	64 Bytes	64 Bytes
Banks	8	8
Access Time	1 cycle	11 cycles
Policy	Pseudo LRU	Pseudo LRU

Table 3.1: Coprocessors core's cache parameters.

Furthermore, fast access to data in another core's cache over the ODI is provided to improve performance when the data already resides "on chip." Using a distributed Tag Directory (TD) mechanism, the cache accesses are kept "coherent" such that any cached data referenced remains consistent across all cores without software intervention. There are two primary instruction processing units. The scalar unit executes code using existing traditional x86 and x87 instructions and registers. The vector processing unit (VPU) executes the Intel Initial Many Core Instructions (IMCI) utilizing a 512-bit wide vector length enabling very high computational throughput for both single and double precision calculations. Along there is an Extended Math Unit (EMU) for high performance key transcendental functions, such as reciprocal, square root, power and exponent functions. The microarchitectural diagram of a core is shown in the Figure 3.2.

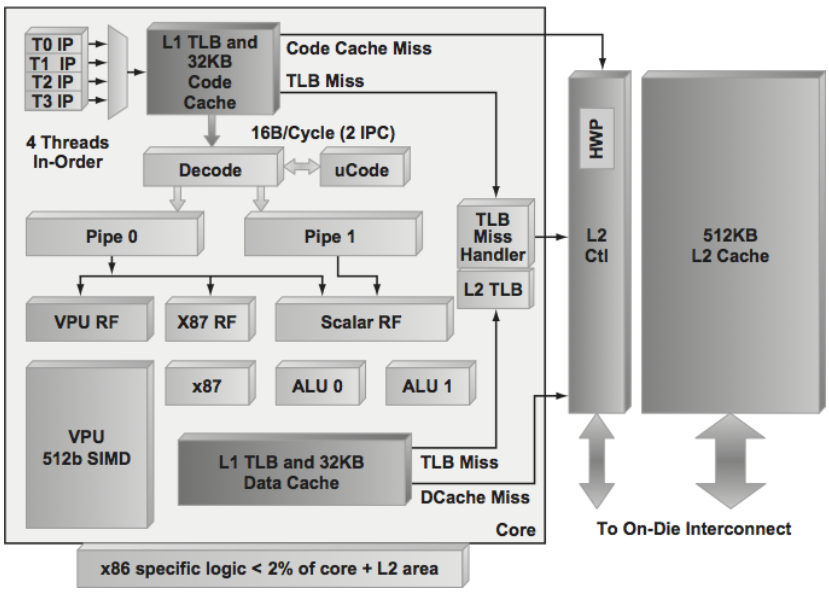


Figure 3.2: Architecture of a Single Intel Xeon Phi Coprocessor Core[46].

Each core's instruction pipeline has an in-order superscalar architecture. It can execute two instructions per clock cycle, one on the U-pipe and one on the V-pipe. The V-pipe cannot execute all instruction types, and simultaneous execution is governed by instruction pairing rules. Vector instructions are mainly executed only on the U-pipe. The instruction decoder is designed as a two-cycle fully pipelined unit, which greatly simplifies the core design allowing for higher cycle rate than otherwise could be implemented. The result is that any given hardware thread that is scheduled back-to-back will stall in decode for one cycle. Therefore, single-threaded code will only achieve a maximum of 50% utilization

of the core's computational potential. However, if additional hardware thread contexts are utilized, a different thread's instruction may be scheduled each cycle and full core computational throughput of the coprocessor can be realized. Therefore, to maximize the coprocessor silicon's utilization for compute-intensive application sequences, at least two hardware thread contexts should be run.

The coprocessor silicon supports virtual memory management with 4 KB (standard), 64 KB (not standard), and 2 MB (huge and standard) page sizes available and includes Translation Lookaside Buffer (TLB) page table entry cache management to speed physical to virtual address lookup as in other Intel architecture microprocessors.

The Intel Xeon Phi coprocessor includes memory prefetching support to maximize the availability of data to the computation units of the cores. Prefetching is a request to the coprocessor's cache and memory access subsystem to look ahead and begin the relative slow process of bringing data we expect to use in the near future into the much faster to access L1 and/or L2 caches. The coprocessor provides two kinds of prefetch support, software and hardware prefetching. Software prefetching is provided in the coprocessor VPU instruction set. The processing impact of the prefetch requests can be reduced or eliminated because the prefetch instructions can be paired on the V-pipe in the same cycle with a vector computation instruction. The hardware prefetching (HWP) is implemented in the core's L2 cache control logic section.

3.1.2 Performance Monitoring Units

In order to monitor hardware events, the coprocessor is supported by a performance monitoring unit (PMU). Each physical Intel Xeon Phi coprocessor core has an independent-programmable core PMU with two performance counters and two event select registers, thus it supports performance monitoring at the individual thread level. User-space applications are allowed to interface with and use the PMU features via specialized instructions such as RDMSR, WRMSR, RDTSC, RDPMC. Coprocessor-centric events are able to measure memory controller events, vector processing unit utilization and statistics, local and remote cache read/write statistics, and more[44]. In Table 3.2, are shown some important hardware events of the coprocessor. The rest can be found on [2].

3.1.3 Power Management

Unlike the multicore family of Intel Xeon processors, there is no hardware-level power control unit in the coprocessor. Instead power management (PM) is controlled by the coprocessor's operating system and is performed in the background. Intel Xeon Phi coprocessor power management software is organized into two major blocks. One is integrated into the coprocessor OS running locally on the coprocessor hardware. The other is part of the host driver running on the host. Each contributes uniquely to the overall PM solution.

The power management infrastructure collects the necessary data to select performance states and target idle states for the individual cores and the whole system. Below, we describe these power states[44, 45]:

Coprocessor in C0 state; Memory in M0 state In this power state, the coprocessor (cores and memory) is expected to operate at its maximum thermal design power (TDP), for our coprocessor that is 300 Watts. While in that state, all cores are active and run at the same P-state, or performance state. P-states are different frequency settings that the OS or the applications can request. Each frequency setting of the coprocessor requires a specific voltage identification (VID) voltage setting in order to guarantee proper operation, thus each P-state corresponds to one of these frequency and voltage pairs.

Event	Description
CPU_CLK_UNHALTED	The number of cycles (commonly known as clock-ticks) where any thread on a core is active. A core is active if any thread on that core is not halted. This event is counted at the core level at any given time, all the hardware threads running on the same core will have the same value.
INSTRUCTIONS_EXECUTED	Counts the number of instructions executed by a hardware thread.
DATA_CACHE_LINES_WRITTEN_BACK	Number of dirty lines (all) that are written back, regardless of the cause.
L2_DATA_READ_MISS_MEM_FILL	Counts data loads that missed the local L2 cache, and were serviced from memory (on the same Intel Xeon Phi coprocessor). This event counts at the hardware thread level. It includes L2 prefetches that missed the local L2 cache and so is not useful for determining demand cache fills or standard metrics like L2 Hit/Miss Rate.
L2_DATA_WRITE_MISS_MEM_FILL	Counts data Reads for Ownership (due to a store operation) that missed the local L2 cache, and were serviced from memory (on the same Intel Xeon Phi coprocessor). This event counts at the hardware thread level.

Table 3.2: Some hardware events of the coprocessor.

P1 is the highest P-state setting and it can have multiple sequentially lower frequency settings referred as P2,P3,…,Pn where Pn is the lowest pair.

Some cores are in C0 state and other cores in C1 state; Memory in M0 state When all four threads in a core have halted, the clock at the core shuts off, changing his state to C1. The last thread to halt is responsible to collect idle residency data and store it in a data structure accessible to the OS. A coprocessor can have some cores in C0 state and some in C1 state with memory in M0 state. In this case, clocks are gated on a core-by-core basis, reducing core power and allowing the cores in C1 state to lose clock source. After a core drops in C1 state, there is the option the core shuts down, become electrically isolated. That is the core C6 state and it is decided by the coprocessor's PM SW, which also writes to a certain status register the current core's status before issuing HALT to all the threads active on that core. The memory clock can be fully stopped to reduce memory power and memory subsystem enters M3 state. The price of dropping into a deeper core C state is an added latency resulting from bringing the core back up to the non-idle state, so the OS evaluates if the power-savings are worthwhile.

The coprocessor in package Auto-C3 state; Memory in M1 state If all the cores enter C1 state, the coprocessor automatically enters auto-package C3 (PC3) state by clock gating also the uncore part of the card. For this transition both the coprossesor's PM software and the host's coprocessor PM are involved, that is because it may be needed a core to return to C0 state and in order to happen the coprossesor PM SW must initiate it. In addition, the host's coprocessors PM may override the request to PC3 under certain conditions, such as when the host knows that the uncore part of the coprocessor is still busy. Finally, the clock source to the memory can be gated off also, thus reducing memory

power. This is the M1 state for the memory.

The coprocessor in package Deep-C3; Memory in M2 state In this state only the host's coprocessor PM SW functions and decides for the transitions as it has a broader sense of the events on the coprocessor and the coprocessor's PM SW is essential suspended for the power savings. So the host's coprocessor PM SW looks at idle residency history, interrupts (such as PCI Express traffic), and the cost of waking the coprocessor up from package Deep-C3 to decide whether to transition from package Auto-C3 state into package Deep-C3 state. In package Deep-C3 the core voltage is further reduced and the voltage regulators (VRs) enter low power mode. The memory changes to self-refresh mode, i.e. M2 state.

The coprocessor in package C6; Memory in M3 state The transition to this state can be initiated from both the coprocessor and the host. More reductions in power consumption are done in the uncore part, the cores are shut off and the memory clock can be fully stopped, reducing memory power to its minimum state (M3).

The Table 3.3 shows the power consumed in each state.

Coprocessor's Power State	Power(Watts)
C0	300
C1	<115
PC3	<50
PC6	<30

Table 3.3: Coprocessor's power consumption on different power states[45].

3.2 Roofline Model

The roofline model is a visual performance model that offers insights to programmers on improving parallel software for floating point computations relatively to the specifications of the architecture used or defined by the user. Proposed by Williams et al [65], it has been used and proved valuable both to guide manual code optimization and in education. Therefore, creating the roofline for our testbed will aid us in the characterization of the autotuning process, how exactly the unoptimized and optimized benchmarks move in the 2D space. Firstly, we describe the roofline model and its background.

3.2.1 Model's Background

The platform's peak computational performance - generally floating operations - together with the peak memory bandwidth - generally between the CPU and the main memory - create a performance "ceiling" in the 2 dimensional space. These peak values are calculated from the hardware specifications. On the x-axis is the operational intensity, which is defined as the amount of floating points operations per byte of main memory traffic. On the y-axis is the performance. Both axis are in log scale. The roofline is visually constructed by one horizontal and one diagonal line. The horizontal line is the peak performance and the diagonal is the performance limited by memory bandwidth. Thus, the mathematical equation is:

$$\text{Roofline(op. intensity)} = \min(\text{BW} * \text{op. intensity}, \text{peak performance})$$

The point where the two lines intersect:

$$\text{Peak Performance} = \text{Operational Intensity} * \text{Memory Bandwidth}$$

is called ridge point and defines the minimum operational intensity that is required in order to reach maximum computational performance. In addition, the area on the right of the ridge point is called computational bound and on the left memory bound.

Besides using the peak values calculated from the architecture, one can create a roofline using software peak values (lower than the ones from hardware) such as performance limited to thread level parallelism , instruction level parallelism and SIMD, without memory optimizations (e.g. prefetches, affinity). In addition, more realistic performance ceilings can be obtained by running standard benchmarks such as the high-performance LINPACK[8] and the Stream Triad[14] benchmarks. We assume that any real world application's performance can be characterized somewhere between totally memory bandwidth bound (represented by Stream Triad) and totally compute or Flop/s bound (represented by Linpack).

For a given kernel, we can find a point on the x-axis based on its operation intensity. A vertical line from that point to the roofline shows what performance is able to achieve for that operational intensity[65]. From the definition of the ridge point, if the operational intensity of the kernel is on the left of the ridge point then the kernel is bound from the bandwidth performance and if it is on the right then the kernel is bound from the peak computational performance. So, by plotting along with the peak performances also the performances from the software tunings, it can be reckoned what optimizations will benefit the most the kernel under examination, guide in other words the developer for the optimum tuning appropriately.

3.2.2 The Roofline of our Testbed

As we noted before, our testbed consists of one Intel Xeon Phi coprocessor 3120A. From the technical specifications we can calculate the theoretical peak computational performance. With 57 cores, each running at maximum 1.1GHz, a 512-bit wide VPU unit and support of the instruction fused multiply and add (FMA) enabling two floating point operations in one instruction, the peak computational performance is obtained from the formula:

$$\text{Clock Frequency} \times \text{Number of Cores} \times 16 \text{ lanes(SP floats)} \times 2\text{(FMA) FLOPs/cycle}$$

So, by substituting the technical specification values we get: 2006.4 GFlops/sec for SP and 1003.2 GFlops/sec for DP, which is usually the reported one. The theoretical bandwidth between the CPU and the main memory is 240GB/sec.

By running the standard performance benchmarks with the appropriate optimizations, we get from Linpack 727.9911[1] GFlops/sec (DP) and from Stream triad we get 128.31[2] GB/sec. Both values are very close to the ones reported by Intel using the same benchmarks[6]. The choice of these two benchmarks provides a strong hypothesis and we may argue that even if they remain far from ideal reference points, they represent a better approximation than the hardware theoretical peaks because they at least include the minimum overhead required to execute an instruction stream on the processing device[21].

[1] The performance reported was achieved with the following configuration: compact affinity, 228 threads, size=24592, ld=24616, 4KB align. The optimized benchmark from the Intel was used[5]

[2] The performance reported was achieved with the following configurations as they are suggested here[10]: 110M elements per array, prefetch distance=64,8, streaming cache evict=0, streaming stores=always, 57 threads, balanced affinity.

So, now we can compute the operational intensity (OI) of the ridge point for both the theoretical and the achievable peak performances, as: (using double precision)

$$OI_R^{th} = \frac{1003.2}{240} = 4.18 Flops/Byte$$

$$OI_R^{ac} = \frac{727.9911}{128.31} = 5.67 Flops/Byte$$

Figure 3.3 shows the roofline model for our testbed in double precision.

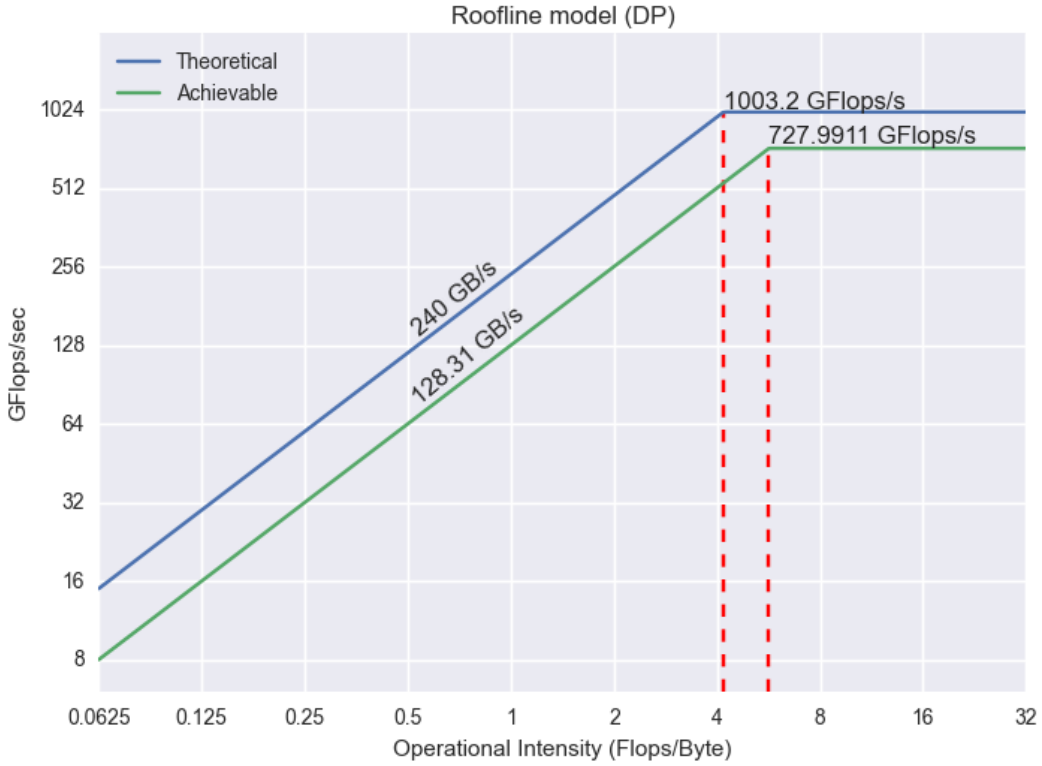


Figure 3.3: Roofline model

3.3 Tuning Parameters

In this section, we describe the parameters we used to build our tuning space. It is composed of compiler's flags as well as environmental configurations for the tuning of each application. The compiler is the system's default, Intel®C Intel®64 Compiler XE for applications running on Intel®64, Version 14.0.3.174 (icc).

3.3.1 Compiler's Flags

The compiler's flags used where chosen from the icc's optimization category.

-O[=n]

Specifies the code optimization for applications.
Arguments:

O2: Enables optimizations for speed. Vectorization is enabled at O2 and higher levels. Some basic loop optimizations such as Distribution, Predicate Opt, Interchange, multi-versioning, and scalar replacements are performed. More detailed information can be found on [16].

O3: Performs O2 optimizations and enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements. The O3 optimizations may not cause higher performance unless loop and memory access transformations take place. The optimizations may slow down code in some cases compared to O2 optimizations.

-opt-prefetch[=n]

This option enables or disables prefetch insertion optimization. The goal of prefetching is to reduce cache misses by providing hints to the processor about when data should be loaded into the cache.
Arguments:

0: Disables software prefetching.

2-4: Enables different levels of software prefetching.

Prefetching is an important topic to consider regardless of what coding method we use to write an algorithm. To avoid having a vector load operation request data that is not in cache, we can make sure prefetch operations are happening. Any time a load requests data not in the L1 cache, a delay occurs to fetch the data from an L2 cache. If data is not in any L2 cache, an even longer delay occurs to fetch data from memory. The lengths of these delays are nontrivial, and avoiding the delays can greatly enhance the performance of an application.

-opt-streaming-stores [keyword]

This option enables generation of streaming stores for optimization. This method stores data with instructions that use a non-temporal buffer, which minimizes memory hierarchy pollution.
Arguments:

never: Disables generation of streaming stores for optimization. Normal stores are performed.

always: Enables generation of streaming stores for optimization. The compiler optimizes under the assumption that the application is memory bound.

Streaming stores are a special consideration in vectorization. Streaming stores are instructions especially designed for a continuous stream of output data that fills in a section of memory with no gaps between data items. An interesting property of an output stream is that the result in memory does not require knowledge of the prior memory content. This means that the original data does not need to be fetched from memory. This is the problem that streaming stores solve - the ability to output a data stream but not use memory bandwidth to read data needlessly. Having the compiler generate streaming stores can improve performance by not having the coprocessor fetch caches lines from memory that will be completely overwritten. This effectively avoids wasted prefetching efforts and eventually helps with memory bandwidth utilization.

-opt-streaming-cache-evict[=n]

This option specifies the cache eviction (clevict) level to be used by the compiler for streaming loads and stores. Depending on the level used, the compiler will generate clevict0 and/or clevict1 instructions that evict the cache-line (corresponding to the load or the store) from the first-level and second-level caches. These cache eviction instructions will be generated after performing the corresponding load/store operation.
Arguments:

0: Tells the compiler to use no cache eviction level.

1: Tells the compiler to use the L1 cache eviction level.

2: Tells the compiler to use the L2 cache eviction level.

3: Tells the compiler to use the L1 and L2 cache eviction level.

-unroll[=n]

This option tells the compiler the maximum number of times to unroll loops.
Arguments:

0: Disables loop unrolling.

N/A: With unspecified n, the optimizer determines how many times loops can be unrolled.

The Intel C Compiler can typically generate efficient vectorized code if a loop structure is not manually unrolled. Unrolling means duplicating the loop body as many times as needed to operate on data using full vectors. For single precision on Intel Xeon Phi coprocessors, this commonly means unrolling 16-times. In other words, the loop body would do 16 iterations at once and the loop itself would need to skip ahead 16 per iteration of the new loop.

3.3.2 Huge Pages

To get good performance for executions on the coprocessor, huge memory pages (2MB) are often necessary for memory allocations on the coprocessor. This is because large variables and buffers are sometimes handled more efficiently with 2MB vs 4KB pages. With 2MB pages, TLB misses and page faults may be reduced, and there is a lower allocation cost.

In order to enable 2MB pages for applications running on the coprocessor we can either manually instrument the program with mmap system calls or use the hugetlbfs library[7]. In our case, we used the hugetlbfs library dynamically linked with the applications.

Although, Manycore Platform Software Stack(MPSS) versions later than 2.1.4982-15 support “Transparent Huge Pages (THP)” which automatically promotes 4K pages to 2MB pages for stack and heap allocated data, we use the hugetlbfs library to allocate data directly in 2MB pages. This is useful because if the data access pattern is such that the program can still benefit from allocating data in 2MB pages even though THP may not get triggered in the uOS.

So, we examined the performance of the applications with huge pages enable or not.

3.3.3 OpenMP Thread Affinity Control

Threading and Thread placement

As a minimum number of threads per physical core we set 2 because of the two-cycle fully pipelined instruction decoder as we mentioned in the coprocessor's architecture. We examine the performance of each application on 19, 38 and 57 physical cores and with the different combinations of enabled threads per core we get 38, 57, 76, 114, 152, 171 and 228 threads with different however mappings on the cores. That is implemented by using the environmental variable *KMP_PLACE_THREADS=ccC,ttT,ooO*, where:

C: denotes the number of physical cores

T: denotes the number of threads per core

O: denotes the offset of cores

So, with this variable we specify the topology of the system to the OpenMP runtime.

Affinity

In order to specify how the threads are bound within the topology we use the environmental variable *KMP_AFFINITY[=type]*, where type:

scatter: The threads are distributed as evenly as possible across the entire system. OpenMP thread numbers with close numerical proximity are on different cores.

balanced: The threads are distributed as evenly as possible across the entire system while ensuring the OpenMP thread numbers are close to each other.

Below, the Figure 3.4 illustrates the two different affinity types. We note that both types use cores before threads, thus they gain from every available core. In addition, while in balanced thread allocation cache utilization should be efficient if the neighbor threads access data that is near in store. Generally however, tuning affinity is a complicated and machine specific process.

In the Table 3.4 we present a summary of the tuning parameters with their possible values. The total combinations are 2880 tuning states.

Flag	Arguments
-O[=n]	n=2,3
-opt-prefetch[=n]	n=0,2,3,4
-opt-streaming-stores [keyword]	keyword=never,always
-opt-streaming-cache-evict[=n]	m=0,1,2,3
-unroll	enabled/disabled
huge pages	enabled/disabled
affinity [type]	type=scatter,balanced
cores	19,38,57
threads per core	2,3,4

Table 3.4: Summary of tuning parameters.

