ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Optimization of GPU Workloads using Natural Language Processing based on Deep Learning techniques

Μελέτη και υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Πέτρου Βαβαρούτσου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
Αθήνα, Φεβρουάριος 2022

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων

Optimization of GPU Workloads using Natural Language Processing based on Deep Learning techniques

Μελέτη και υλοποίηση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Πέτρου Βαβαρούτσου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25^{η} Φεβρουαρίου, 2022.

....	....	....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2022

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

στην οικογένεια μου

Περίληψη

Η ρύθμιση των παραμέτρων ενός προγράμματος είναι δύσκολη λόγω της αφηρημένης σχέσης μεταξύ υλικού και λογισμικού. Απαιτούνται αλγόριθμοι αυτόματης βελτιστοποίησης που είναι ακριβείς για την αντιμετώπιση της πολυπλοκότητας και της ποικιλίας του τρέχοντος υλικού και λογισμικού. Η αυτόματη βελτιστοποίηση κώδικα βασιζόταν πάντα σε χρονοβόρες προσεγγίσεις δοκιμών και σφαλμάτων. Η μηχανική μάθηση (ML) και η Επεξεργασία Φυσικής Γλώσσας (NLP) έχουν ακμάσει την τελευταία δεκαετία με την έρευνα να επικεντρώνεται σε αρχιτεκτονικές βαθιών νευρωνικών δικτύων. Σε αυτό το πλαίσιο, η χρήση τεχνικών επεξεργασίας φυσικής γλώσσας στον πηγαίο κώδικα για τη διεξαγωγή εργασιών αυτόματης βελτιστοποίησης είναι ένα αναδυόμενο πεδίο μελέτης. Ενώ η προηγούμενη έρευνα έχει ολοκληρώσει επιτυχώς μια ποικιλία διαφορετικών εργασιών αυτόματης βελτιστοποίησης χρησιμοποιώντας μια ποικιλία διαφορετικών γλωσσών πηγαίου κώδικα, στη πλειονότητα των δεδομένων βρίσκεται πηγαίος κώδικας CPU, και όχι τόσο κώδικας GPU.

Σε αυτή την εργασία, κάνουμε δύο συνεισφορές. Αρχικά χρησιμοποιούμε το σύνολο δεδομένων των πυρήνων OpenCL από την εργασία των Cummings et al. [1] για να αξιολογήσουμε και να συγκρίνουμε έξι διαφορετικά προτεινόμενα βαθιά νευρωνικά δίκτυα με το state-of-the-art δίκτυο. Το καλύτερο μοντέλο μας ξεπερνά αυτό της δουλειάς των Cummins et al., παρέχοντας, συνολικά, 2.65% βελτίωση στην ακρίβεια πρόβλεψης.

Στη δεύτερη συνεισφορά μας, επεκτείνουμε την έρευνά μας στους πυρήνες CUDA και δημιουργούμε μια end-to-end μεθοδολογία που ενσωματώνει έναν source-to-source μεταγλωττιστή για thread και block coarsening μετασχηματισμούς πυρήνων CUDA, έναν CUDA rewriter που αφαιρεί σημασιολογικά άσχετες πληροφορίες από τους πυρήνες, τη δημιουργία έτοιμων για εκπαίδευση ακολουθιών, ένα profiling εργαλείο για τη μέτρηση της απόδοσης των μετασχηματισμένων πυρήνων