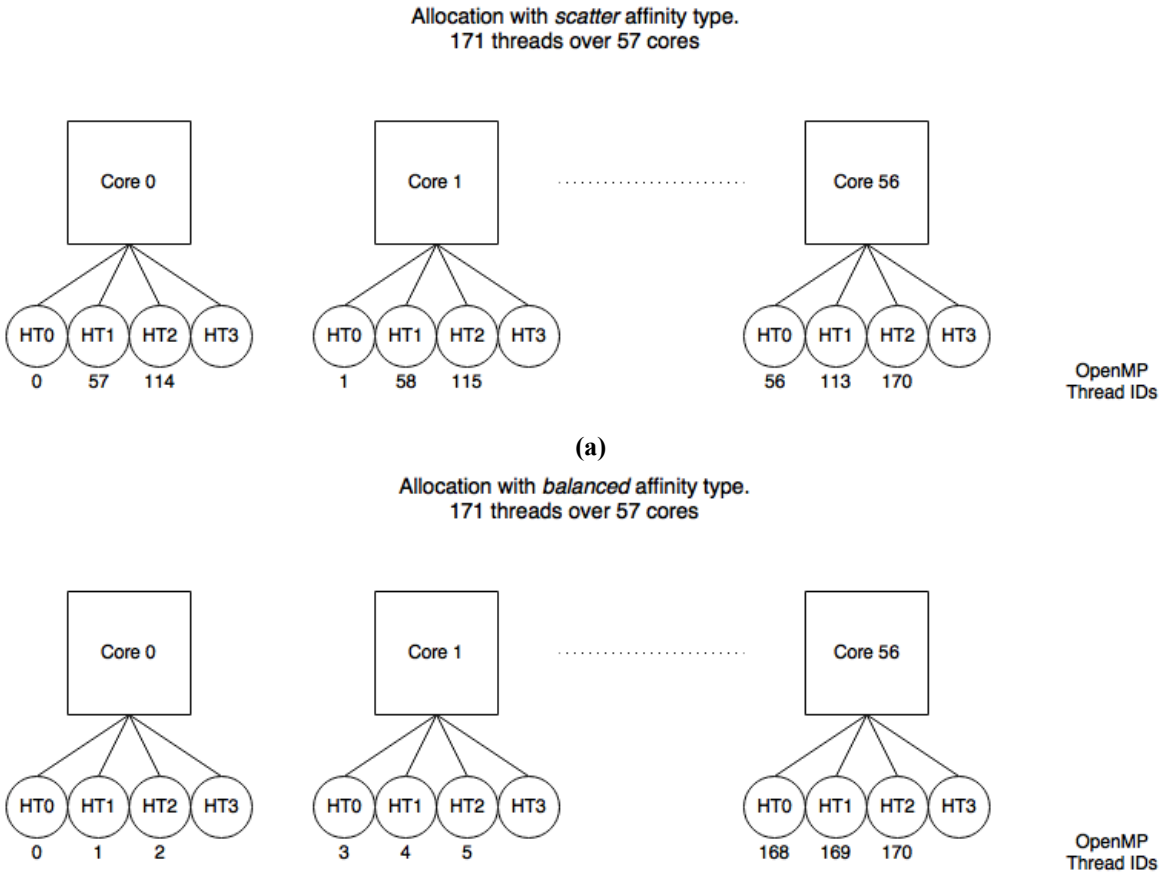


Figure 3.4: Affinity Types (a) Scatter (b) Balanced.

3.4 Applications Used

In order to build and evaluate our autotuner for the coprocessor we used applications from two benchmarks suites, the Rodinia Benchmark Suite[28] and the NAS Parallel Benchmarks[23]. We focused only on applications with floating point operations and profiled them against the previous tuning states. We used the OpenMP implementations.

3.4.1 Rodinia Benchmarks Suite

Rodinia is a benchmark suite for heterogeneous computing. It includes applications and kernels which target multi- and manycore CPU and GPU platforms. The choice of applications is inspired by Berkeley's dwarf taxonomy. It has been shown[28] that Rodinia covers a wide range of parallel communication patterns, synchronization techniques and power consumption and has led to some important architectural insight, such as the growing importance of memory bandwidth limitations and the consequent importance of data layout.

Below we list the applications and kernels we used along with some specifications.

LU Decomposition(LUD)

LU Decomposition is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. This application has many row-wise and column-wise interdependencies and requires significant optimization to achieve good parallel performance. The size of the matrix was 8000x8000.

Hotspot

Hotspot is a thermal simulation tool used for estimating processor temperature based on an architectural floor plan and simulated power measurements. It includes the 2D transient thermal simulation kernel of HotSpot, which iteratively solves a series of differential equations for block temperatures. The inputs to the program are power and initial temperatures. Each output cell in the grid represents the average temperature value of the corresponding area of the chip. There is also a 3 dimensional implementation of the same application. For the standard version we used arrays with 1024x1024 elements for the temperature and the power.

Streamcluster

Streamcluster solves the online clustering problem. For a stream of input points, it finds a pre-determined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances (SSQ) metric. The original code is from the Parsec Benchmark suite developed by Princeton University. We used 32768 data points per block and 1 block.

K-means

K-means is a clustering algorithm used extensively in data mining. This identifies related points by associating each data point with its nearest cluster, computing new cluster centroids, and iterating until convergence. We used 494020 objects with 34 attributes each.

LavaMD

The code calculates particle potential and relocation due to mutual forces between particles within a large 3D space. This space is divided into cubes, or large boxes, that are allocated to individual cluster nodes. The large box at each node is further divided into cubes, called boxes. 26 neighbor boxes surround each box (the home box). Home boxes at the boundaries of the particle space have fewer neighbors. Particles only interact with those other particles that are within a cutoff radius since ones at larger distances exert negligible forces. Thus the box size is chosen so that cutoff radius does not span beyond any neighbor box for any particle in a home box, thus limiting the reference space to a finite number of boxes. The space examined was divided into 8000 cubes each with dimensions 20x20x20.

Heartwall

The Heart Wall application tracks the movement of a mouse heart over a sequence of 30 (maximum 104) 744x656 ultrasound images to record response to the stimulus. In its initial stage, the program performs image processing operations on the first image to detect initial, partial shapes of inner and

outer heart walls. These operations include: edge detection, SRAD despeckling, morphological transformation and dilation. In order to reconstruct approximated full shapes of heart walls, the program generates ellipses that are superimposed over the image and sampled to mark points on the heart walls (Hough Search). In its final stage, program tracks movement of surfaces by detecting the movement of image areas under sample points as the shapes of the heart walls change throughout the sequence of images.

Myocyte

Myocyte application models cardiac myocyte (heart muscle cell) and simulates its behavior. The model integrates cardiac myocyte electrical activity with the calcineurin pathway, which is a key aspect of the development of heart failure. The model spans large number of temporal scales to reflect how changes in heart rate as observed during exercise or stress contribute to calcineurin pathway activation, which ultimately leads to the expression of numerous genes that remodel the heart's structure. It can be used to identify potential therapeutic targets that may be useful for the treatment of heart failure. Biochemical reactions, ion transport and electrical activity in the cell are modeled with 91 ordinary differential equations (ODEs) that are determined by more than 200 experimentally validated parameters. The model is simulated by solving this group of ODEs for a specified time interval. The process of ODE solving is based on the causal relationship between values of ODEs at different time steps, thus it is mostly sequential. At every dynamically determined time step, the solver evaluates the model consisting of a set of 91 ODEs and 480 supporting equations to determine behavior of the system at that particular time instance. If evaluation results are not within the expected tolerance at a given time step (usually as a result of incorrect determination of the time step), another calculation attempt is made at a modified (usually reduced) time step. Since the ODEs are stiff (exhibit fast rate of change within short time intervals), they need to be simulated at small time scales with an adaptive step size solver. The simulation time interval used is 30msec and the number of instances of simulation 228.

Speckle Reducing Anisotropic Diffusion(SRAD)

SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs). It is used to remove locally correlated noise, known as speckles, without destroying important image features. SRAD consists of several pieces of work: image extraction, continuous iterations over the image (preparation, reduction, statistics, computation 1 and computation 2) and image compression. The sequential dependency between all of these stages requires synchronization after each stage (because each stage operates on the entire image). The inputs to the program are ultrasound images and the value of each point in the computation domain depends on its four neighbors. The dimensions used were 502x458 over 1000 iterations. For the second version we used 6000x6000 image over 100 iterations.

Back Propagation

Back Propagation is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. The application is comprised of two phases: the Forward Phase, in which the activations are propagated from the input to the output layer, and the Backward Phase, in which the error between the observed and requested values in the output layer is propagated backwards to adjust the weights and bias values. In each layer, the processing of all the nodes can be done in parallel. The size of the layer used was 4,194,304.

Nearest Neighbors(NN)

NN finds the k-nearest neighbors from an unstructured data set. The sequential NN algorithm reads in one record at a time, calculates the Euclidean distance from the target latitude and longitude, and evaluates the k nearest neighbors. We looked for the k=8 nearest neighbors over 22,800,000 records.

Computational Fluid Dynamics(CFD)

The CFD solver is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow. The CFD solver is released with two versions: one with precomputed fluxes, and the other with redundant flux computations. Both versions used 97,000 elements.

The Table 3.5 shows all the Rodinia applications used characterized by their domain and their floating point operations. We calculated the floating point operations manually by scrutinizing the source code.

Application	Domain	MFlops
LUD	Linear Algebra	350,950.0
Hotspot	Physics Simulation	3,144.5
Hotspot3D	Physics Simulation	3,770.0
Streamcluster	Data Mining	1,716.0
K-means	Data Mining	63,492.0
LavaMD	Molecular Dynamics	14,720.0
Heartwall	Medical Imaging	175.9
Myocyte	Biological Simulation	2331.2
srad_v1	Image Processing	103,462.0
srad_v2	Image Processing	151,200.0
Back Propagation	Pattern Recognition	469.8
NN	Data Mining	182.4
CFD	Fluid Dynamics	157,347.4
pre-CFD	Fluid Dynamics	168,371.0

Table 3.5: Summary of Rodinia Applications

3.4.2 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications in the original ”pencil-and-paper” specification. The benchmark suite has been extended to include new benchmarks for unstructured adaptive mesh, parallel I/O, multi-zone applications, and computational grids. Problem sizes in NPB are predefined and indicated as different classes. Reference implementations of NPB are available in commonly-used programming models like MPI and OpenMP.

For our work, once again we used only benchmarks with floating point operations and profiled them against the tuning states defined in the previous section.

Block Tri-diagonal solver (BT)

BT is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating

Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are Block-Tridiagonal of 5x5 blocks and are solved sequentially along each dimension. The problem size used was class A.

Scalar Penta-diagonal solver (SP)

SP is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension. The problem size used was class A.

Lower-Upper Gauss-Seidel solver (LU)

LU is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems. The problem size used was class A.

Discrete 3D fast Fourier Transform (FT)

FT contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFT's, one for each dimension. The problem size used was class B.

Multi-Grid on a sequence of meshes (MG)

MG uses a V-cycle MultiGrid method to compute the solution of the 3-D scalar Poisson equation. The algorithm works continuously on a set of grids that are made between coarse and fine. It tests both short and long distance data movement. The problem size used was class C.

Conjugate Gradient (CG)

CG uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured grid computations and communications by using a matrix with randomly generated locations of entries. The problem size used was class B.

The Table 3.6 shows all the NAS Benchmarks used along with their class size and their total floating point operations.

3.5 Characterization of the Tuning Space

In this section we evaluate the tuning's space level of variance over the applications used. As a performance rate we use the (MFlops/sec)/Watt[3]. In particular, we compare how the different arguments of the tuning variables affect the performance of the applications. In addition, we prove that the chosen

[3] The power consumed is the maximum power reported by micsmc tool, for the whole coprocessor during the execution.

Benchmark	Class	MFlops
BT	A	168,300.0
SP	A	85,000.0
LU	A	119,280.0
FT	B	92,050.0
MG	C	155,700.0
CG	B	54,700.0

Table 3.6: Summary of NAS benchmarks.

tuning space is able to improve the performance of an application relative to a base configuration [4]. Hence, we are aiming for the configurations that correspond to a performance greater than 1.0.

The largest deviation can be observed in the configurations of working threads and their affinity. The Figure 3.5 depicts how different applications respond to variation of the previous tuning parameters. If we try to derive a general rule of cores, threads and affinity from our benchmark set, it appears difficult to choose a configuration with certainty, as Figures 3.5e and 3.5f show. That is an expected result because each application is affected differently from the configurations.

Following, we present the violin plots for the tuning parameters prefetch, unroll and optimization level. Especially for the latter one we noticed small variations. Figure 3.6 shows the performance gain or deterioration for two selected applications.

Concerning the prefetch parameter we notice that between the 3 levels of software prefetch (2,3 and 4) no obvious difference exists. Therefore, that parameter should me examined more thoroughly and precisely in the future.

Noticeable alterations also occur in the configuration of huge pages and streaming stores. Figure 3.7 shows that. Some configurations may not change dramatically the performance of an application as we observe in Figure 3.7c.

To compare with the base configuration, Figure 3.8 shows the distribution of the performance relative to the base for the different configuration parameters. We see that there are many variations and some parameters benefit applications positively and some negatively. For instance, the mean performance of the application *mg* (Figure 3.8f) when evaluated on the optimization level seems to be degraded. Yet, the 4th quartiles exceed the base performance and these are the configurations we need. For the *hotspot3D* (Figure 3.8d) the mean performance is constantly over 1.0 and under 1.2. In particular for the option 0 (disabled prefetches) the ratio exceeds 1.2. The performance gain can even reach $\times 2$ the base configurations, such as in Figure 3.8e. The applications that have been selected to illustrate the distributions are the ones that presented the most variation from the corresponding parameter.

Overall, we can argue that our benchmark set is characterized by adequate deviation in respect of the tuning space and can be used in our goal of extracting features characterized by our tuning parameters. We will explain in more detail our method in the next chapter.

[4] As a base configuration we define the execution of an application without any compiler optimization but using every available hardware thread for its parallel execution. The thread affinity by default is set to balanced.

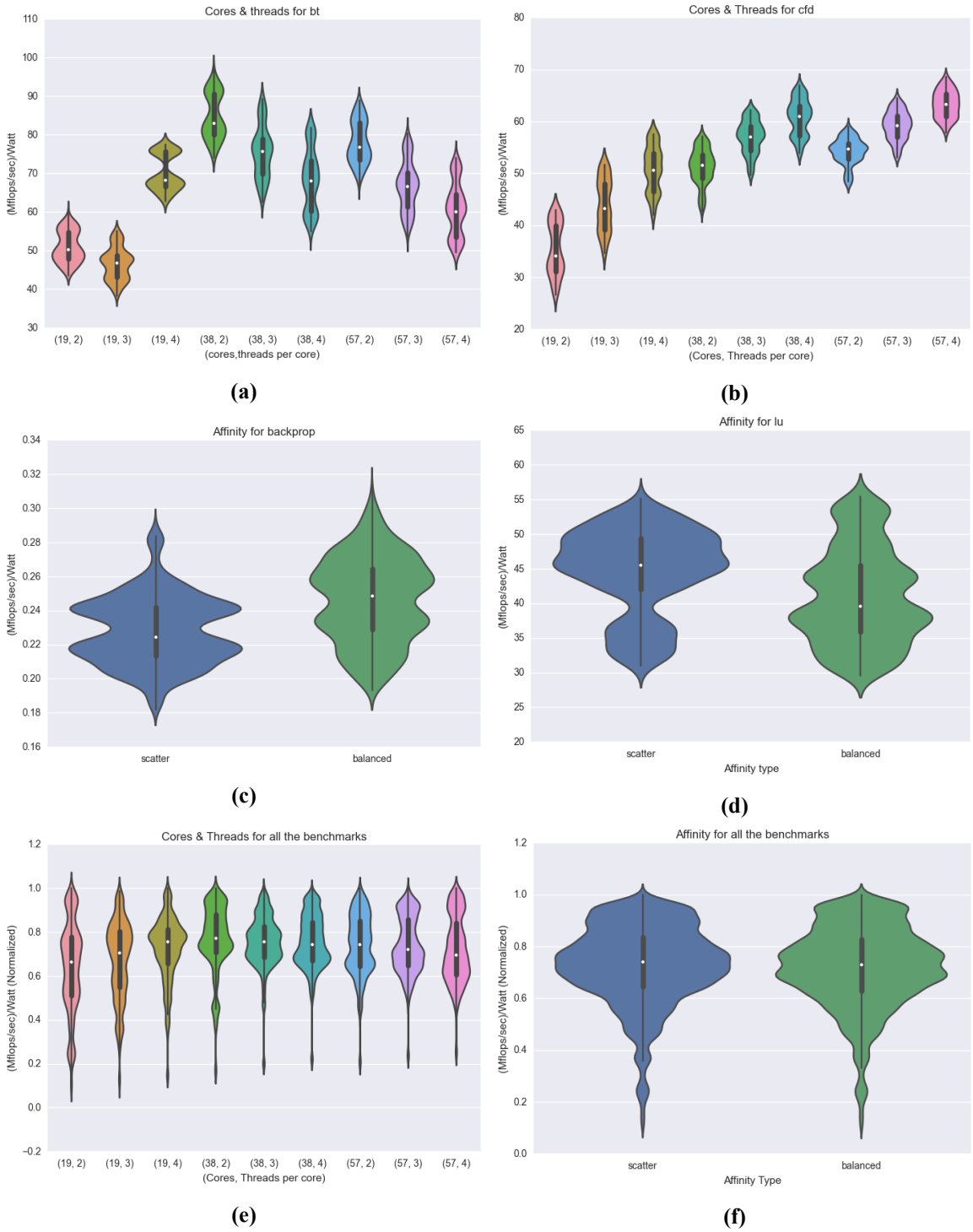


Figure 3.5: Violin plots for the affinity and cores-threads tuning parameters.

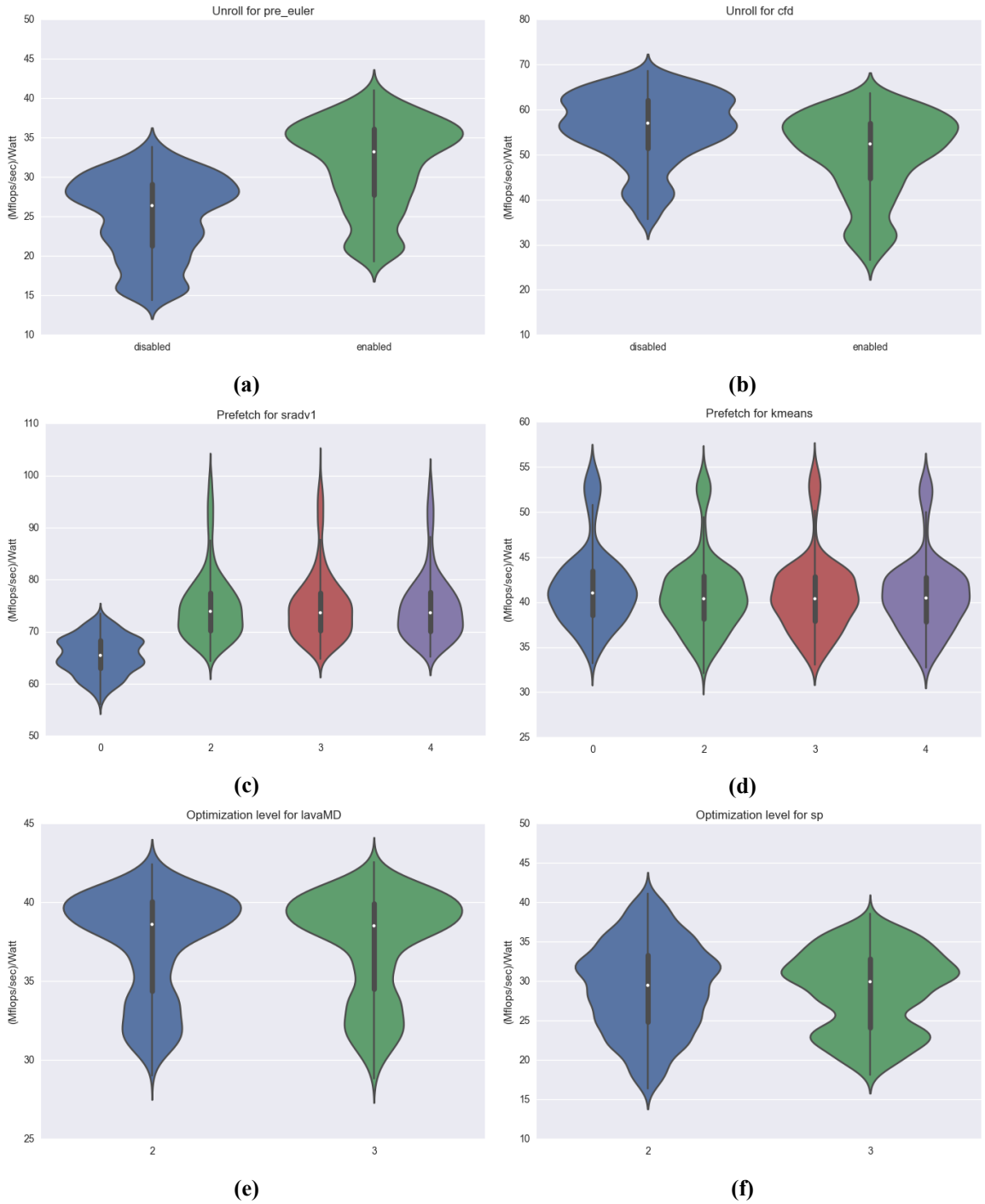


Figure 3.6: Violin plots for the prefetch, unroll and optimization level tuning parameters.

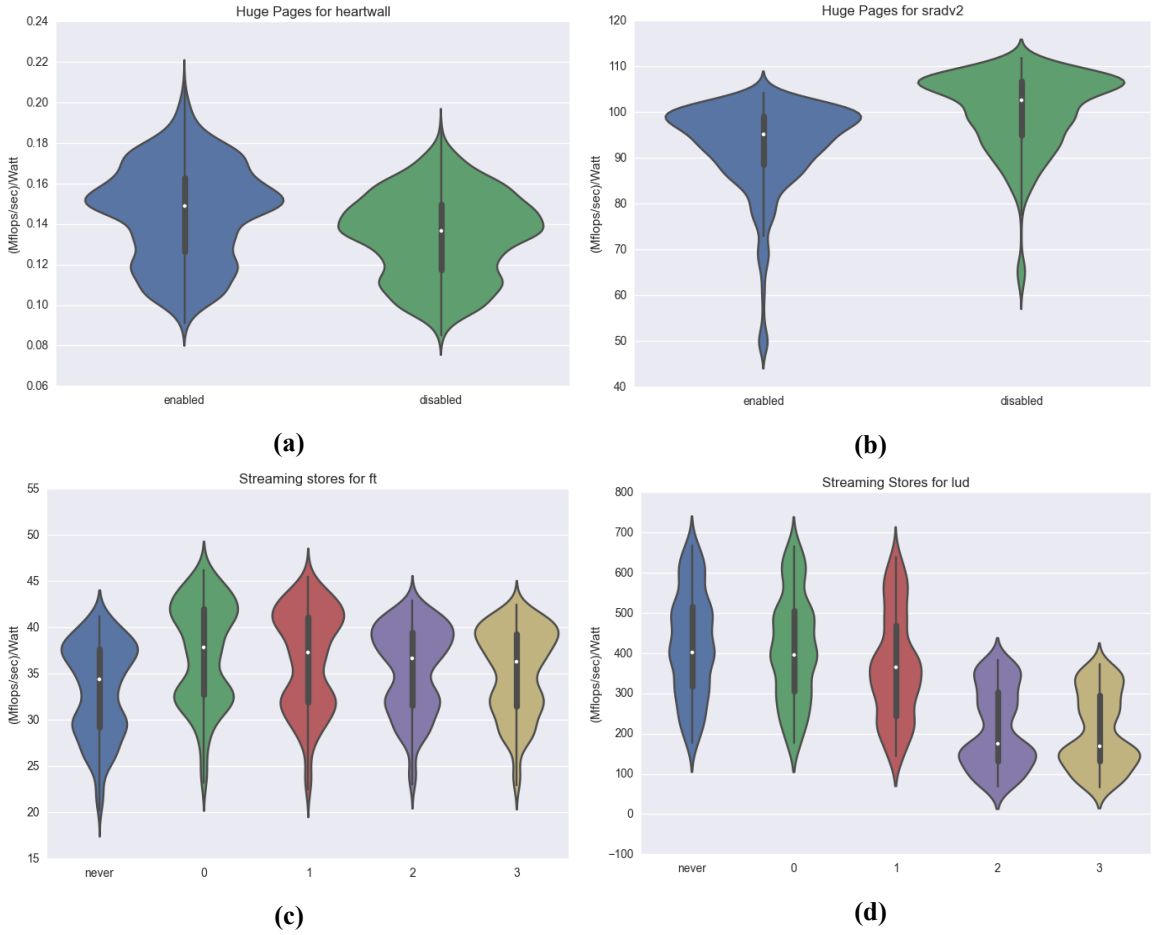


Figure 3.7: Violin plots for the huge pages and streaming stores tuning parameters.

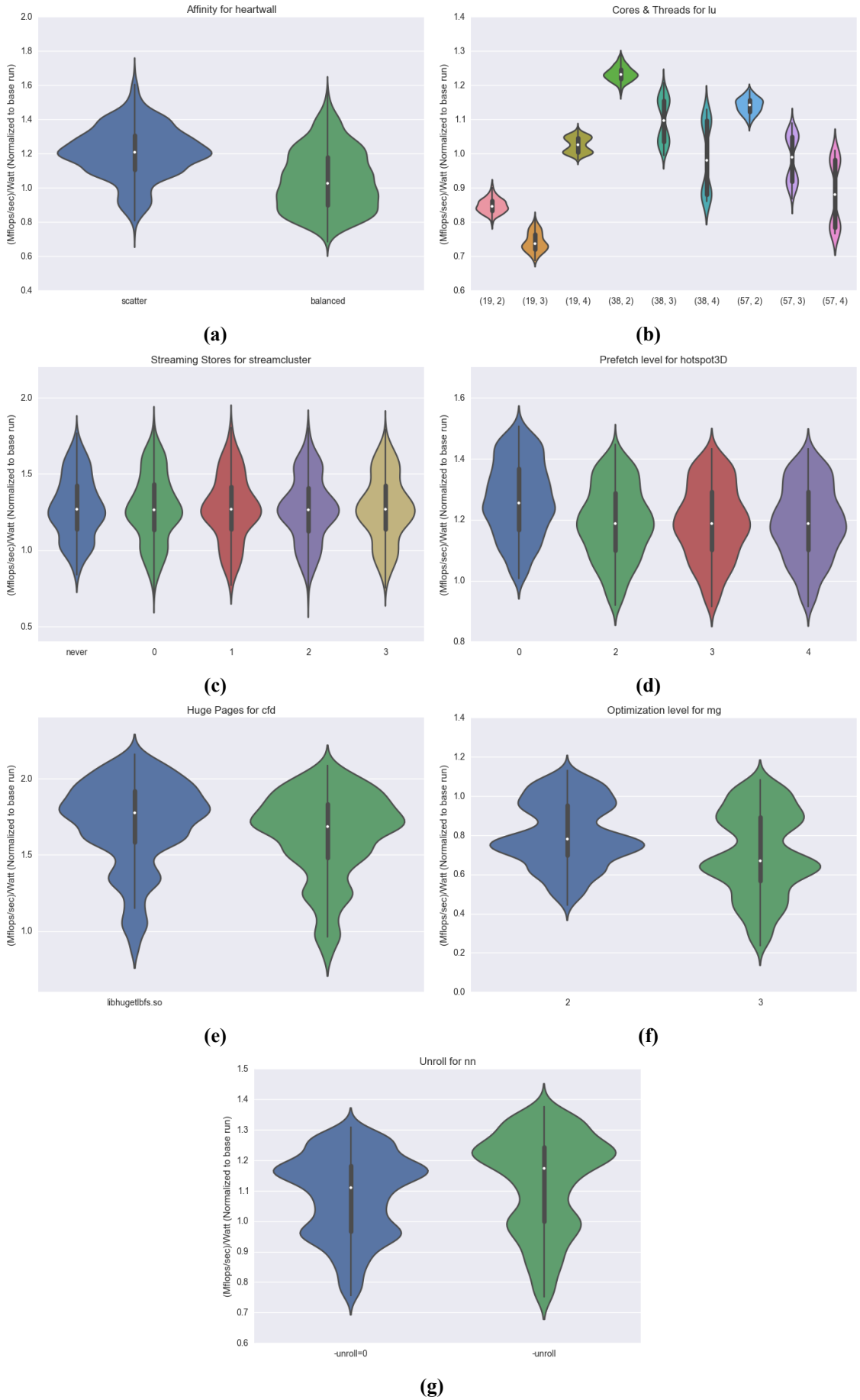


Figure 3.8: Violin plots for different tuning parameters relative to base configuration.

Chapter 4

The Autotuner: Background & Implementation

In this chapter, we present our autotuner. We answer the questions how it works and of what it is composed. The autotuner has two stages the offline and the online. During the offline stage the learning base is being built by profiling a set of representative applications against every configuration in our tuning space. That needs to happen only once and it is apparently the most time consuming part. Then, the autotuner is always online, meaning that he is able to provide us with a tuning recommendation for any application we query him, as long as the application has been profiled against a very small percentage of the tuning space ($\approx 10\%$). To derive its recommendations it uses a collaborative filtering technique based on matrix factorization, similar to the solutions given on the Netflix Prize competition[48].

In the following section, we define collaborative filtering in order to substantiate our methodology in the development of the autotuner. Then, we describe the offline and the online stages.

4.1 Collaborative Filtering

The world nowadays is overwhelmed by products for specific tastes and variety of needs, making a difficult choice for the consumers. Hence, an essential goal arises to match consumers with the most appropriate products. That is accomplished by developing recommender systems, which analyze patterns of user interest in items to provide personalized suggestion. Many services like Amazon, Netflix, Google and Yahoo want to enhance their customers experience by making recommender systems a rigid part of their businesses. An example can be found in the entertainment business where movies and TV shows are rated by each user. Their ratings expose a trend and a personal taste which can be processed in order to suggest them unrated movies with high possibility of success.

Recommendation systems use a number of different technologies. We can classify these systems into two broad groups[50].

- *Content-based systems* examine properties of the user or the item to characterize its nature. For example, a user profile could include demographic information, his favorite actor and genre, whereas a movie profile could include attributes regarding its genre, the participating actors, its box office popularity, and so forth. Of course, content-based systems require explicit information that might not be available or easy to collect.
- *Collaborative filtering systems* analyze relationships between users and interdependencies among products to identify new user-item associations. Based on these associations they make their recommendations. Their advantage over content filtering is they do not require domain knowledge and avoid the need for extensive data collection. In addition, relying directly on user behavior

allows uncovering complex and unexpected patterns that would be difficult or impossible to profile using known data attributes. Lastly, they rely only on past user behavior.

Collaborative filtering (CF) has been a very popular approach in the past decade to recommendation systems. Especially from 2006, when Netflix announced their contest to find a more accurate movie recommendation system than their current, Cinematch, and from 2009 when the grand prize was awarded to team ”Bellkor’s Pragmatic Chaos” for developing a system which improved by 10% the root mean square error (RMSE) of the recommendations compared with its predecessor.

To deepen into the case, CF has two primary areas: *neighborhood methods* and *latent factor models* [48].

The neighborhood methods are centered on computing the relationships between items or, alternatively, between users. For a user oriented approach, the user’s rating for a particular item is calculated based on the similar users’ ratings for that item. Similar (or neighbor) users are other users that tend to give the same ratings to the same items. In a sense, these methods transform items to the user space by viewing them as baskets with assigned users. This way, we no longer need to compare users to items, but rather directly relate users to users. It is important to realize that the same applies to items also and that is called the duality of the similarity[50]. To measure similarity many functions are used such us the Jaccard Distance, the Cosine Distance, the Pearson Distance and more.

The latent factor models, such as Singular Value Decomposition (SVD), comprise an alternative approach by transforming both items and users to the same latent factor space (20 to 100 factors inferred from the ratings pattern), thus making them directly comparable. From another perspective, these factors comprise a computerized alternative to the human created movie attributes in the content-based systems. For example, when the products are movies, factors might measure obvious dimensions such as comedy vs. drama, amount of action, or orientation to children; less well defined dimensions such as depth of character development or “quirkiness”; or completely uninterpretable dimensions. For users, each factor measures how much the user likes movies that score high on the corresponding movie factor.

Figure 4.1 illustrates the latent factor approach, which characterizes both users and movies using two axes. Male vs. Female and Serious vs. Escapist. For this model, similarity between user-movie, user-user and movie-movie can be reckoned by their dot product. For instance, Gus we expect to give high rating to *Dump and Dumper* and to *Independence Day* because their cosine is close to 1. In contrast, Gus will not like at all *The Color Purple*. In addition, Jane and Peter have a cosine close to 1 hence, they will have many similar ratings.

If we change users with applications and items with tuning configurations then we can map our problem of autotuning to the one of suggesting movies to users. In our approach we use a latent factor model, an instance of SVD, which we present in the following section.

4.1.1 The Latent Factor Model

Our model is a combination of two:

a. Baseline predictor

b. Matrix Factorization

For the following we define as $\mathcal{K}$ the set of the known ratings, r_{ui} the rating of user u for the item i.

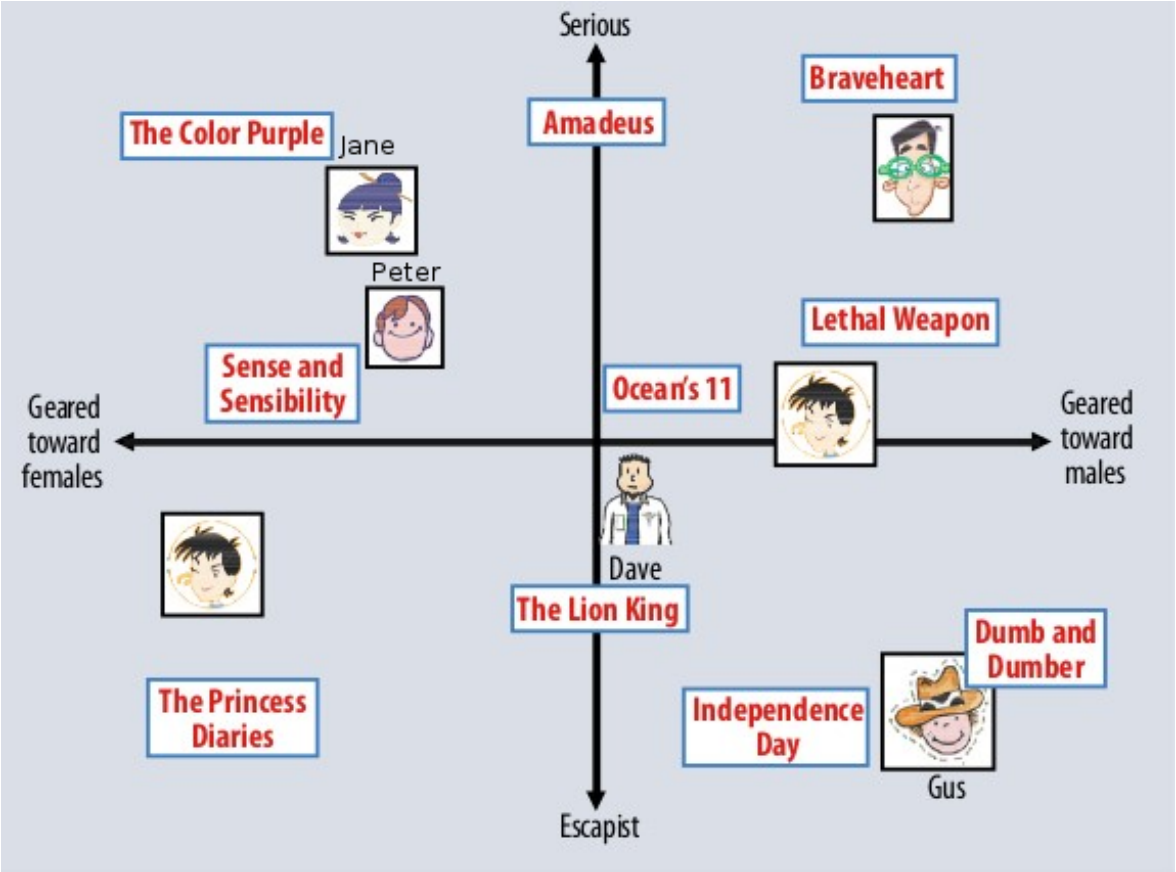


Figure 4.1: Fictitious latent factor illustration for users and movies[48].

Baseline Predictor

It has been noted that much of the observed rating values are due to effects associated with either users or items independently of their interaction, thus standard CF models are unable to capture the true interactions between them. That is explained by the fact that CF data exhibit large user and item biases, i.e. systematic tendencies for some users to rate higher than others, and for some items to receive lower ratings than others. Baseline predictors filter these biases, and leave the part of the signal that truly represents user-item interaction.

A baseline prediction for an unknown rating r_{ui} is denoted by b_{ui} and accounts for the user and item effects:

$$b_{ui} = \mu + b_u + b_i$$

where μ denotes the global average rating. The parameters b_u, b_i measure the observed variations of user u and item i from the average.

An example from users movies, suppose we want a baseline prediction for the rating of the movie Independence Day by user Dave. Now, say that the average rating over all movies is $\mu = 3.4$. On the one hand, Independence Day is better than an average movie, so it tends to be rated 0.6 stars above the average. On the other hand, Dave is a generous user, who tends to rate 0.3 stars higher than the average. Thus, the baseline predictor for Independence Day's rating by Dave would be $3.4 + 0.6 + 0.3 = 4.3$. The biases b_u, b_i can be calculated by solving the least squares problem:

$$\min_{b*} \sum_{(u,i)\in\mathcal{K}} (r_{ui} - \mu - b_u - b_i)^2 + \lambda(\sum_u b_u^2 + \sum_i b_i^2)$$

Here, the first term $\sum_{(u,i)\in\mathcal{K}}(r_{ui}-\mu-b_u-b_i)^2$ strives to find b_u's and b_i's that fit the given ratings. The regularized term $\lambda(\sum_u b_u^2+\sum_i b_i^2)$ avoids overfitting by penalizing the magnitudes of the parameters. This least square problem can be solved fairly efficiently by the method of stochastic gradient descent.

Matrix Factorization

Matrix factorization is the cause behind the most successful realizations of latent factor models. It is able to map both users and items to a joint latent factor space of dimensionality *k*, such that user-item interactions are modeled as inner product in that space. Accordingly, each item i is associated with a vector $q_i \in \mathbb{R}^k$, and each user u is associated with a vector $p_u \in \mathbb{R}^k$. For a given item i the elements of the q_i measure the extent to which the item possesses those *k* factors. A positive value denotes the item is completely characterized by the corresponding factor, whereas a negative value the exact opposite. For a given user u the elements of the p_u measure the extent to which the user is interested towards those factors. Again a positive values means he likes that factor, while on the contrary a negative value means he dislike it. The resulting dot product, $p_u q_i^T$, captures the interaction between user u and item i, the user's overall interest in the item's characteristics. This approximates user's u rating of item i, which is denoted by r_{ui}, leading to the estimate[48]:

$$\hat{r}_{ui} = p_u q_i^T$$

The major challenge is computing the mapping of each user and item to their factor vectors. After the recommender system completes this mapping, it is relative easy to estimate the rating a user will give to any item.

By examining that model, we can relate it to singular value decomposition (SVD), a well established technique for identifying latent semantic factors in information retrieval. However, applying SVD to explicit ratings in the CF domain raises difficulties due to the sparseness of the user-item rating matrix. It needs a complete matrix in order to produce legit results. Moreover, carelessly addressing only the relatively few known entries is highly prone to overfitting. Earlier works relied on imputation to make the rating matrix dense. Yet, it is proven very expensive as it significantly increases the amount of data. In addition, inaccurate imputation might distort the data considerably. Hence, processing only the observed ratings and building directly on those the model, while avoiding overfitting, is considered a reliable approach.

A recent enlightening example, is the Netflix challenge. The data consisted of approximately 500K users and 17K movies producing a total of 8.5 billions ratings. However, of those total ratings only around 100M where known. There were missing 98.8% of the values. Hence, any imputation to make the matrix dense would be based on a minute proportion of the total ratings, and probably falsifying the data. In addition, a complete matrix would need 34 GBytes (supposing 4 Bytes per rating) which is a huge amount of memory and very time consuming to process. And lastly, computing the SVD of a immense matrix is nearly impossible and excessively demanding. Instead, if there exists a map of the users and the movies in the latent factor space of dimensionality k=40 for instance, the storage requirements and the computational work are lesser (matrices 500Kx40 and 17Kx40 total 82.72 MBytes).

Back to the definitions, in order to calculate the factor vectors p_u and q_i, the system minimizes the regularized squared error on the set of known ratings[48]:

$$\min_{p^*,q^*} \sum_{(u,i)\in\mathcal{K}} (r_{ui} - p_u q_i^T)^2 + \lambda(\|p_u\|^2 + \|q_i\|^2)$$

The system learns the model by fitting the previously observed ratings. However, the goal is to generalize those previous ratings in a way that predicts future, unknown ratings. Thus, the system should avoid

overfitting the observed data by regularizing the learned parameters, whose magnitudes are penalized. The constant λ controls the extent of regularization and is usually determined by cross-validation. The minimization is typically performed by either stochastic gradient descent or alternating least squares.

Our Model

By combining the two previous models, we benefit from both approaches. We model in parallel both biases and true interactions between users and items. So, our model is comprised of four components, as follows:

$$\hat{r}_{ui} = \mu + b_u + b_i + p_u q_i^T \tag{4.1}$$

The system learns by minimizing the squared error function (or loss function):

$$\min_{p^*,q^*,b^*} L = \sum_{u,i \in \mathcal{K}} (r_{ui} - \mu - b_u - b_i - p_u q_i^T)^2 + \lambda_1 \sum_u \|p_u\|^2 + \lambda_2 \sum_i \|q_i\|^2 + \lambda_3 \sum_u b_u^2 + \lambda_4 \sum_i b_i^2 \tag{4.2}$$

In our work, for each learned parameter we used different regulators $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ in order to achieve better accuracy. In addition, to minimize the regularized squared error we used stochastic gradient descent which we explain in the next section.

4.1.2 Stochastic Gradient Descent

For each given rating, the system computes the corresponding prediction error:

$$e_{ui} = r_{ui} - \hat{r}_{ui} = r_{ui} - (\mu + b_u + b_i + p_u q_i^T) \tag{4.3}$$

According to stochastic gradient descent, the current estimate of the error function (4.3) is updated by one training example at a time. So, for each training example, we take the derivative of the loss function with respect to each parameter and adjust the latter according to the following iterative formula:

$$x_{k+1} \leftarrow x_k - \eta \frac{\partial L}{\partial x_k}$$

where η is the learning rate which leverages how much our update modifies the feature weights and is unique for each parameter.

Let's calculate for instance the derivative with respect to p_u:

$$\frac{\partial L}{\partial p_u} = 2(r_{ui} - \mu - b_u - b_i - p_u q_i^T)(-q_i) + 2\lambda_1 p_u$$

$$\frac{\partial L}{\partial p_u} = -2(e_{ui} q_i + \lambda_1 p_u)$$

Hence, the corresponding formula becomes:

$$p_u \leftarrow p_u - \eta_1 \frac{\partial L}{\partial p_u}$$

$$p_u \leftarrow p_u + \eta_1 (e_{ui} q_i - \lambda_1 p_u) \tag{4.4}$$

Accordingly, for the rest parameters, the formulas end up being:

$$q_i \leftarrow q_i + \eta_2 (e_{ui} p_u - \lambda_2 q_i) \tag{4.5}$$

$$b_u \leftarrow b_u + \eta_3 (e_{ui} - \lambda_3 b_u) \tag{4.6}$$

$$b_i \leftarrow b_i + \eta_4 (e_{ui} - \lambda_4 b_i) \tag{4.7}$$

The updates continue until there is no meaningful change in the parameters after two consecutive iterations or until we reach a predefined number of iterations. We need to note that the learning rates $\eta_j, j = 1, 2, 3, 4$ and the regulators $\lambda_j, j = 1, 2, 3, 4$ are obtained by grid exploration.

By having established the analytical background of our methodology, we proceed with the description of the autotuner.

4.2 Offline Stage

4.2.1 Structure

In this stage, the learning base is constructed in order to be used later. An application is evaluated against every tuning configuration by extracting metrics from the hardware performance counters while it is executed. Towards that goal, we need to use the performance counters the coprocessor provides and calculate Cycles per Instruction (CPI) for every thread, bandwidth between the CPUs and the main memory, vectorization, power consumed and execution time.

A tool that can easily probe the performance counters with minute latency is Likwid[62], a performance monitor tool for the GNU Linux operating system.

So, for an application we establish the tuning state (compiler's flags and execution environment) and run the executable inside *likwid-perfctr*, a command line tool from the likwid suite, for simple end-to-end measurements that can be used as an application wrapper. With that command line tool we define the event set which consists of that many events as there are physical counters on a given CPU, in our case two. Because we need to take measurements from seven hardware events, we run the executable four times.

The hardware events which are monitored are:

- **INSTRUCTION_EXECUTED:** Counts the number of instructions executed by a hardware thread.
- **CPU_CLK_UNHALTED:** The number of cycles (commonly known as clockticks) where any thread on a core is active. A core is active if any thread on that core is not halted. This event is counted at the core level at any given time, all the hardware threads running on the same core will have the same value.
- **L2_DATA_READ_MISS_MEM_FILL:** Counts data loads that missed the local L2 cache, and were serviced from memory (on the same Intel Xeon Phi coprocessor). This event counts at the hardware thread level. It includes L2 prefetches that missed the local L2 cache and so is not useful for determining demand cache fills or standard metrics like L2 Hit/Miss Rate.
- **L2_DATA_WRITE_MISS_MEM_FILL:** Counts data Reads for Ownership (due to a store operation) that missed the local L2 cache, and were serviced from memory (on the same Intel Xeon Phi coprocessor). This event counts at the hardware thread level.
- **DATA_CACHE_LINES_WRITTEN_BACK:** Number of dirty lines (all) that are written back, regardless of the cause.
- **VPU_ELEMENTS_ACTIVE:** Increments by 1 for every element to which an executed VPU instruction applies. For example, if a VPU instruction executes with a mask register containing 1, it applies to only one element and so this event increments by 1. If a VPU instruction executes with a mask register containing 0xFF, this event is incremented by 8. Counts at the hardware thread level.

- **VPU_INSTRUCTIONS_EXECUTED:** Counts the number of VPU instructions executed by a hardware thread. It is a subset of INSTRUCTIONS_EXECUTED.

Specifically for the execution time, the value from the time stamp counter (TSC) is taken, and it reports the wall clock time. Each physical core has a TSC for all the hardware threads which counts cycles while the core is in the C0 state. While we monitor only the performance counters of the threads we have enabled, meaning that the assigned cores are always in C0 state, the TSC increments based on the active core frequency. Hence, we argue that it is a rational way of measuring the wall clock time.

After collecting the hardware events we need to pre-process them and reckon the metrics, before the learning base is populated. Below, we present the formulas used, where the values are the average ones from the threads used.

- **CPI:**

$$CPI_{thread} = \frac{\text{CPU\_CLK\_UNHALTED}}{\text{INSTRUCTION\_EXECUTED}}, CPI_{core} = \frac{CPI_{thread}}{\text{\#threads per core}}$$

- **BW:**

$$\begin{aligned}\text{Bytes transferred} = (&\text{L2\_DATA\_READ\_MISS\_MEM\_FILL} \\ &+ \text{L2\_DATA\_WRITE\_MISS\_MEM\_FILL} \\ &+ \text{DATA\_CACHE\_LINES\_WRITTEN\_BACK}) * 64Bytes\end{aligned}$$

$$BW = \frac{\text{Bytes transferred}}{\text{Execution Time}}$$

- **Vectorization:**

$$Vectorization = \frac{\text{VPU\_ELEMENTS\_ACTIVE}}{\text{VPU\_INSTRUCTIONS\_EXECUTED}}$$

Lastly, the power is extracted by using *micsmc* utility developed by Intel. It uses the Symmetric Communications Interface (SCIF), the capabilities designed into the coprocessor OS, and the host driver to deliver the coprocessor's status. That method is called "in-band". *Micsmc* reports the 1-second moving average of the total power that is being consumed on the coprocessor at any given time, in watts. This form of power analysis is chosen because it is thermally relevant. This averaging provides correlation to real-world measureable thermal events. For example, if we only measure instantaneous power, we may see spikes in power levels for very short durations that will not have measureable impacts to the silicon temperature on the heat sink or other thermal solution[67].

The result is a comma separated values (csv) file for each application, in the form:

Configuration, CPI, BW, Vectorization, Power, Time

where configuration is the set of the tuning parameters that produced the corresponding metrics. The file has 2880 lines, as many configurations in our tuning space.

4.2.2 The Composition of the Learning Base

A very important step to build our autotuner is to choose the applications that will compose the learning base. An application that will be part of the base needs to be different from the already existing ones and exhibit, by itself or as a member of a small number of similar applications, a set of features than

can represent a larger group of applications. In other words, we need the learning base to be diverse so that any new incoming application has as many similarities as possible with the others in the base.

One way to achieve this is to factorize all of our set. The applications are mapped to the latent factor space and can be projected into a 2-dimensional space (alternatively 3-dimensional) in respect of the 2 (or 3) major latent factors, which are the most descriptive dimensions for applications and tuning configurations.

The factorization is based on the Singular Value Decomposition (SVD). Through SVD, a matrix $A : n \times m$ can be written in the form:

$$A = U\Sigma V^T$$

where:

1. r is the rank of A. We refer to this value as the dimension of the latent factor space.
2. U is a $n \times r$ column-orthonormal matrix; that is, each of its columns is a unit vector and the dot product of any two columns is 0.
3. Σ is a $r \times r$ diagonal matrix. The elements of Σ are called singular values of A and they appear in decreasing order of magnitude.
4. V is a $m \times r$ column-orthonormal matrix. V^T is the transposed form.

In order to create the application and the configurations latent factor vectors, we can use the products of SVD in the following way:

$$p_u = U\sqrt{\Sigma}$$
$$q_i = V\sqrt{\Sigma}^T$$

So, p_u measures the correlation between application u and each feature and q_i measures the correlation between tuning configuration and each feature. Note however, this method is validated only when the initial matrix A is complete. Unless A is full, not legit correlation factors can be produced.

Let's examine the Netflix example which is more intuitive. Figure 4.2 shows the first two factors from the Netflix data matrix factorization. Movies are placed according to their factor vector (q_i). By observing the movies shown we can easily derive the meaning of the factor vectors. The first one (x-axis) has on the one side lightweight comedies and horror movies aimed at a male or adolescent audience (*Road Trip, Freddy vs. Jason*), while the other side contains drama or comedy with serious undertones and strong female leads (*Sophie's Choice, Moonstruck*). The second factor vector (y-axis) has independent, critically acclaimed, quirky films (*Punch-Drunk Love, Being John Malkovich*) on the top, and on the bottom, mainstream formulaic films (Armageddon, The Fast and the Furious). Appealing to all types seems to be *The Wizard of Oz*, right in the middle. Moreover there are interesting intersections in the corners. On the top left corner, *Kill Bill:Vol. 1* meets *Natural Born Killer*, both arty movies that play off violent themes. On the bottom right, *The Sound of Music* meets *Sister Act*, two serious female-driven movies and mainstream crowd-pleasers[48].

Based on the previous observations, we came up with the idea that if we fully know the ratings of some movies then we will be able to produce the ratings of other similar movies, partially unknown, rather easily as they have near factor vectors. So, in the Figure 4.2 we may try and group some movies and choose one or two to represent a neighborhood. For instance, consider the group *The Longest Yard, The Fast and the Furius, Armageddon, Catwoman, Coyote Ugly* and that we choose as representatives the movies *The Fast and the Furius and Catwoman*. Then, for the rest of the movies as well as for any other that it would happen to have the same features (mainstream formulaic and lightweight films, e.g. Transporter, Batman), their full rating vector would be approximated with a small error, even with

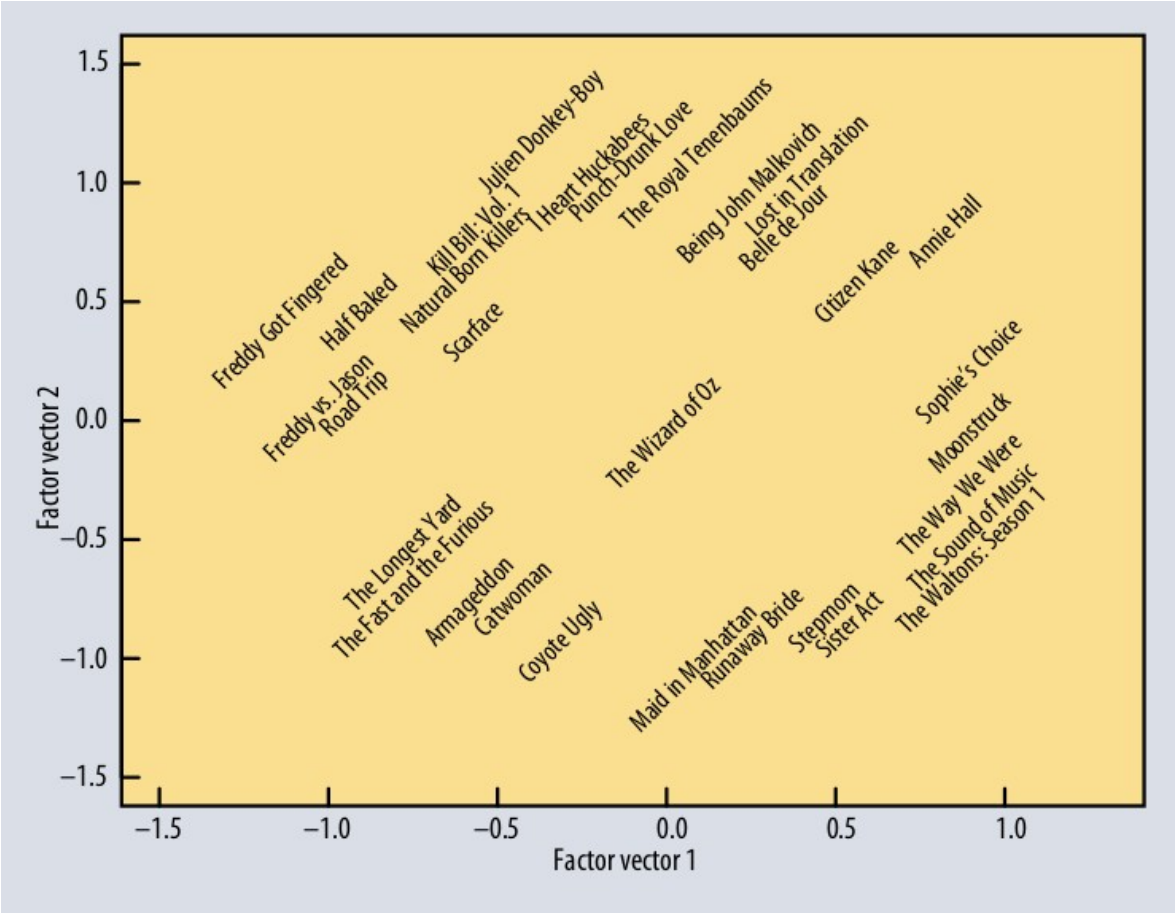


Figure 4.2: The first two features from a matrix decomposition of the Netflix Prize data[48].

very few ratings known. Hence, with high certainty we would provide ratings for partially unknown movies.

We should note however that the 2-dimensional space is able to mislead us, as it hides differences between items which are separated by their 3th, 4th, …, Rth feature and present them next to each other. For that reason we will examine our autotuner in respect of the learning base also.

Another way to create our learning base, more analytically substantiated, is by finding the correlations between the applications in the latent factor space using a similarity function. A widely used function is the Pearson's correlation and that is because of its attribute being invariant to adding and scaling. If we have one dataset $\{x_1, \ldots, x_n\}$ containing n values and another dataset $\{y_1, \ldots, y_n\}$ containing n values then the Pearson's correlation coefficient is defined as:

$$corr_{x,y} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and analogously for $\bar{y}$.

Based on the coefficients, we may divide the set of applications into a training and a test set. We define the similarity threshold at the 0.80.

The second method works best when we have a large enough latent factor space, and consequently the 2-dimensional projection hides a large proportion of the information for each application. Later, in our analysis we use both methods to derive a learning base.

Back to our problem of applications-configurations, we define two ratings of an application (u) for a

configuration (i) as:

$$r_1^{ui} = \frac{MFlops/sec}{Watts} \text{ (Normalized to the base run)} \tag{4.8}$$

$$r_2^{ui} = \frac{IPC}{Watts} \tag{4.9}$$

For the r_1^{ui}, we define as the base rating the rating achieved when the application runs on the co-processor with no compiler optimizations and using every hardware thread available with balanced affinity.

We create therefore two matrices A_1, A_2 of size NxM where $N = 20$ is the number of the applications and $M = 2880$ is the number of the total tuning configurations. By applying the matrix factorization (SVD) to both matrices we map the applications and the configurations to a joint latent factor space from which we are able to create a 2-dimensional graph with respect to the two major latent factors. Figures 4.3 and 4.5 present the first two latent vectors of our data for the applications.

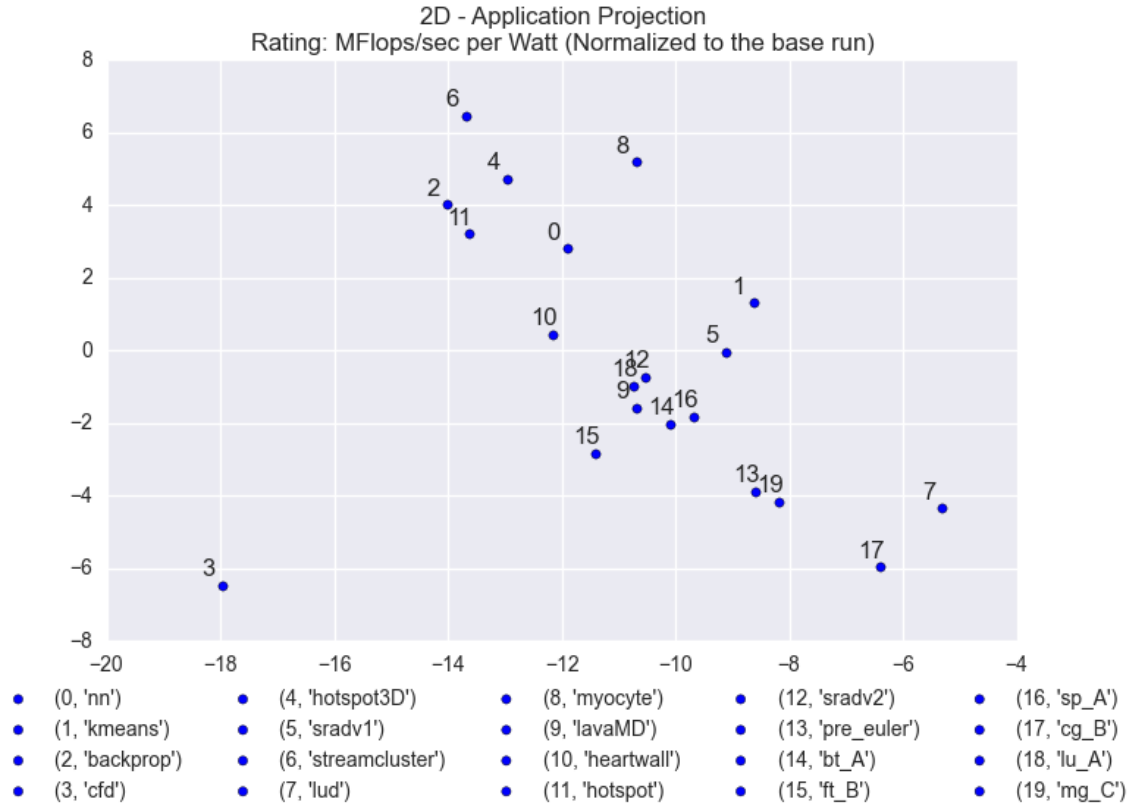


Figure 4.3: Two-dimensional graph of the applications, rating MFlops/sec/Watts (Normalized to the base rating).

Considering the 2-dimensional latent factor space (Figure 4.3) which was created based on the rating 1, we can divide the applications into a training and a test set (Table 4.1). (Note that this is a typical neighboring as we will variate our learning base later)

However, from the 3-dimensional projection we see the groups are ambiguous. Figure 4.4 shows that. For instance, it seems that *myocyte* is close to *streamcluster* and *backprop* to *kmeans*. We need to substantiate our composition of learning base and towards that goal we will use the second method.

Using the Pearson's similarity function 4.2.2, the following coefficient matrix is calculated, Table 4.2. We choose the neighbor threshold at 0.80 and with boldface we annotate the applications that have

Training set	Test Set
cfd, pre_euler	backprop
streamcluster,lavaMD	lu
hotspot3D, kmeans	bt
hotspot, lud	bt
nn, ft	sp
myocyte, cg	mg
sradv1, sradv2	

Table 4.1: Table with the training and test sets from the 2D neighboring.

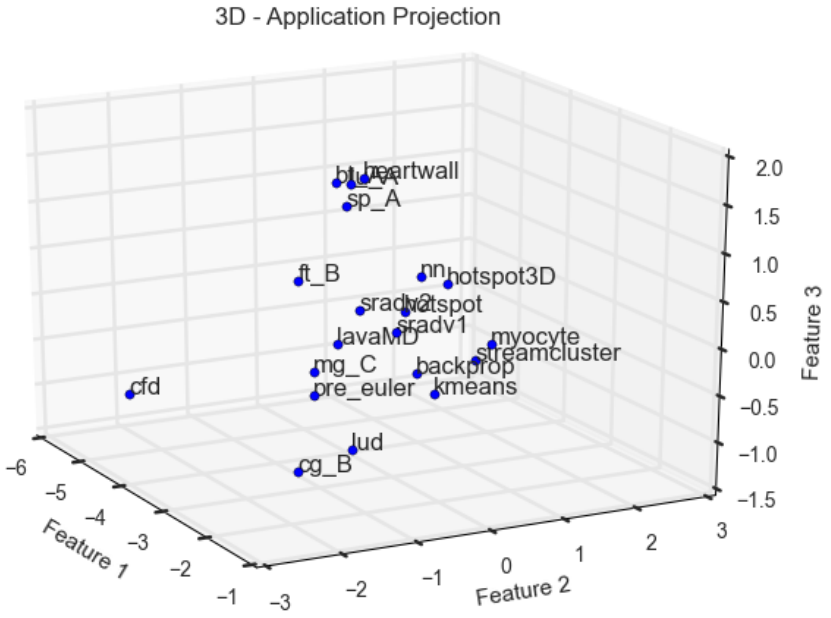


Figure 4.4: Three-dimensional graph of the applications, rating MFlops/sec/Watts (Normalized to the base rating).

similarity greater than or equal to the threshold. That neighboring is more rational and we expect to get more satisfying and accurate predictions. The training and the test sets are composed as:

	nn	kmeans	BP	cfd	HS3D	sradv1	SC	lud	myocyte	lavaMD	HW	HS	sradv2	pre_euler	bt	ft	sp	cg	lu	mg
nn	1.00	0.64	0.73	0.63	0.75	0.79	0.68	0.29	0.73	0.71	0.60	0.75	0.74	0.53	0.67	0.71	0.70	0.40	0.65	0.50
kmeans	0.64	1.00	0.69	0.65	0.73	0.63	0.66	0.28	0.69	0.69	0.59	0.66	0.67	0.53	0.53	0.60	0.55	0.42	0.59	0.47
BP	0.73	0.69	1.00	0.68	**0.83**	0.72	0.77	0.29	0.78	0.78	0.63	0.76	0.75	0.56	0.59	0.67	0.61	0.38	0.66	0.51
cfd	0.63	0.65	0.68	1.00	0.65	0.73	0.61	0.36	0.62	**0.84**	0.67	0.67	0.76	0.66	0.66	0.77	0.71	0.75	0.71	0.65
HS3D	0.75	0.73	**0.83**	0.65	1.00	0.72	**0.80**	0.33	**0.80**	0.78	0.73	**0.81**	0.74	0.54	0.64	0.72	0.64	0.35	0.75	0.51
sradv1	0.79	0.63	0.72	0.73	0.72	1.00	0.67	0.44	0.67	0.77	0.63	0.73	0.77	0.63	0.69	0.73	0.71	0.49	0.71	0.62
SC	0.68	0.66	0.77	0.61	**0.80**	0.67	1.00	0.22	**0.81**	0.66	0.64	0.71	0.65	0.46	0.55	0.56	0.58	0.25	0.60	0.41
lud	0.29	0.28	0.29	0.36	0.33	0.44	0.22	1.00	0.33	0.41	0.27	0.24	0.44	0.49	0.38	0.29	0.37	0.30	0.30	0.60
myocyte	0.73	0.69	0.78	0.62	**0.80**	0.67	**0.81**	0.33	1.00	0.68	0.62	0.76	0.64	0.50	0.59	0.60	0.59	0.36	0.64	0.43
lavaMD	0.71	0.69	0.78	**0.84**	0.78	0.77	0.66	0.41	0.68	1.00	0.70	0.74	**0.85**	0.73	0.74	**0.82**	0.74	0.62	0.78	0.63
HW	0.60	0.59	0.63	0.67	0.73	0.63	0.64	0.27	0.62	0.70	1.00	0.68	0.66	0.54	0.69	0.72	0.68	0.37	0.75	0.53
HS	0.75	0.66	0.76	0.67	**0.81**	0.73	0.71	0.24	0.76	0.74	0.68	1.00	0.74	0.54	0.62	0.64	0.65	0.33	0.67	0.50
sradv2	0.74	0.67	0.75	0.76	0.74	0.77	0.65	0.44	0.64	**0.85**	0.66	0.74	1.00	0.67	0.72	**0.83**	0.71	0.61	0.75	0.60
pre_euler	0.53	0.53	0.56	0.66	0.54	0.63	0.46	0.49	0.50	0.73	0.54	0.54	0.67	1.00	0.65	0.63	0.60	0.61	0.57	0.50
bt	0.67	0.53	0.59	0.66	0.64	0.69	0.55	0.38	0.59	0.74	0.69	0.62	0.72	0.65	1.00	0.72	**0.88**	0.43	**0.88**	0.55
ft	0.71	0.60	0.67	0.77	0.72	0.73	0.56	0.29	0.60	**0.82**	0.72	0.64	**0.83**	0.63	0.72	1.00	0.75	0.52	0.75	0.65
sp	0.70	0.55	0.61	0.71	0.64	0.71	0.58	0.37	0.59	0.74	0.68	0.65	0.71	0.60	**0.88**	0.75	1.00	0.50	**0.86**	0.57
cg	0.40	0.42	0.38	0.75	0.35	0.49	0.25	0.30	0.36	0.62	0.37	0.33	0.61	0.61	0.43	0.52	0.50	1.00	0.41	0.49
lu	0.65	0.59	0.66	0.71	0.75	0.71	0.60	0.30	0.64	0.78	0.75	0.67	0.75	0.57	**0.88**	0.75	**0.86**	0.41	1.00	0.57
mg	0.50	0.47	0.51	0.65	0.51	0.62	0.41	0.60	0.43	0.63	0.53	0.50	0.60	0.50	0.55	0.65	0.57	0.49	0.57	1.00

Table 4.2: Coefficient matrix for all the applications when rated by 1

Training set	Test Set
nn, kmeans, hotspot	hotspot3D
backprop, cfd	lavaMD
sradv1, sradv2	myocyte
streamcluster, lud	ft
cfd, pre_euler	sp
bt,cg, mg	lu

Table 4.3: Table with the training and test sets from the coefficient matrix 4.2.

Accordingly, from the Figure 4.5, we can create application neighbors, when rated by 2: (similarly this is a typical neighboring)

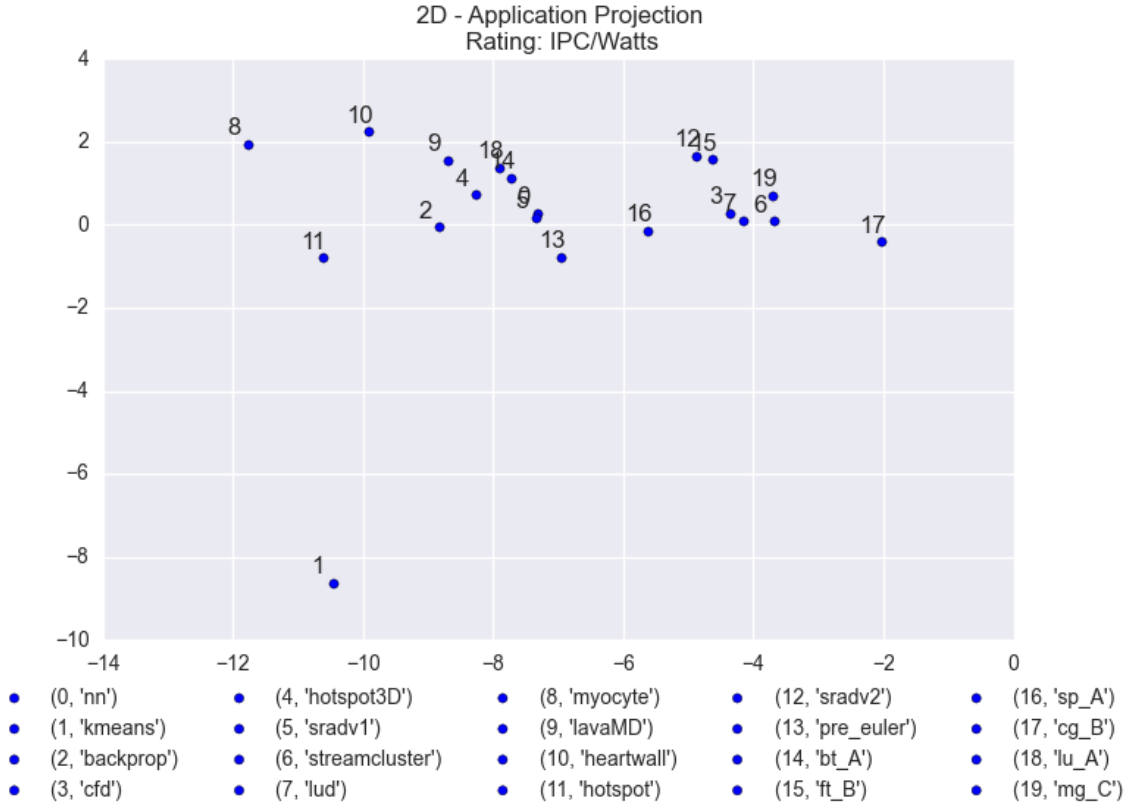


Figure 4.5: Two-dimensional graph of the applications, rating IPC/Watts.

Training set	Test Set
myocyte, kmeans	lu
hotspot, hotspot3D	nn
heartwall, backprop	sradv2
bt, sp, sradv1	mg
lavaMD, pre_euler	cfd
ft, lud, cg	streamcluster

Table 4.4: Table with the training and test sets from the 2D projection 4.5.

Analogously with the previous rating, we create also the learning base that the coefficient matrix depicts. Using the Pearson's similarity function for this rating, we get the following correlations. The

similarity threshold is set again to 0.80.

	nn	kmeans	BP	cfd	HS3D	sradv1	SC	lud	myocyte	lavaMD	HW	HS	sradv2	pre_euler	bt	ft	sp	cg	lu	mg
nn	1.00	0.65	**0.85**	0.64	**0.95**	**0.93**	0.68	0.38	**0.83**	**0.82**	**0.84**	0.76	0.54	0.73	**0.88**	0.59	**0.81**	0.61	0.79	0.52
kmeans	0.65	1.00	0.63	0.51	0.63	0.69	0.54	0.33	0.57	0.57	0.59	0.64	0.39	0.71	0.64	0.41	0.61	0.59	0.55	0.37
BP	**0.85**	0.63	1.00	0.60	**0.85**	**0.84**	0.67	0.37	0.77	0.75	0.78	0.71	0.52	0.70	0.79	0.59	0.72	0.55	0.74	0.50
cfd	0.64	0.51	0.60	1.00	0.63	0.67	0.57	0.15	0.59	0.62	0.63	0.58	0.49	0.29	0.61	0.55	0.60	0.46	0.61	0.50
HS3D	0.95	0.63	**0.85**	0.63	1.00	**0.90**	0.68	0.38	**0.83**	**0.82**	**0.86**	0.75	0.56	0.71	**0.87**	0.62	0.79	0.59	**0.81**	0.52
sradv1	**0.93**	0.69	**0.84**	0.67	**0.90**	1.00	0.68	0.42	**0.82**	**0.82**	**0.83**	0.77	0.59	0.70	**0.88**	0.64	**0.83**	0.62	0.79	0.62
SC	0.68	0.54	0.67	0.57	0.68	0.68	1.00	0.32	0.64	0.53	0.63	0.52	0.43	0.50	0.64	0.56	0.60	0.51	0.54	0.48
lud	0.38	0.33	0.37	0.15	0.38	0.42	0.32	1.00	0.39	0.38	0.39	0.31	0.29	0.52	0.43	0.31	0.37	0.29	0.36	0.41
myocyte	**0.83**	0.57	0.77	0.59	**0.83**	**0.82**	0.64	0.39	1.00	0.77	**0.84**	0.72	0.58	0.69	**0.83**	0.62	0.68	0.55	0.75	0.53
lavaMD	**0.82**	0.57	0.75	0.62	**0.82**	**0.82**	0.53	0.38	0.77	1.00	**0.87**	0.73	0.75	0.73	**0.89**	0.74	0.72	0.55	**0.83**	0.59
HW	**0.84**	0.59	0.78	0.63	**0.86**	**0.83**	0.63	0.39	**0.84**	**0.87**	1.00	0.76	0.71	0.70	**0.89**	0.76	0.75	0.51	**0.88**	0.62
HS	0.76	0.64	0.71	0.58	0.75	0.77	0.52	0.31	0.72	0.73	0.76	1.00	0.55	0.66	0.76	0.55	0.69	0.53	0.71	0.50
sradv2	0.54	0.39	0.52	0.49	0.56	0.59	0.43	0.29	0.58	0.75	0.71	0.55	1.00	0.57	0.70	0.76	0.49	0.36	0.65	0.52
pre_euler	0.73	0.71	0.70	0.29	0.71	0.70	0.50	0.52	0.69	0.73	0.70	0.66	0.57	1.00	0.75	0.59	0.68	0.47	0.71	0.52
bt	**0.88**	0.64	0.79	0.61	**0.87**	**0.88**	0.64	0.43	**0.83**	**0.89**	**0.89**	0.76	0.70	0.75	1.00	0.75	**0.82**	0.59	**0.81**	0.62
ft	0.59	0.41	0.59	0.55	0.62	0.64	0.56	0.31	0.62	0.74	0.76	0.55	0.76	0.59	0.75	1.00	0.51	0.33	0.66	0.56
sp	**0.81**	0.61	0.72	0.60	0.79	**0.83**	0.60	0.37	0.68	0.72	0.75	0.69	0.49	0.68	**0.82**	0.51	1.00	0.74	0.73	0.55
cg	0.61	0.59	0.55	0.46	0.59	0.62	0.51	0.29	0.55	0.55	0.51	0.53	0.36	0.47	0.59	0.33	0.74	1.00	0.50	0.44
lu	0.79	0.55	0.74	0.61	**0.81**	0.79	0.54	0.36	0.75	**0.83**	**0.88**	0.71	0.65	0.71	**0.81**	0.66	0.73	0.50	1.00	0.53
mg	0.52	0.37	0.50	0.50	0.52	0.62	0.48	0.41	0.53	0.59	0.62	0.50	0.52	0.52	0.62	0.56	0.55	0.44	0.53	1.00

Table 4.5: Coefficient matrix for all the applications when rated by 2

The training and the test set are composed of the following applications:

Training Set	Test Set
nn, kmeans	backprop
hotspot, hotspot3D	heartwall
sradv2, sradv1,	myocyte
cfd, pre_euler	lavaMD
streamcluster, lud	sp
ft, mg, bt, cg	lu

Hence, starting from the above groups we form our learning base and investigate how our recommendation system behaves.

4.3 Online Stage

In this stage, the tuning recommendation happens. Any new incoming application to run natively on the coprocessor, is profiled against a small proportion of the tuning configuration, in the same way as the profiling was implemented in the offline stage. Then it gets into the autotuner who uses the application's partial profiling along with his own established base to produce the application's personal tuning configuration. The mechanism behind the recommendation is the latent factor model we described in Section 4.1.1. In particular, a set of applications ratings for configurations, which includes the whole learning base plus the ratings from the partial profiling of the new application, is used to train our model 4.1. Using the stochastic gradient descent 4.1.2, we try to minimize the squared error function 4.2. So, a rating is described as in Figure 4.6 and therefore the prediction is the sum of the four parts.

The outcome is two vectors of application and configuration biases and two matrices, one for the applications and one for the configurations with values that represent the preferences over the latent factor space.

The size of the partial profiling will be examined along with the composition of the learning base, targeting both speed and accuracy over the system. In addition, towards improving our base and consequently recommendation's accuracy, we consider providing feedback to the system from the partial

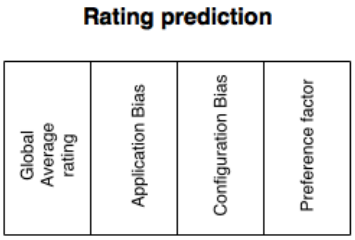


Figure 4.6: Break up of a rating.

profilings. So, gradually our autotuner refines his recommendations. That also means that the size of the tuning configurations' set for the online stage could also decrease while the autotuner is online, contributing to lesser time spent on new applications' profiling. However, as the base grows, the processing time of the latent factor model also increases. Thus, we need to investigate that also and find a trade-off between recommendation accuracy and time.

The autotuner and its building blocks are illustrated in the Figure 4.7 below.

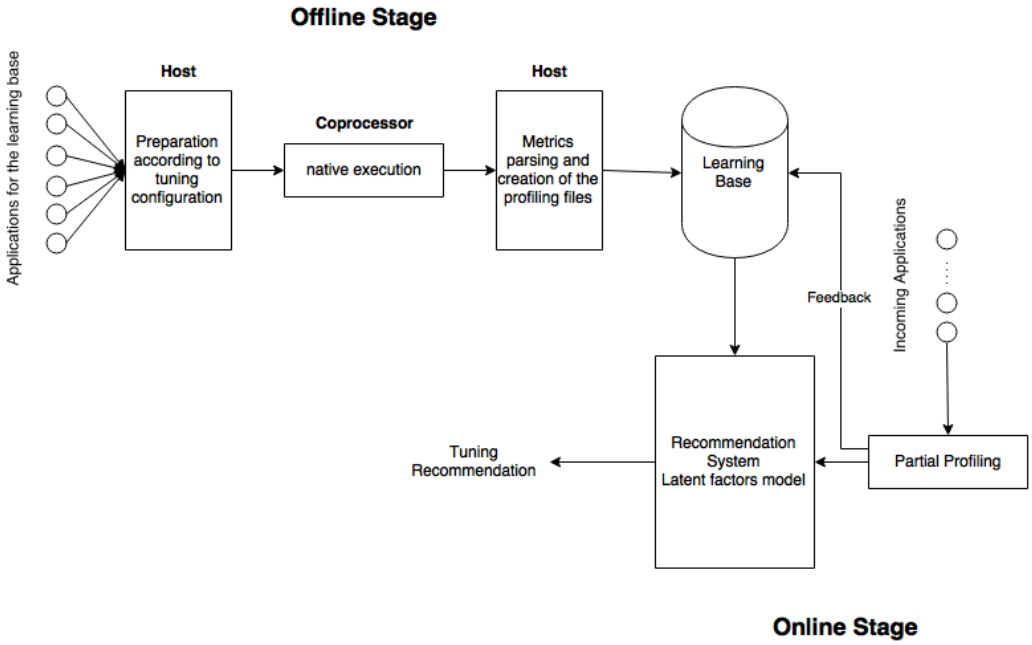


Figure 4.7: Autotuner's components.

Chapter 5

Experimental Results

In this chapter we present the accuracy of our autotuner in respect of the learning base composition, of the number of features, of the profile size for the incoming applications and of their feedback. In the last part we compare the tuning configurations when we use an energy aware and an no energy aware rating, so that we measure power savings.

To summarize our setup, our learning base (see 4.2.2) has been fully profiled against 2880 tuning configurations. Each application that belongs to the learning base ran natively on the Intel Xeon Phi coprocessor. Each new application was partially profiled against a varying but small proportion of the tuning configurations.

Based on the aforementioned setup's description we examine our autotuner.

5.1 Accuracy of Predictions

In order to measure our predictions, we employ the root mean square error (RMSE) metric and then we compare the best predicted configuration with the best actual one as measured during normal execution. The RMSE is defined as:

$$RMSE = \sqrt{\frac{\sum_{(u,i)\in\mathcal{K}}(r_{ui} - \hat{r}_{ui})^2}{|\mathcal{K}|}}$$

where $\mathcal{K}$ is the training set.

The ratings, as we mentioned before, are two:

1. $r_1^{ui} = \frac{MFlops/sec}{Watts}$ (normalized to the base rating)
2. $r_2^{ui} = \frac{IPC}{Watts}$

and we examine both.

5.1.1 Rating: MFlops/sec per Watt

The initial learning base includes the applications: *nn, kmeans, backprop, cfd, sradv1, streamcluster, lud, heartwall, hotspot, sradv2, pre_euler, bt, cg and mg*. A total of 14 applications. Ratings are predicted for *hotspot3D, lavaMD, myocyte, ft, sp and lu*. This composition of the learning base and the test applications is based on the Pearson's similarity function described in Section 4.2.2. In addition, as a training percent from the incoming applications we use 10% of the total tuning configurations, chosen randomly.

η_1	η_2	η_3	η_4	λ_1	λ_2	λ_3	λ_4
0.002	0.002	0.002	0.003	0.01	0.01	0.02	0.03

Table 5.1: Learing and regulating rates used.

For the latent factor model, we used the following learning and regulating rates, Table 5.1, which where reckoned by grid search.

Firstly, we examine the number of features used to describe the latent factor space with and without feedback. Figure 5.1 shows the RMSE.

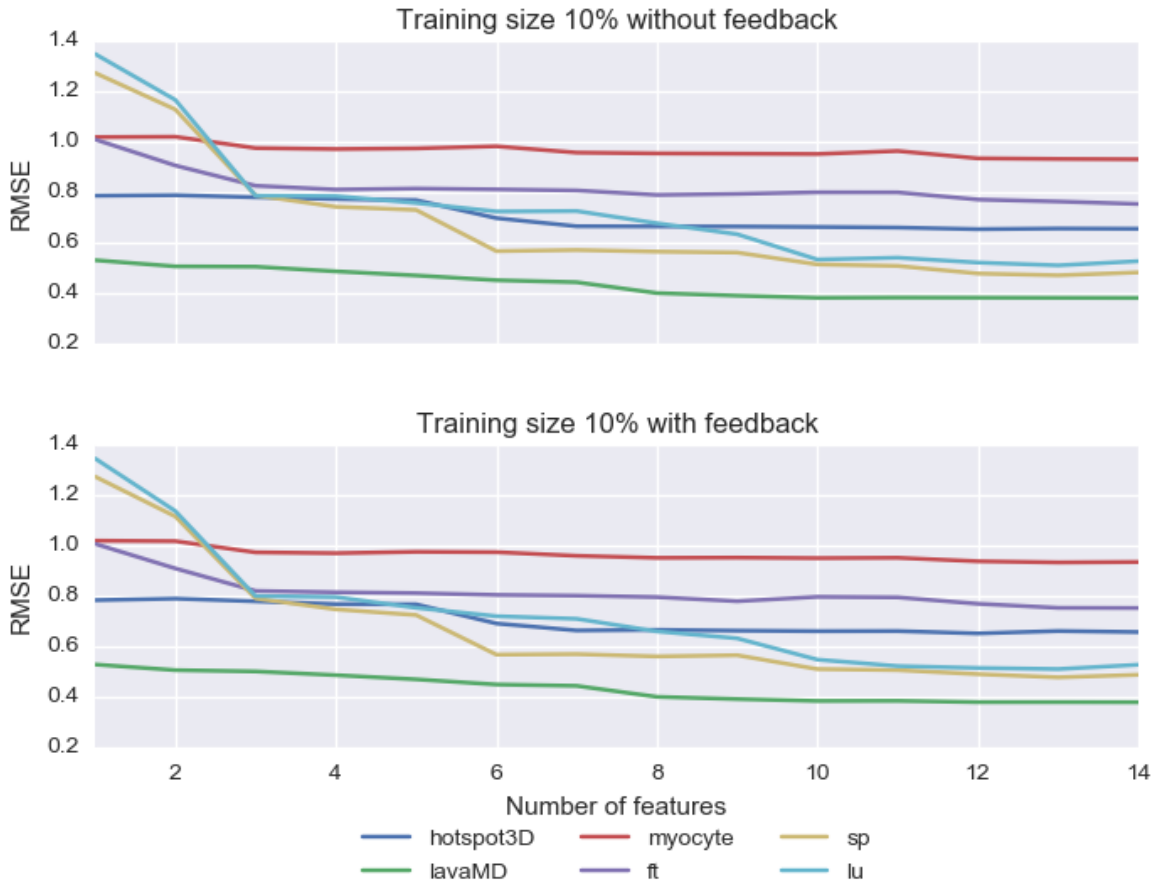


Figure 5.1: RMSE for a number of features with or without feedback.

We notice that after 12 features every test application has its RMSE stabilized. Some applications, such us *myocyte* and *hotspot3D* reach their best RMSE with about 8 features. However, that is caused by the amount of similar applications in the learning base. For example, *myocyte* is close to only *streamcluster*. Hence, we can choose 12 features as our dimension of latent factor space, the minimum number as we are time aware and we care about the respond time of the autotuner. Furthermore, from the RMSE with feedback we do not see any obvious improvement and that may be caused to the small amount of feedback ratings used from the incoming applications that do not add significant information to our learning base. However, from the similarity definition we know that some test applications have resemblances, *lavaMD and ft, sp and lu*, so we choose the feedback version.

Now, we may examine the average RMSE for different training sizes of the incoming applications and number of features. Figure 5.2 shows the RMSE as a function of two variables, number of features and size of training set, with and without feedback.

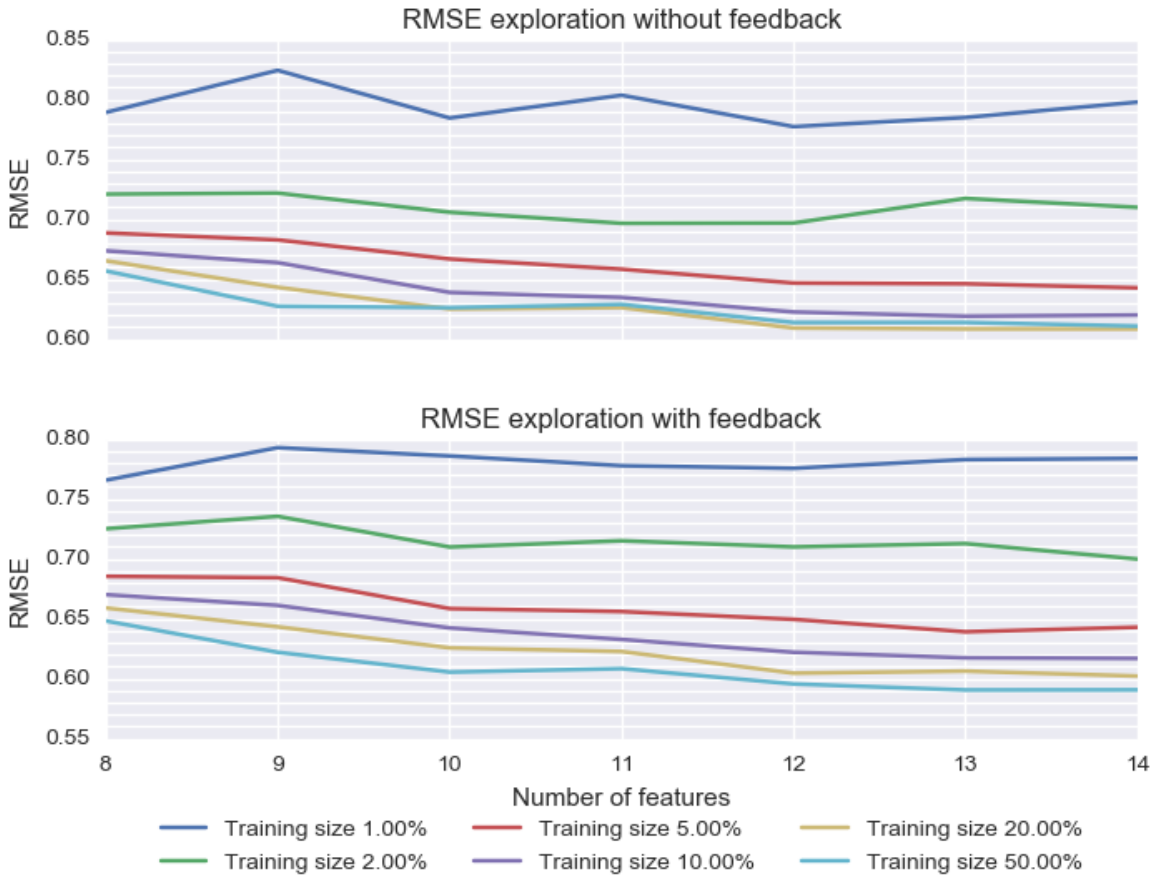


Figure 5.2: Average RMSE for a number of features and training size with or without feedback.

It is apparent that the average RMSE for the known 1% and 2% training ratings of the new applications is high comparable with the rest training sizes, however not at prohibited levels. The 5% of the configurations, which accounts for only 144 random tuning configurations, provides an average RMSE very close to the one achieved when we know half the ratings (50% training size). The 10% does not lead to any major improvement, but it is also an acceptable time consuming option which should be examined also. So, we will examine the predictions of both the 5% and 10% training sizes. The dimension of the latent factor space to which the model exhibits the best RMSE is 12 features and slight better it is for 13 features. The feedback version also seems to add some accuracy, which is more notable at the higher training sizes (>10%).

With reference to a latent feature space of dimension 12, we compare the predicted rating for different training sizes of the incoming applications, with or without feedback as well as the correlation between the predicted configurations and the actual best one in the latent factor space. For that relation we use the Pearson function (4.2.2). The smaller the training size the faster the tuning prediction, thus we start from very low sizes: *0.1%, 0.2%, 0.5%, 1%, 2%, 5%, 10%*. From the previous Figure (5.2), as the RMSE for small sizes was increasing, we expect that the small sizes' predicted configurations will present major divergences from the actual best one, which translates to small correlations. Figure 5.3 shows the average predicted ratings normalized to the best one.

We notice that on average the lowest rating we achieve we the predicted configurations is 90.49% without feedback and 92.65% with feedback relative to the best rating. In addition, the base configuration reaches the 70.0%, so we have a major improvement. However, we need to see also the configurations that correspond to these values and their relation with the actual best. Table 5.2 shows the correlation coefficients of the tuning configurations by application and by average.

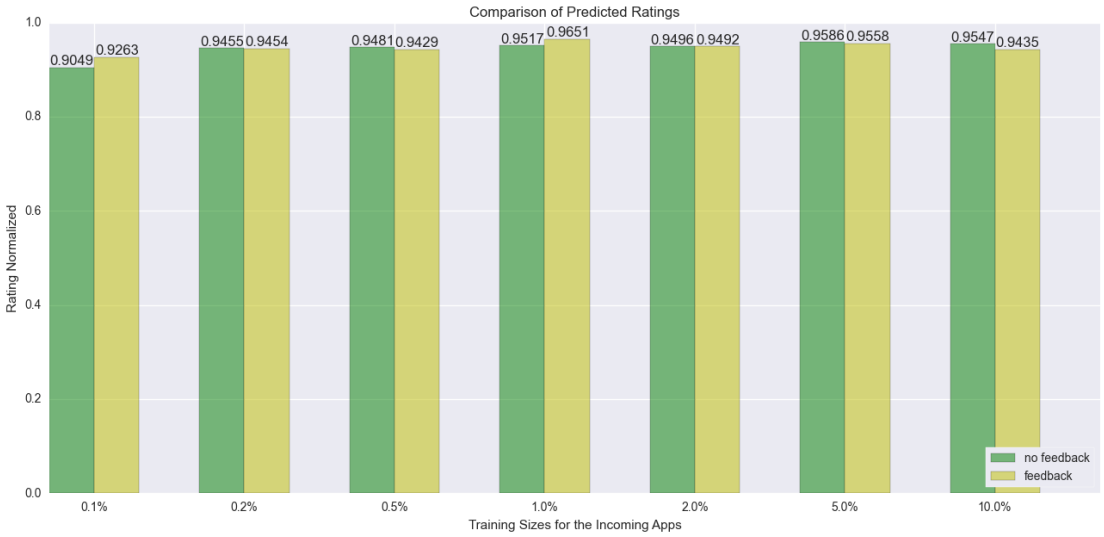


Figure 5.3: Normalized ratings of predicted configurations with respect to training size.

	Training Sizes													
	0.1%		0.2%		0.5%		1%		2%		5%		10%	
	nfb	fb	nfb	fb	nfb	fb	nfb	fb	nfb	fb	nfb	fb	nfb	fb
hotspot3D	0.56	0.56	0.56	0.56	0.56	0.56	0.70	0.71	**0.73**	0.70	**0.73**	**0.73**	**0.73**	**0.73**
lavaMD	**0.77**	**0.77**	**0.77**	**0.77**	**0.77**	**0.77**	**0.77**	**0.77**	0.69	0.69	0.66	0.66	0.66	0.66
myocyte	**0.81**	**0.81**	0.78	0.78	0.78	0.61	0.60	0.60	0.78	0.78	0.78	0.60	0.78	0.60
ft	0.77	0.78	0.78	0.87	0.87	0.87	0.87	0.87	0.92	**0.93**	0.92	0.92	0.92	0.92
sp	**0.95**	0.75	0.89	0.88	**0.95**	0.88	0.85	**0.95**	**0.95**	**0.95**	0.91	**0.95**	**0.95**	0.88
lu	0.68	**1.00**	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.85	0.90	0.90	0.90
Average	0.76	0.78	0.78	0.80	0.81	0.77	0.78	0.80	**0.83**	**0.83**	0.81	0.80	0.82	0.78

Table 5.2: Correlations between the predicted configurations and the best one by application and by average for different training sizes.

With boldface are annotated the maximum correlations for each application and average. Overall, the predicted tuning configurations have from the least 0.1% a correlation greater than 0.70 with the best one. On average, the closest configurations to the actual one occur for the 2%. Yet, since for every training size we have a rating greater than 90% of the best and a tuning configuration with correlation to the best one greater than 0.70 we may choose the training size which satisfies our needs both in accuracy and time.

Further to the evaluation of the predictions, we present the performance achieved of the predicted values specifically for the 2%, 5% and 10% training sizes, compared to the best ones and to the base configuration. These are the training sizes with the most relative predicted configurations to the best one (>0.80). Figure 5.4 shows the four performances for test set, as it was created in the Section 4.2.2, when the predictions are run with feedback.

We see that the predicted configurations are very close to the actual ones. In addition, the improvement from the base configuration is apparent. The predicted configurations from every training size are almost equal, so the best choice would be the one that satisfies also the time constraints, i.e. provides the fastest result. Thus the 2% is the optimum size of known ratings for the current model. Table 5.3 shows the exact ratings.

In order to finish our evaluation we need also to look with more detail and compare the configuration

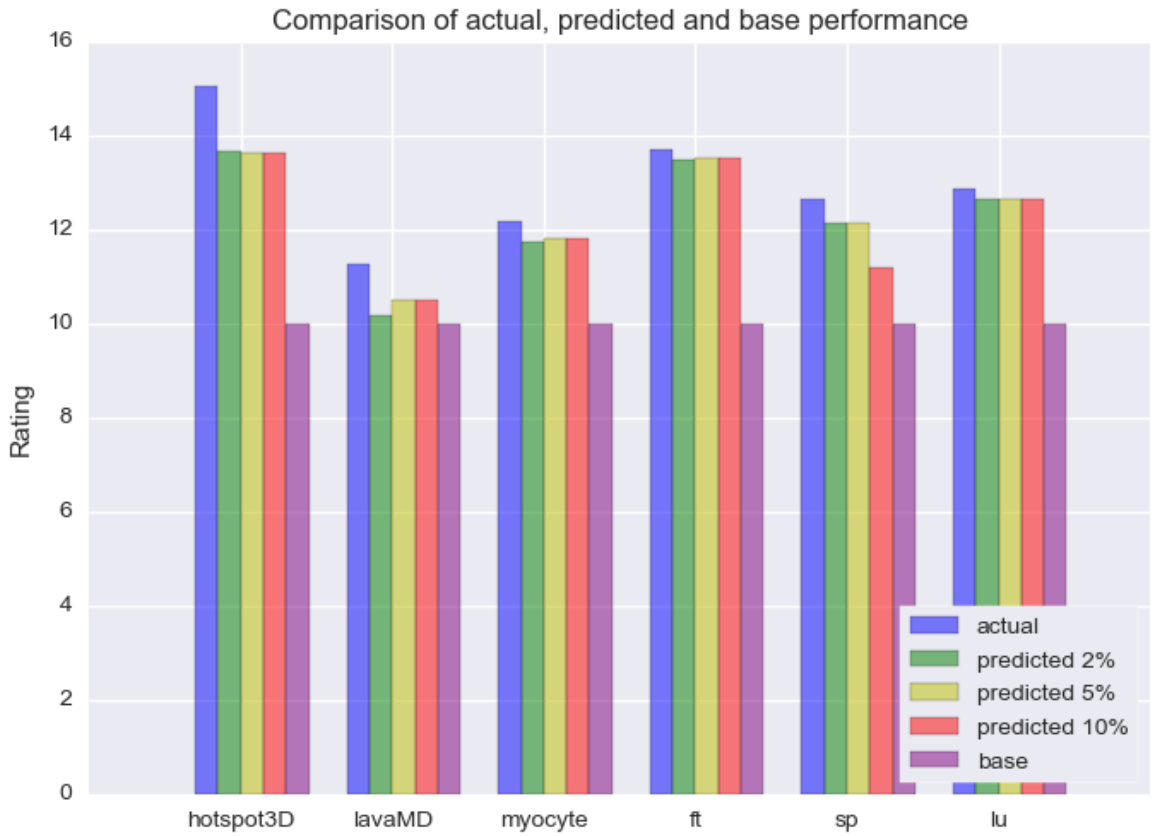


Figure 5.4: Performance comparison for 12 features, 2%, 5% and 10% known ratings, with feedback.

	Ratings				
Application	Actual	Predicted 2%	Predicted 5%	Predicted 10%	Base
hotspot3D	15.066466	13.667847	13.635916	13.635916	10.00
lavaMD	11.271719	10.160522	10.502893	10.502893	10.00
myocyte	12.166096	11.732373	11.812218	11.812218	10.00
ft	13.712121	13.478020	13.536496	13.536496	10.00
sp	12.665992	12.132006	12.132006	11.203003	10.00
lu	12.879007	12.643508	12.643508	12.643508	10.00

Table 5.3: Predicted ratings for 12 features, 2%, 5% and 10% known ratings, with feedback along with the actual and the base ratings.

vectors of the best and the predicted performances. Then we can find the features that benefit each application. Table 5.4 presents the configuration vectors for the best actual and the predicted 2%, 5% and 10% ratings.

The configuration vectors generally agree over the number of cores and threads and over thread affinity. They also have common optimization level and unroll policy. Some alterations are noted over the huge pages, e.g. *lavaMD*, however that may be caused to the fact that huge pages do not affect a lot the execution of the application. The memory requests may be few and can be served both from a 4KB and a 2MB page without comparable latency. Furthermore, streaming stores do not always agree on the cache eviction, but that may be also irrelevant because the cache eviction depends on the problem size. Lastly, prefetch presents also some differences between the best actual and the predicted configurations vector. As a compiler's flag prefetch represents a finer tuning, hence it is difficult to expose

	Tuning Configurations								
	opt	prefetch	sstores	cache evict	unroll	huge pages	affinity	cores	threads per core
hotspot3D									
Actual	2	0	always	2	✓	✓	scatter	19	2
Predicted 2%	3	0	always	0	✓	✓	balanced	19	3
Predicted 5%	3	0	always	3	✓	-	balanced	19	3
Predicted 10%	3	0	always	3	✓	-	balanced	19	3
lavaMD									
Actual	3	4	always	2	✓	-	balanced	38	2
Predicted 2%	3	4	always	1	✓	✓	balanced	57	4
Predicted 5%	2	2	always	0	✓	-	balanced	57	3
Predicted 10%	2	2	always	0	✓	-	balanced	57	3
myocyte									
Actual	3	2	never	-	-	✓	balanced	19	4
Predicted 2%	2	3	never	-	✓	-	balanced	19	3
Predicted 5%	3	0	always	3	✓	-	balanced	19	3
Predicted 10%	3	0	always	3	✓	-	balanced	19	3
ft									
Actual	2	2	always	0	-	-	scatter	57	2
Predicted 2%	3	2	always	0	-	✓	scatter	57	2
Predicted 5%	2	2	always	0	✓	-	scatter	57	2
Predicted 10%	2	2	always	0	✓	-	scatter	57	2
sp									
Actual	2	2	always	1	✓	-	balanced	38	2
Predicted 2%	2	2	always	0	✓	✓	balanced	38	2
Predicted 5%	2	2	always	0	✓	✓	balanced	38	2
Predicted 10%	3	4	always	3	✓	-	balanced	38	2
lu									
Actual	2	2	never	-	✓	✓	balanced	38	2
Predicted 2%	2	2	always	0	✓	✓	balanced	38	2
Predicted 5%	2	2	always	0	✓	✓	balanced	38	2
Predicted 10%	2	2	always	0	✓	✓	balanced	38	2

Table 5.4: Best configurations both actual and predicted.

its effects. In addition, it is also a parameter affected by the problem size. By looking at the results as a whole, the predictions are more than satisfying with fine accuracy.

To continue our evaluation, we will vary our learning base in order to observe how it behaves. The test set will remain the same. From the coefficient matrix we can get the applications that are similar to the test set (correlation >0.80). So, each testing application is connected with the following ones:

- **hotspot3D**, backprop, streamcluster, myocyte, hotspot
- **lavaMD**, cfd, sradv2, ft
- **myocyte**, hotspot3D, streamcluster
- **ft**, lavaMD, sradv2
- **sp**, bt, lu
- **lu**, bt, sp

If we leave only those relatives applications in our learning base we expect not to get a big degradation in our predictions. We refer to the new learning base as ”learning base 2”. Figure 5.5 shows the RMSE for the test applications.

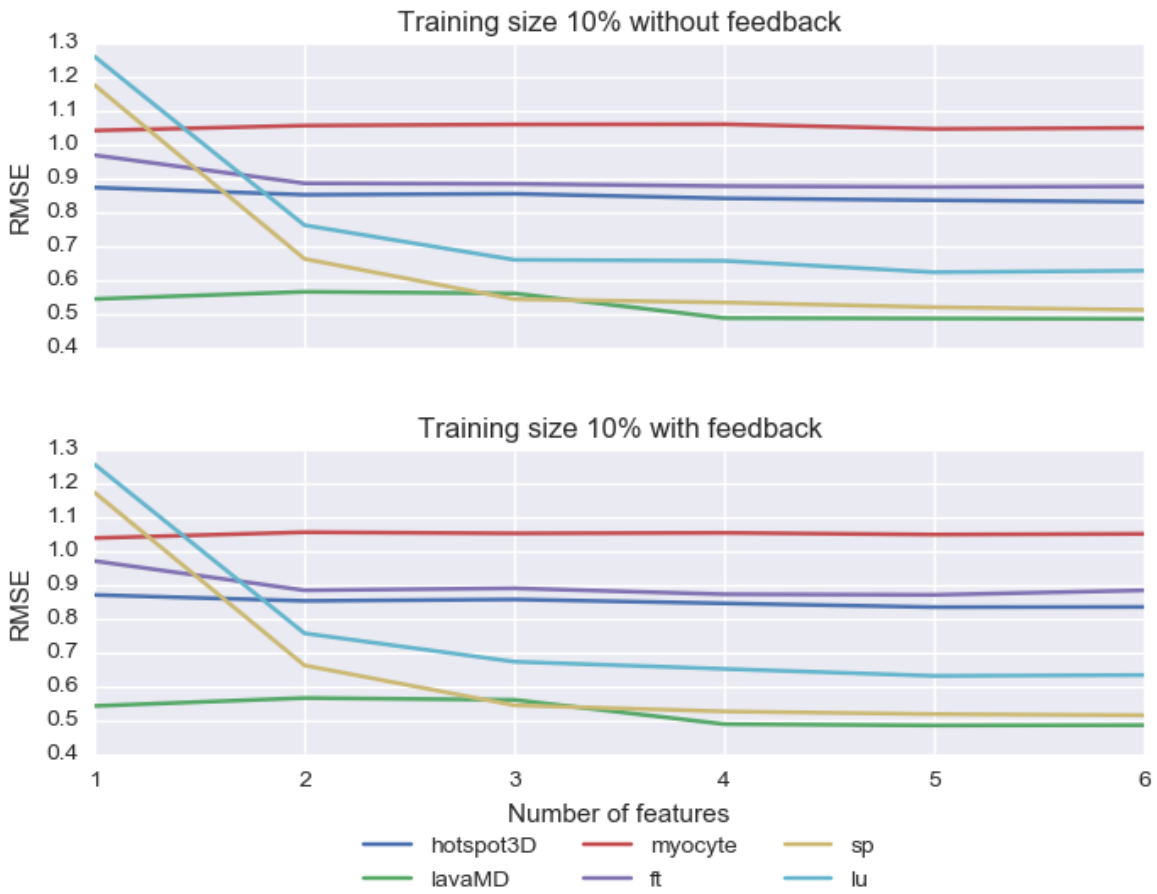


Figure 5.5: RMSE for different number of features, with or without feedback based upon the learning base 2.

Eventually, the learning base 2 seems to be insufficient and the threshold of the 0.80 does not fully expose the similarities between the applications. For instance, *myocyte*’s RMSE is constant at around

1.0 which means that *streamcluster* does not fully characterize its attributes. Furthermore, even though hotspot3D has 0.8 correlation with *myocyte*, in the feedback version it does not provide noteworthy improvement. The same applies also to the applications *ft, hotspot3D, lu*. If we lower the threshold to 0.70 then the learning base will almost be identical with the first tested but omitting *lud, cg, mg*. Hence, we will have the same outcome.

Overall, good results were achieved with a latent factor dimension of 12 and feedback. The training size of the incoming applications depends on the accuracy of the predictions we aim, hence for training sizes of 0.1%-1% that require 3-30 minutes we get an accuracy around 0.70 and for training sizes of 2%-5% that require 57-144 minutes we get an accuracy greater than 0.80. Still, always our performance exceeds the 90% of the best configuration.

5.1.2 Rating: IPC per Watt

The initial learning base includes the applications: *myocyte, kmeans, hotspot, heartwall, backprop, lavaMD, hotspot3D, lud, sradv1, pre_euler, sp, ft, bt and cg*. A total of 14 applications. Ratings are predicted for *sradv2, cfd, streamcluster, nn, lu and mg*. The composition of the learning base was derived from the neighboring of the 2 dimensional projection described in Section 4.2.2. In addition, as a training percent from the incoming applications we use 10% of the total tuning configurations, chosen randomly. For the latent factor model, we used the following learning and regulating rates, Table 5.5, which where reckoned by grid search.

η_1	η_2	η_3	η_4	λ_1	λ_2	λ_3	λ_4
0.002	0.001	0.005	0.005	0.01	0.01	0.01	0.01

Table 5.5: Learing and regulating rates used.

Firstly, we examine the number of features required to describe the latent factor space with and without feedback. Figure 5.6 shows the RMSE.

We notice that after 9 features the RMSE is stabilized at 0.2375 without feedback and 0.2387 with feedback, on average. Thus, we can choose 9 features as our dimension of latent factor space, which provides the fastest predictions as higher dimensions impute more latency. In addition, feedback does not seem to benefit a lot our predictions. The best RMSE with feedback happens for features=11, RMSE=0.2373 on average. So, we may argue that our learning base is consistent and does not need any more information or that the knowledge from the partial profilings do not provide any substantial contribution to the recommender model. Hence, we may also choose the non feedback version in order to save time along with space. It should be noted that in order to be considerable the time and the space latency from the feedback version, many applications should be tested as one application may add a couple of seconds and space in the scale of bytes, depending on the training size.

Now, we may examine the average RMSE for different training sizes of the incoming applications and number of features. Figure 5.7 shows the RMSE as a function of two variables, number of features and size of training set, with and without feedback.

We notice that the biggest alterations happen between training sizes 1%, 2% and 5%, while the RMSEs for the higher training sizes are very close with the latter one. For once more, we observe that the RMSE converges to 0.24 after 9 features for training sizes >5%. Overall, a good choice would be 5% or 10% for the training size, as they produce a satisfactory RMSE. However, that should also be examined by the prediction accuracy. The 20% does not provide considerable improvement when we take into account the time needed for the partial profiling, around 576 tuning configurations. Furthermore, by looking at the 50% we see no dramatic improvement, so probably the error remaining is due to the

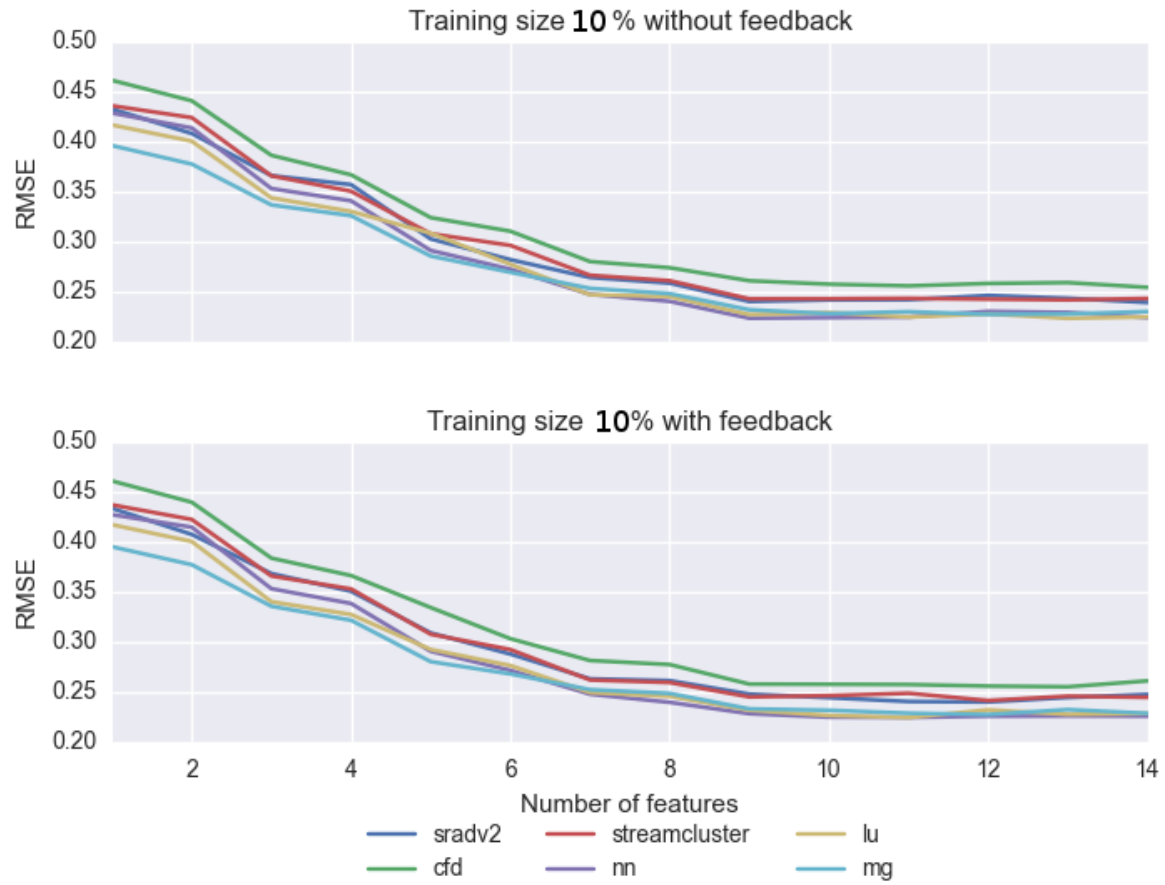


Figure 5.6: RMSE for a number of features with or without feedback.

incompleteness of the model used. Yet, it is negligible. Lastly, feedback seems that it does not affect our current model, hence we choose the most time saving approach, i.e. without feedback.

Besides the RMSE for the training sizes, the predictions accuracy compared to the best tuning configuration should be evaluated. For that reason, the autotuner is run with training sizes *0.1%, 0.2%, 0.5%, 1%, 2%, 5%, 10%*, a latent space dimension of 9 with and without feedback. The parameter ”prediction accuracy” consists of both performance achieved relative to the maximum and correlation between the predicted tuning configuration and best one. The relation is defined by the Pearson function, as done before. Figure 5.8 shows the average predicted ratings normalized to the best one.

As we expected, the bigger the training size the better the performance achieved. Still, for training sizes >1%, the rating is steady at around 95%. For every training size, the ratings exceed the 85% of the best configuration, both in the feedback and non feedback version. Slightly worse is the feedback version so we could choose the no feedback version. That is not a general rule, as the specific test set behaves in that way, for a more wider set of incoming applications in respect of features, feedback should be considered because it has been proved to improve the tuning suggestions. Lastly, the base configuration accounts to only 67.8% of the best one, so our predictions increase by a 25% which is a considerable percentage.

To deepen our evaluation, we will examine the predicted tuning configurations of each application that correspond to the previous training sizes with respect to their similarity with the best tuning configuration. The similarity is calculated in the full latent feature space. Table 5.6 presents the similarity coefficients.

For all the training sizes the correlation is greater than or equal to 0.75 and in particular for the training

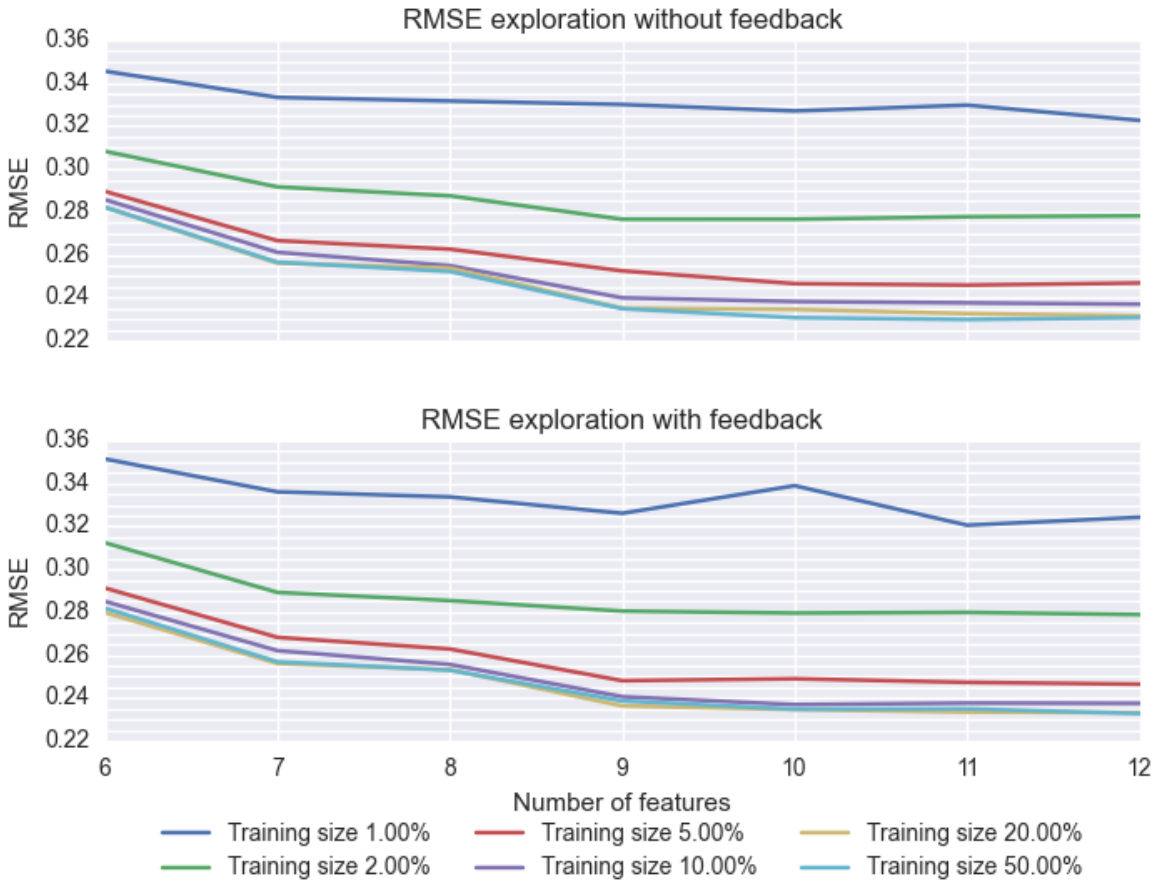


Figure 5.7: Average RMSE for a number of features and training size with or without feedback.

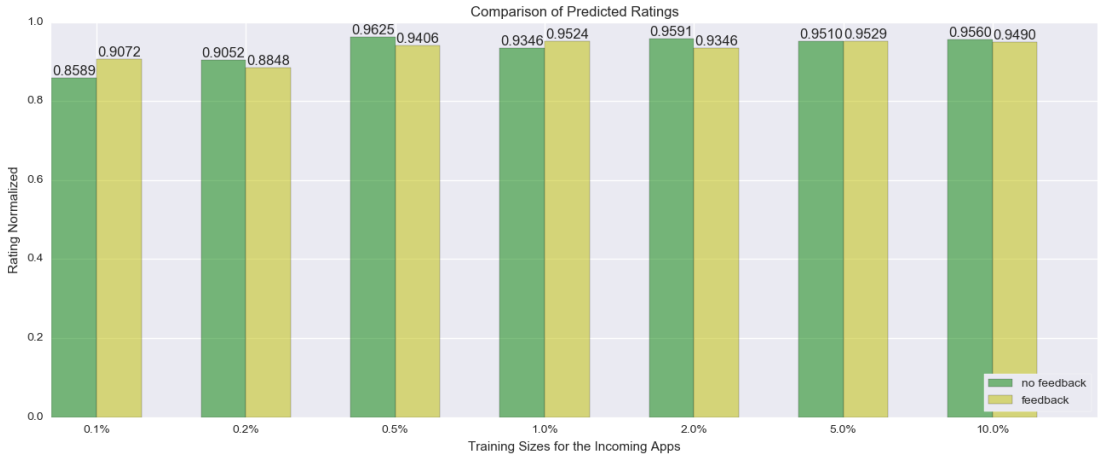


Figure 5.8: Normalized ratings of predicted configurations with respect to training sizes.

sizes *0.5%,1%,2%,5%* and *10%* the similarities are over 0.80, which is a good result. With boldface are annotated the coefficients with the largest values over the training sizes for each application. On average, the best tuning configurations are produced for training sizes greater than 0.5%.

Next we will examine more meticulously the accuracy and the efficiency of our predictions for 9 features, without feedback, for *0.5%, 1%, 2%* and *5%* as the training size, because these values provide good performance and tuning configurations.

	Training Sizes													
	0.1%		0.2%		0.5%		1%		2%		5%		10%	
	nfb	fb	nfb	fb	nfb	fb	nfb	fb	nfb	fb	nfb	fb	nfb	fb
sradv2	0.75	0.75	0.73	0.73	0.78	0.73	0.76	0.73	0.76	0.81	**0.83**	**0.83**	**0.83**	**0.83**
cfd	0.75	0.82	0.82	0.75	0.93	**1.00**	0.95	**0.99**	**0.99**	0.96	0.96	0.96	0.96	0.95
streamcluster	0.78	0.78	0.78	0.74	0.78	**0.80**	**0.80**	0.78	0.79	**0.80**	**0.80**	0.79	**0.80**	**0.80**
nn	0.77	0.77	0.76	0.77	**0.84**	0.76	**0.84**	**0.84**	0.83	0.83	0.83	0.79	0.79	0.79
lu	0.59	0.62	0.59	0.62	0.59	0.61	0.79	0.61	0.63	0.72	**0.81**	0.72	0.72	0.71
mg	0.89	0.96	0.96	0.93	0.96	0.96	**0.98**	0.96	**0.98**	**0.98**	**0.98**	0.94	**0.98**	**0.98**
Average	0.75	0.78	0.77	0.76	0.81	0.81	**0.85**	0.82	0.83	**0.85**	**0.87**	0.84	**0.85**	0.84

Table 5.6: Correlations between the predicted configurations and the best one by application and by average for different training sizes.

Figure 5.9 shows the best actual, the best predicted and the base rating for the six files that were tested on the autotuner under the previous settings.

Figure 5.9: Performance comparison for 9 features, 0.5%, 1%, 2% and 5 % known ratings, without feedback.

We observe that the predicted ratings are close to the actual ones and that profiling over smaller percentages provide satisfactory tuning configurations, instead of the higher more time consuming percentages. The 0.5% seems to be a good choice for the partial profiling. It accounts for only 14 tuning configurations and requires approximately only 14 minutes. Lastly, it is apparent the improvement from the base rating, which can reach up to 112% (sradv2) and at least 2.6% (cfd). In particular, the Table 5.7 shows the exact ratings, where the percentage changes can be calculated.

	Ratings					
			Predicted			
Application	Actual	Base	0.5%	1%	2%	5%
sradv2	4.497278	2.069584	4.416150	4.401062	4.401062	4.469881
cfd	4.579562	4.449481	4.570718	4.100565	4.527340	4.242349
streamcluster	4.012238	2.461628	3.923021	3.638051	3.933886	3.954854
nn	6.373608	4.996673	6.271984	6.271984	6.146772	6.146772
lu	6.785066	4.658377	6.123672	6.181696	6.182525	6.135312
mg	3.896091	2.143704	3.626442	3.628947	3.628947	3.628947

Table 5.7: Predicted ratings for 9 features, 5% and 10% known ratings, without feedback along with the actual and the base ratings.

It would be interesting to see also the tuning configurations to which these ratings correspond, and try to extract the attributes that benefit each application. Table 5.8 presents the best configurations, both actual and predicted, meticulously. We notice that our predictions agree with the actual ratings in many fields, affinity, threading, optimization level. Thereafter we get some small variations. For example, for *streamcluster* the best configuration use loop unroll whereas the predicted configurations do not. Similarly, for *mg* the best configuration use the L1 as cache eviction level for the streaming stores while the predicted configurations do not use any cache. Despite those small notes, the result satisfies our aim.

To continue our evaluation of the autotuner, we will change our learning base in order to observe how the model behaves and the consequences of such an action to the predictions. The test set will remain the same. Towards that goal the two dimensional projection 4.5 from Section 4.2.2 would assist us. We see that the test applications belong to two neighborhoods. Applications *nn, lu* belong to the group *lavaMD, hotspot3D, backprop, bt, sradv1, pre_euler* and applications *sradv2, cfd, streamcluster and mg* belong to the group *ft, lud*. So a first notion would be to get rid of the applications outside those groups, referring to *cg, myocyte, hotspot, kmeans, heartwall, sp*, resulting in a learning base consisting of *lavaMD, hotspot3D, backprop, bt, sradv1, pre_euler, ft, lud*. This learning base is referred as ”learning base 2”.

Figure 5.10 shows the RMSE that is produced under the new learning base. We notice that after 6 features the RMSE converges and the values are satisfying while for particular applications, such as *nn, streamcluster*, it is even lower. Again there is not any obvious benefit from the feedback. This result was expected as we kept the full neighborhoods as before, yet some alterations can be noted, such as *lu, cfd*, which could possibly be ascribed to the fact that we omitted some relations between the applications by looking only to the 2D projection. The number of features reduced to 6 as we removed some ”irrelevant” applications that would provide their own features.

If we further reduce the learning base to the set *bt, sradv1, ft, lud* which are the closest neighbors to the testing applications in the 2D projection, we get a slight worse RMSE comparing to the previous. This learning base is referred as ”learning base 3”. Figure 5.11 shows that. We note also that after 4 features the RMSE is stable. That happens because the learning base is far too small and the applications too similar, in order for the model to extract a big enough latent factor space. Therefore, even though we are able to predict ratings for the testing applications with a mediocre RMSE, we are missing important information and the learning base could be characterized deficient. Again we may have omitted other related applications to the ones in the test set as we are based only on the 2D factor space.

Concerning the learning base as an fundamental factor of our autotuner, we need to choose a set of applications with a lot of diversity, which the model will be able to use and reckon a complete latent factor space where the majority of the application will be able to map efficiently. For that reason,

	Tuning Configurations								
	opt	prefetch	sstores	cache evict	unroll	huge pages	affinity	cores	threads per core
sradv2									
Actual	3	4	always	3	✓	✓	balanced	19	4
Predicted 0.5%	3	2	always	1	-	✓	scatter	19	4
Predicted 1%	3	2	always	1	-	✓	scatter	19	4
Predicted 2%	3	2	always	1	-	-	scatter	19	4
Predicted 5%	2	2	always	3	-	-	balanced	19	4
cfd									
Actual	2	2	always	3	✓	✓	scatter	19	4
Predicted 0.5%	2	3	always	2	✓	✓	balanced	19	4
Predicted 1%	2	2	always	3	✓	-	scatter	19	4
Predicted 2%	2	4	always	3	✓	✓	scatter	19	4
Predicted 5%	2	4	always	3	✓	-	scatter	19	4
streamcluster									
Actual	2	4	always	0	✓	-	balanced	19	4
Predicted 0.5%	2	2	always	0	-	✓	balanced	19	4
Predicted 1%	2	4	never	-	-	✓	balanced	19	4
Predicted 2%	2	2	always	1	-	✓	balanced	19	4
Predicted 5%	2	4	always	1	-	✓	balanced	19	4
nn									
Actual	3	4	always	2	✓	-	balanced	19	4
Predicted 0.5%	2	3	always	3	-	✓	balanced	19	4
Predicted 1%	2	3	always	3	-	✓	balanced	19	4
Predicted 2%	2	3	always	2	-	✓	balanced	19	4
Predicted 5%	2	3	always	2	-	✓	balanced	19	4
lu									
Actual	2	3	always	0	✓	-	balanced	19	3
Predicted 0.5%	2	2	always	0	-	✓	scatter	19	4
Predicted 1%	2	2	never	-	✓	-	scatter	19	4
Predicted 2%	2	2	always	2	-	✓	scatter	19	4
Predicted 5%	2	3	always	2	✓	-	scatter	19	4
mg									
Actual	2	3	always	1	-	-	scatter	19	4
Predicted 0.5%	2	2	always	0	-	✓	scatter	19	4
Predicted 1%	2	2	always	0	-	-	scatter	19	4
Predicted 2%	2	2	always	0	-	-	scatter	19	4
Predicted 5%	2	2	always	0	-	-	scatter	19	4

Table 5.8: Best configurations both actual and predicted.

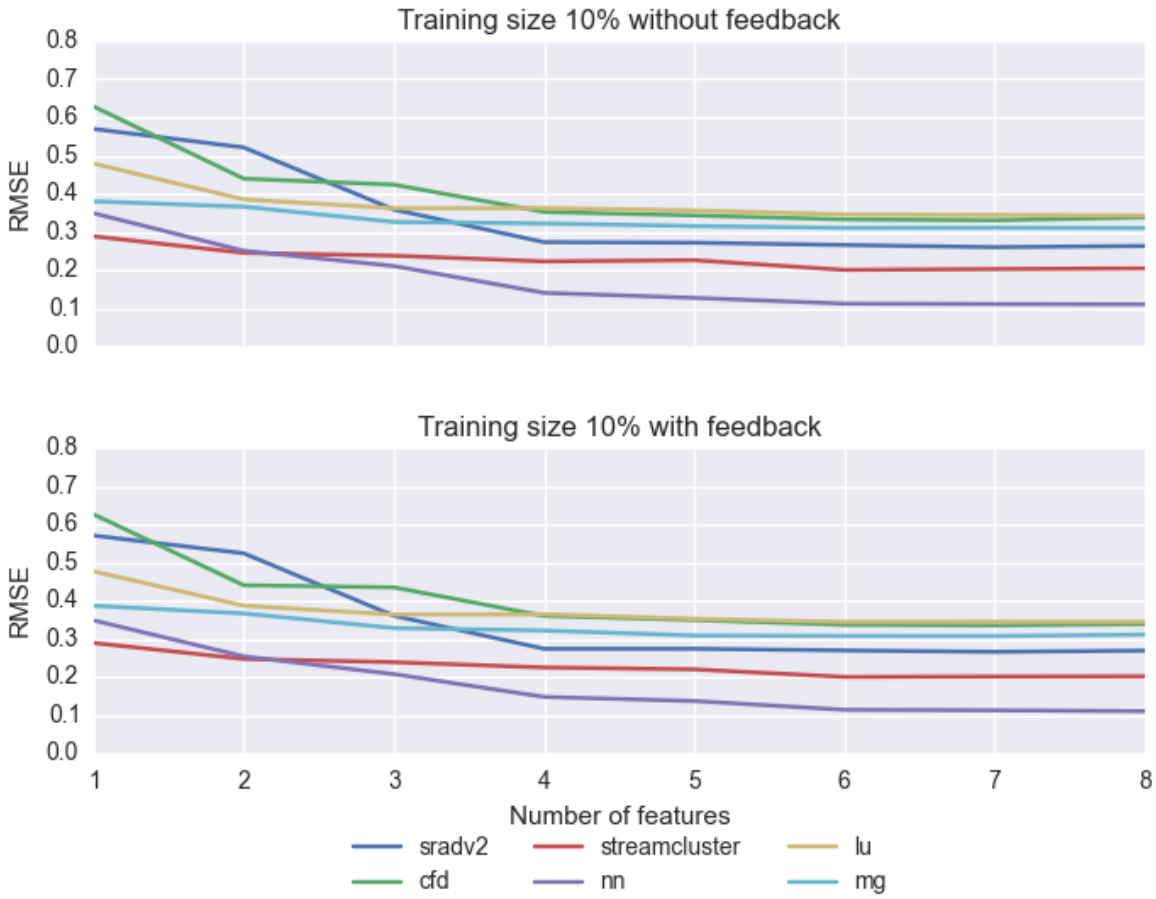


Figure 5.10: RMSE for different number of features, with or without feedback based upon the learning base 2.

between the three we created the most formal is the initial one. It proves to have more variance than the rests.

For the particular rating formula and chosen learning base, the autotuner works best with a latent factor of dimension 9. The training size is satisfying at 0.5% of known configurations. The feedback is not essential however as the autotuner suggests more and more configurations the extra information could be handy.

Now we will examine the learning base which was composed by the Pearson's similarity function. It consists of the applications: *nn, kmeans, hotspot3D, cfd, streamcluster, lud, hotspot, sradv1, sradv2, pre_euler, cg, bt, mg* and *ft*. A total of 14 applications. Ratings are predicted for the applications *backprop, myocyte, lavaMD, heartwall, sp, lu*. We use 10% of the ratings as known values for the incoming applications. Figure 5.12 shows the RMSE for each application with respect to number of features and with and without feedback.

We observe that early on, from 6 features, every application's RMSE is steady. Every application but *myocyte* has a RMSE below 0.5. That may not necessarily mean a bad outcome for *myocyte*. We will evaluate it in the comparison between the actual and predicted best configurations. The fact that we need only 6 features to characterize the test files can be justified by their neighborhood. Most of the applications in the test set have strong correlations between them and they are similar to only 4 application in the learning base, *nn, hotspot3D, sradv1* and *bt*. Hence, it is a very focused test set and the rest applications should affect predictions by a small degree.

By varying the training size in the Figure 5.13, the predicted configurations can be compared with the

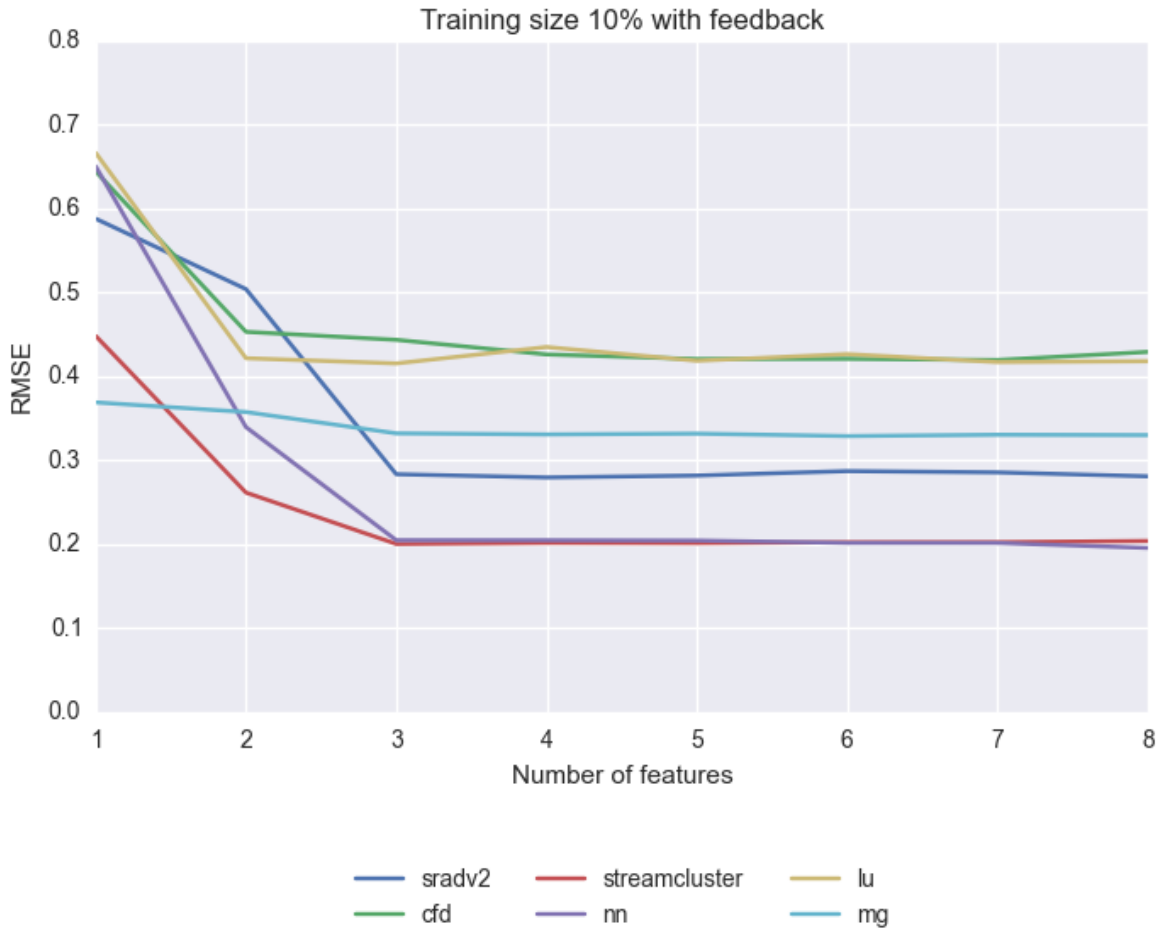


Figure 5.11: RMSE for different number of features, with or without feedback based upon the learning base 3.

best actual one.

From the very small training size of 0.1% the rating achieved is slightly over 93.0% and for every higher size it gets better and better reaching 96.0%. That is a very good performance and on the contrary with the previous test set that was based on the 2D projection, these values show the maximum performance that can be obtained, based on a robust and correlated test set. So, even if we use the 0.1% of the tuning configurations for the partial profiling we get very promptly a decent suggestion. Note that the 0.1% requires approximately 3 minutes of profiling.

So, if we specifically examine each application's predicted configurations for the training sizes 0.1%, 0.2% and 5% with feedback we get Figure 5.14. The ratings of predictions from the two smaller training sizes begin from 90.0% and escalate to 99.8% of the best rating. Generally, the predictions are always over 90% of performance. The 5% on the contrary performs a bit better as its lowest rating is at 94% and its highest 99.0%. It has a smaller range yet not important when we think of the time required to achieve it, which is 50 times greater than the 0.1%.

Based on the previous results, we expect that the predicted configurations have high correlation coefficients with the best configuration. That argument is true as it can be seen in the Table 5.9.

The coefficients are over 0.80 which means a strong similarity, and that also proves the fact that the test set was very focused along some specific applications.

Overall, the coefficient matrix calculated by the Pearson's correlation provides a strong learning base

Figure 5.12: RMSE for different number of features, with or without feedback based upon the learning base by Pearson's similarity.

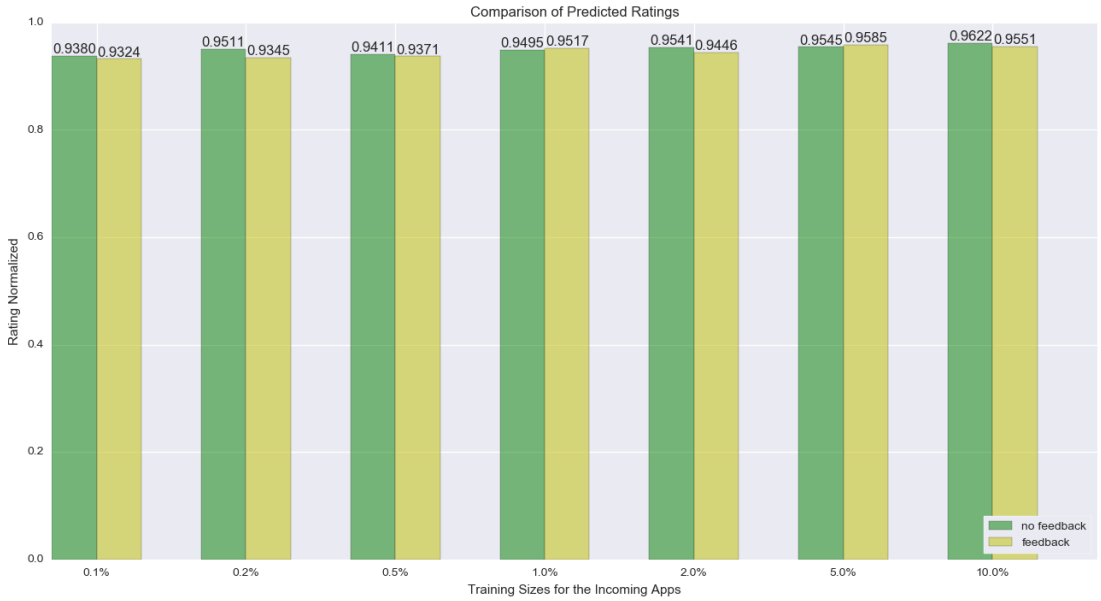


Figure 5.13: Predicted configurations ratings normalized to the best one with respect to training size.

	Training Sizes					
	0.1%		0.2%		5%	
	nfb	fb	nfb	fb	nfb	fb
min	0.70	0.64	0.67	0.71	0.72	0.72
max	0.98	0.98	0.98	0.82	0.92	0.93
average	0.85	0.80	0.83	0.78	0.84	0.84

Table 5.9: Minimum, maximum and average correlations of the 0.1%, 0.2% and 5% predicted configurations with the best one.

and consequently very accurate predictions as the model is able to use effectively every application.

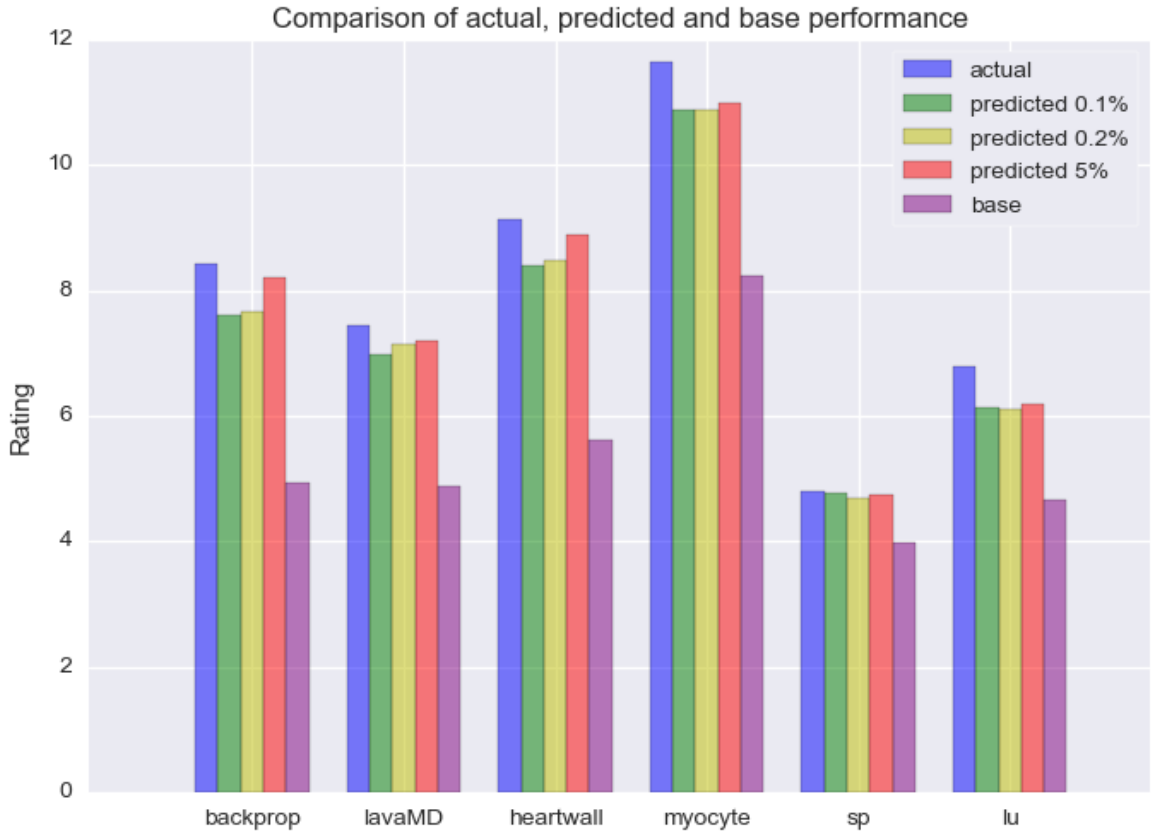


Figure 5.14: Performance comparison for 6 features, 0.1%, 0.2% and 5% known ratings, without feedback.

5.2 Comparison with the Brute Force Search

In order to show the efficiency of our autotuner we need to compare him with the brute force search over this large exploration space. Two variables are taken into account, time needed and performance achieved relative to the best of the tuning configurations. We use the rating 1 and the applications we used in that evaluation *hotspot3D, lavaMD, myocyte, ft, sp* and *lu* for the comparison. We examine two training sizes, 0.2% and 1% with and without feedback. The time required is calculated as the average execution time of each application over the whole tuning space multiplied by 4, which is the times needed to take every performance counter required and following multiplied by the size of the tuning

configurations' subset for the autotuner and by the whole tuning space, i.e. 2880 configurations, for the brute force.

Figure 5.15 shows how the autotuner without feedback performs along with the brute force search and Figure 5.16 shows the same but the autotuner uses feedback.

The mapping of the applications is: 1. *hotspot3D*, 2. *lavaMD*, 3. *myocyte*, 4. *ft*, 5. *sp* and 6. *lu*.

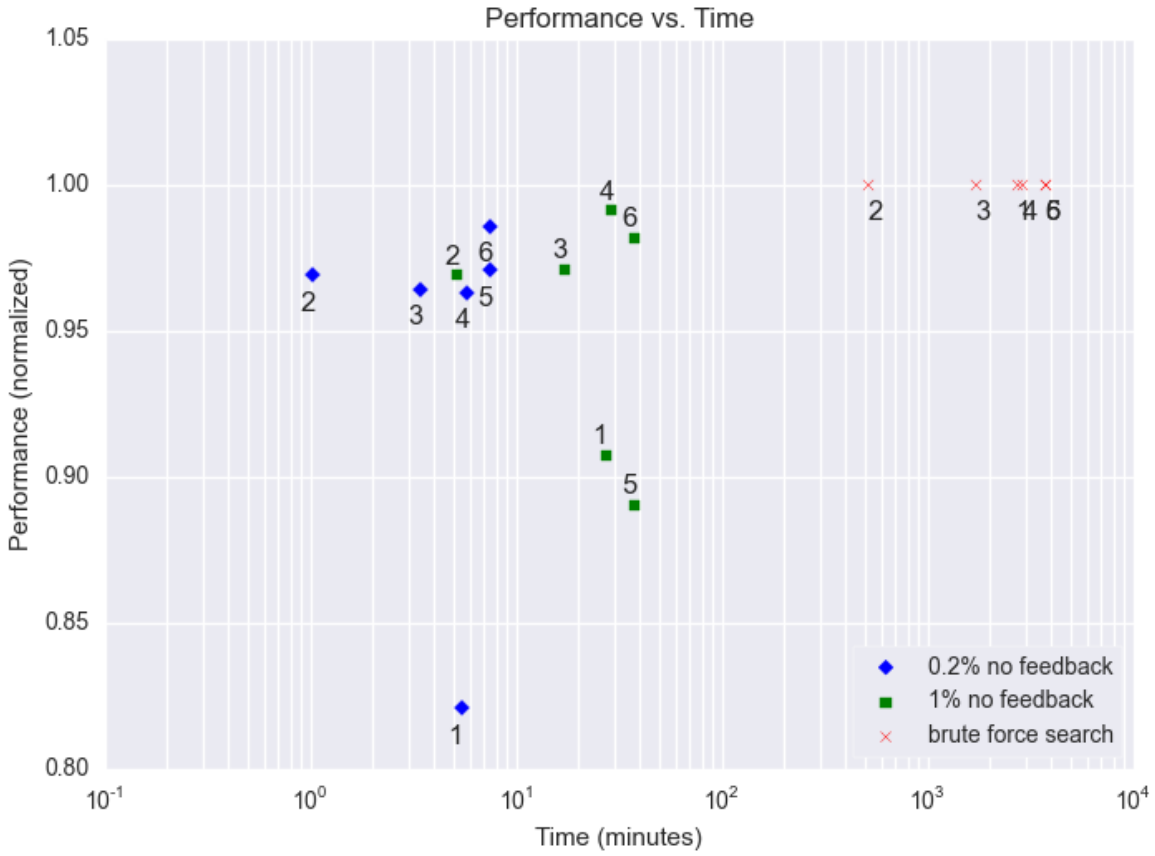


Figure 5.15: Performance vs Time, autotuner without feedback, 0.1% and 2% training sizes

We observe that with 0.2% training size for every prediction we need less than 8 minutes while we achieve 82% to 98% performance relative to the best one. For the 1% training size we need maximum 40 minutes (5 times more configurations) to reduce the range of the performance achieved between 89% and 99%. The brute force search is prohibitively slow. Particularly, it requires 500 to 3800 minutes for each application in order to improve the performance by 22% to a mere 1%. Thus, it is apparent that a 62.5 fold increase in time does not compensate for a 22% improvement in performance.

For the version with feedback, the times required for the predictions are the same as before however the performances have less variations and are more focused around 90%. Specifically, the 0.2% training size reaches performances of 99% with 94.5% on average and the 1% further reduces its range between 92% and 99% with an average of 96.5%. More than before, the performances from the predictions in less than 8 minutes and even those in less than 40 minutes, value more than the best ones that need 8.5 to 63 hours.

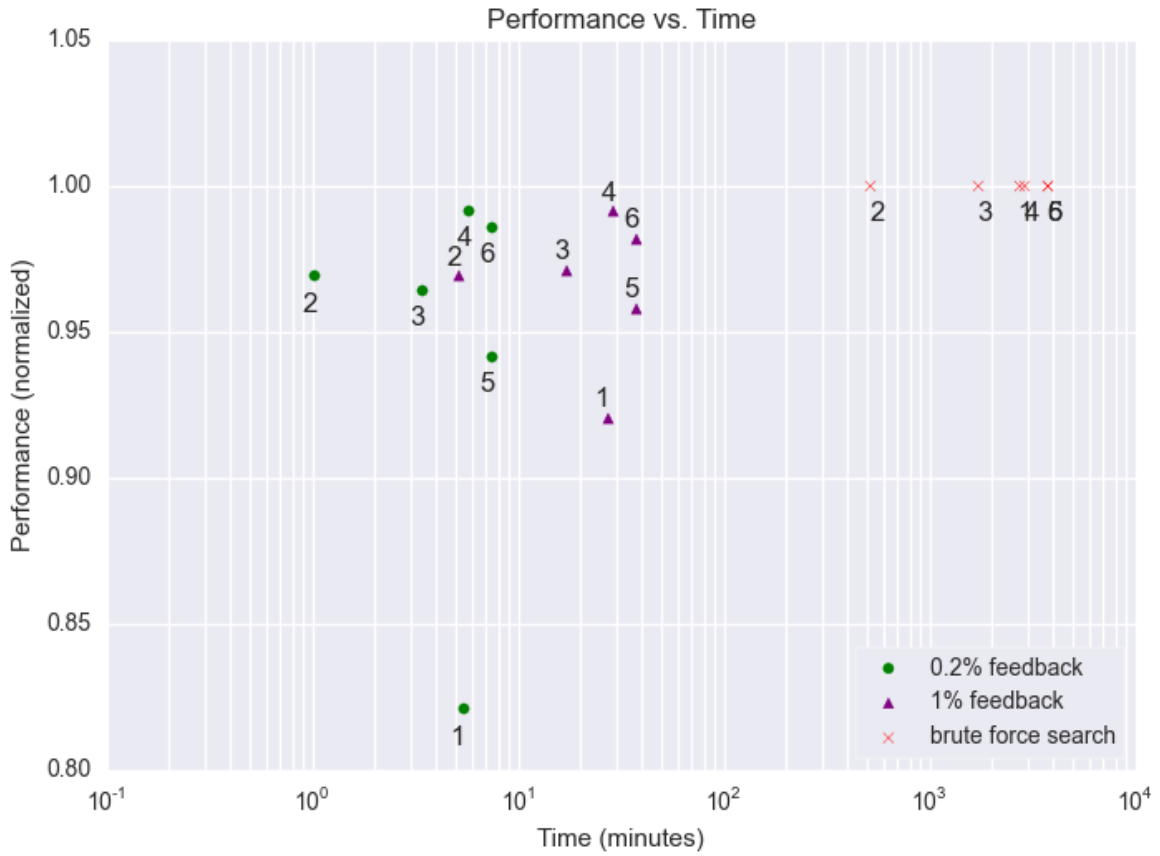


Figure 5.16: Performance vs Time, autotuner with feedback, 0.1% and 2% training sizes

5.3 Energy Aware and Unaware Predictions

In this section we compare the tuning configuration predictions from the energy aware rating 1 with the ones from non aware $r_3^{ui} = \frac{MFlops}{sec}$(Normalized to the base). The test files are the same in both versions and we evaluate also the power savings.

By executing the autotuner with rating 5.3 for the test applications *hotspot3D, lavaMD, myocyte, ft, sp* and *lu* we get the following tuning configurations along with the relative ratings, Table 5.11. We use 10% of the total tuning configurations as the training size. We notice that the power consumption of the coprocessor agree with the ratings, e.g. energy aware configurations consume less power than the non aware. The major variations in power seem to happen proportionally to the physical cores and hardware threads activated on the card. The power range depending on the physical cores:

19 : 100-125 Watts

38 : 140-165 Watts

57 : 135-200 Watts

The performance reported is $\frac{MFlops}{sec}$ normalized. If we calculate the percentage change of the energy aware configurations from the non aware we get the following results, Table 5.10. We see that from the predicted configurations we have on average 7.49% power savings for only 0.8% raw performance degradation. That result is important as it demonstrates also energy efficiency of the coprocessor and

	Power	Performance
hotspot3D	-16.30%	-6.91%
lavaMD	-15.76%	-8.70%
myocyte	0.95%	1.40%
ft	0%	0.01%
sp	-6.32%	8.62%
lu	-7.50%	0.80%
Average	-7.49%	-0.80%

Table 5.10: Percentage change of raw performance between energy aware and unaware predictions.

shows also that an energy aware approach has more advantages and is limited in deteriorating raw performance.

	Tuning Configurations										
	opt	prefetch	sstores	cache evict	unroll	huge pages	affinity	cores	threads/core	Power	Performance
hotspot3D											
Actual Energy	2	0	always	2	✓	✓	scatter	19	2	113.0 W	111.00
Actual	2	0	always	3	✓	✓	scatter	19	2	113.0 W	111.47
Predicted 10% Energy	3	0	always	3	-	✓	balanced	19	3	113.0 W	101.37
Predicted 10%	2	0	always	3	✓	-	scatter	57	2	135.0 W	108.89
lavaMD											
Actual Energy	3	4	always	2	✓	-	balanced	38	2	150.0 W	90.90
Actual	2	4	always	1	✓	-	balanced	57	4	185.0 W	111.11
Predicted 10% Energy	2	2	always	0	✓	✓	balanced	38	4	155.0 W	91.01
Predicted 10%	2	4	always	0	✓	✓	scatter	57	4	184.0 W	99.68
myocyte											
Actual Energy	3	2	never	-	-	✓	balanced	19	4	104.0 W	119.37
Actual	2	0	never	-	✓	-	balanced	19	4	109.0 W	120.24
Predicted 10% Energy	3	0	always	0	-	-	balanced	19	3	106.0 W	118.12
Predicted 10%	3	0	always	1	-	✓	balanced	19	4	105.0 W	116.50
ft											
Actual Energy	2	2	always	0	-	-	scatter	57	2	199.0 W	136.44
Actual	3	3	always	0	✓	-	scatter	57	2	203.0 W	137.46
Predicted 10% Energy	2	4	always	0	-	-	scatter	57	2	200.0 W	135.88
Predicted 10%	2	2	always	0	✓	✓	scatter	57	2	200.0 W	135.90
sp											
Actual Energy	2	2	always	1	✓	-	balanced	38	2	161.0 W	110.23
Actual	2	4	always	3	✓	✓	scatter	57	2	177.0 W	113.95
Predicted 10% Energy	2	3	always	1	✓	-	balanced	38	2	163.0 W	108.77
Predicted 10%	3	3	never	-	✓	-	scatter	57	2	174.0 W	100.14
lu											
Actual Energy	2	2	never	-	✓	✓	balanced	38	2	146.0 W	108.69
Actual	2	2	always	0	✓	-	scatter	57	2	162.0 W	111.18
Predicted 10% Energy	2	3	always	1	✓	-	balanced	38	2	148.0 W	108.12
Predicted 10%	3	4	always	0	✓	✓	scatter	57	2	160.0 W	107.26

Table 5.11: Best configurations both actual and predicted for energy aware and non-aware ratings.

Chapter 6

Discussion

6.1 General Assessment

Our autotuner proves to be effective in various environments. While varying the learning base, the rating formula and the training size. However, caution should be exercised when tuning the stochastic gradient descent's parameters, i.e. learning rates and regulators, so that the model avoids overfitting the training data and at the same time converges to satisfactory values. The predicted configurations surpass the base performance and exhibit fine tuning for each application specifically. That is a very important feature as for a developer to fine tune an application is a very toilsome task. He needs to run performance analysis on each version of the application he makes in order to find critical sections and modify them accordingly. He achieves higher performance gradually and not instantly.

In addition, the autotuner is relative fast once he is online. The offline part is the most time consuming but the advantage is that it needs to happen only once. If the learning base consists of 14 applications, with an average execution time on the coprocessor of 15 seconds, then for 2,880 tuning configurations and 4 executions each we get a sum of 2,419,200 seconds or 672 hours. Thereafter, for each new incoming application we need a partial profiling over the 0.1%-1% of the total tuning configurations. That step needs 4-40 minutes maximum and the autotuner returns in less than 30 seconds the predicted best configuration. Depending on the performance levels we aim for, the training size varies and consequently the time needed in the online stage. Yet, always we exceed the 90% of the best performance. Besides the best configuration someone may ask the predicted rating/performance for a particular tuning configuration which is also provided by the model.

Lastly, if we use feedback we expect to get improved predictions as the model further learns from the incoming applications. In our experiments, we did not notice a major improvement because the test size was small and the effect of the feedback was not propagated. However, it is a valid argument a should be taken as plausible.

Table 6.1 shows the average percentage levels of the predicted configurations that were tested in our experiments.

To compare with other tuning tools, the autotuner presents many advantages. Firstly, the manual guidelines are not able to provide assistance for particular applications, instead they suggest general modifications and execution environments for the majority of the applications. Certainly, they do not guide to the best tuning configuration. Secondly, the autotuner is able to respond for the best configuration over a very large space of tuning exploration in a timely fashion outperforming iterative searching tools. Even though the iterative searching tools guarantee the best configuration, our autotuner constantly reports a configuration achieving more than the 90% performance of the best one, a very important realization.

As a tool our autotuner was built to tune applications that run natively on the Intel Xeon Phi Coprocessor. However, the platform is not a restriction and by modifying the execution and the profiling for

Training size	Best Rating	Base Rating
0.1%	90.06%	131.42%
0.2%	93.39%	136.28%
0.5%	95.06%	138.71%
1%	94.53%	137.94%
2%	95.43%	139.25%
5%	95.47%	139.31%
10%	95.85%	139.87%

Table 6.1: Average percentages of the predicted configurations from the best and base ratings.

a different architecture the autotuner can easily be adapted and work efficiently.

6.2 Future Work

The current Autotuner can be expanded in various directions. Firstly, it can be extended by evaluating also concurrent execution on the Intel Xeon Phi Coprocessor. That means to be able and measure the interference between the executing applications and assign them to exclusive sets of cores, under performance and power restrictions. Furthermore, the host, the Intel Xeon Processor, can also be ”recruited” and change the approach of application execution. The main running environment changes to the processor and the coprocessor is used for offloading computing intensive parts of the code. So, the Autotuner has to monitor the execution both on the processor and on the coprocessor as well as explore the tuning configurations of the processor and exploit his architecture. Then the Autotuner can acquire a generic profile by supporting different architectures and modes of execution, extending also to GPUs and other multicore processors.

The employment of machine learning into the automatic tuning is very propitious approach and we proved that we can achieve great results.

Bibliography

[1] Automatically tuned linear algebra software, atlas project. http://math-atlas.sourceforge.net/. online; accessed 29-April-2016.

[2] Events for intel xeon phi coprocessor (code name: Knights corner). https://software.intel.com/en-us/node/589941. online, accessed 10-May-2016.

[3] *gprof(1) Linux user's manual*.

[4] Intel many integrated core architecture. https://software.intel.com/mic-developer. online; accessed 28-April-2016.

[5] Intel©optimized linpack benchmark for linux* os. https://software.intel.com/en-us/node/528615. online, accessed 18-May-2016.

[6] Intel©xeon phi™coprocessor linpack* and stream* performance. http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-linpack-stream.html. online, accessed 18-May-2016.

[7] Library libhugetlb. https://github.com/libhugetlbfs/libhugetlbfs. online, accessed 31-May-2016.

[8] Linpack benchmark. http://www.netlib.org/benchmark/hpl/. online, accessed 17-May-2016.

[9] Optimized sparse kernel interface library. http://bebop.cs.berkeley.edu/oski/. online; accessed 29-April-2016.

[10] Optimizing memory bandwidth on stream triad. https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad. online, accessed 18-May-2016.

[11] Paradyn performance measurement tool. http://www.paradyn.org/. online; accessed 29-April-2016.

[12] Readex. runtime exploitation of application dynamism for energy-efficient exascale computing. http://www.readex.eu/. online; accessed 4-May-2016.

[13] Scalasca tool. http://www.scalasca.org/. online; accessed 29-April-2016.

[14] Stream benchmark. http://www.cs.virginia.edu/stream/. online, accessed 17-May-2016.

[15] Top500 list. http://top500.org/. online; accessed 28-April-2016.

[16] User and reference guide for the intel® c++ compiler 14.0. https://software.intel.com/en-us/compiler_14.0_ug_c.

[17] Vampir tool. https://www.vampir.eu/. online; accessed 28-April-2016.

[18] X-tune. autotuning for exascale: Self-tuning software to manage heterogeneity. http://ctop.cs.utah.edu/x-tune/. online; accessed 4-May-2016.

[19] Slowdown. *The Economist*, 418(8980):69–70, March 2016.

[20] Martin S. Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[21] Cédric Andreolli, Philippe Thierry, Leonardo Borges, Gregg Skinner, and Chuck Yount. Chapter 23 - characterization and optimization methodology applied to stencil computations. In James Reinders and Jim Jeffers, editors, *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume 1, pages 377 – 396. Morgan Kaufmann, Boston, MA, USA, 2015.

[22] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. *SIGPLAN Not.*, 44(6):38–49, June 2009.

[23] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[24] Siegfried Benkner, Franz Franchetti, Hans Michael Gerndt, and Jeffrey K. Hollingsworth. Automatic Application Tuning for HPC Architectures (Dagstuhl Seminar 13401). *Dagstuhl Reports*, 3(9):214–244, 2014.

[25] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *Innovative Parallel Computing (InPar), 2012*, pages 1–9, May 2012.

[26] Shirley Browne, Christine Deane, George Ho, and Phil Mucci. Papi: Performance application programming interface. http://icl.cs.utk.edu/papi/index.html. online; accessed 28-April-2016.

[27] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[29] Chun Chen, Jacqueline Chame, and Mary W. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, Jun 2008.

[30] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society.

[31] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2008.

[32] Christina Delimitrou and Christos Kozyrakis. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *ACM Transactions on Computer Systems (TOCS)*, 2013.

[33] Jianbin Fang, Henk Sips, Lilun Zhang, Chuanfu Xu, Yonggang Che, and Ana Lucia Varbanescu. Test-driving intel xeon phi (best paper award). In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE'14)*, March 2014.

[34] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[35] Simon Funk. Netflix update: Try this at home. http://sifter.org/~simon/journal/20061211.html, December 2006. online; accessed 28-April-2016.

[36] Karl Fürlinger. *Encyclopedia of Parallel Computing*, chapter OpenMP Profiling with OmpP, pages 1371–1379. Springer US, Boston, MA, 2011.

[37] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 6 2011.

[38] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[39] Michael Gerndt, Eduardo César, and Siegfried Benkner, editors. *Automatic Tuning of HPC Applications, The Periscope Tuning Framework*. Shaker Verlag, April 2015.

[40] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):3:1–3:45, January 2009.

[41] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 123–132, Washington, DC, USA, 2005. IEEE Computer Society.

[42] Wim Heirman, Trevor E. Carlson, Kenzo Van Craeynest, Ibrahim Hur, Aamer Jaleel, and Lieven Eeckhout. Automatic smt threading for openmp applications on the intel xeon phi co-processor. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, pages 7:1–7:7, New York, NY, USA, 2014. ACM.

[43] Intel. Intel® vtune™amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe. online; accessed 28-April-2016.

[44] Intel. *Intel Xeon Phi Coprocessor System Software Developers Guide*, March 2014. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-system-software-developers-guide.

[45] Intel. *Intel Xeon Phi Coprocessor x100 Product Family, Datasheet*, April 2015. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html.

[46] Jim Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., Boston, MA, USA, 2013.

[47] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[48] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[49] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning–based optimising compilation. volume 11, pages 14:1–14:32, New York, NY, USA, Feb 2014. ACM.

[50] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets*. Cambridge university Press, 2014.

[51] C. K. Luk, R. Newton, W. Hasenplaugh, M. Hampton, and G. Lowney. A synergetic approach to throughput computing on x86-based multicore desktops. *IEEE Software*, 28(1):39–50, Jan 2011.

[52] Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, and François Bodin. *Applied Parallel and Scientific Computing: 11th International Conference, PARA 2012, Helsinki, Finland, June 10-13, 2012, Revised Selected Papers*, chapter AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications, pages 328–342. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[53] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. Mate: Monitoring, analysis and tuning environment for parallel/distributed applications. *Concurrency and Computation: Practice and Experience*, 19(11):1517–1531, 2007.

[54] Futoshi Mori, Masaharu Matsumoto, and Takashi Furumura. *High Performance Computing for Computational Science – VECPAR 2014: 11th International Conference, Eugene, OR, USA, June 30 – July 3, 2014, Revised Selected Papers*, chapter Performance Optimization of the 3D FDM Simulation of Seismic Wave Propagation on the Intel Xeon Phi Coprocessor Using the ppOpen-APPL/FDM Library, pages 66–76. Springer International Publishing, Cham, 2015.

[55] Yoonju Lee Nelson, B. Bansal, M. Hall, Aiichiro Nakano, and K. Lerman. Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[56] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer US, Boston, MA, 2011.

[57] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: adaptive control of distributed applications. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 172–179, Jul 1998.

[58] A. Sclocco, H. E. Bal, J. Hessels, J. v. Leeuwen, and R. V. v. Nieuwpoort. Auto-tuning dedispersion for many-core accelerators. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 952–961, May 2014.

[59] Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huynh, Xibai Li, and Rick Siow Mong Goh. Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 136–145, Washington, DC, USA, 2015. IEEE Computer Society.

[60] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[61] A. Tiwari and J. K. Hollingsworth. Online adaptive code generation and tuning. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 879–892, May 2011.

[62] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[63] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.

[64] Samuel Williams, Dhiraj D. Kalamkar, Amik Singh, Anand M. Deshpande, Brian Van Straalen, Mikhail Smelyanskiy, Ann Almgren, Pradeep Dubey, John Shalf, and Leonid Oliker. Optimization of geometric multigrid for emerging multi- and manycore processors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 96:1–96:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[65] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[66] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353349.

[67] Claude J. Wright. Chapter 14 - power analysis on the intel xeon phi coprocessor. In James Reinders and Jim Jeffers, editors, *High Performance Parallelism Pearls: Multicore and Manycore Programming Approaches*, volume 1, pages 239 – 253. Morgan Kaufmann, Boston, MA, USA, 2015.

Appendix

6.3 Source Code

The source code of the *Autotuner* tool that was developed for the purpose of this diploma thesis can be found at `https://github.com/LefterisChris/thesis-NTUA`. The code is licensed under the GPLv3 licence and can be modified and redistributed under these terms.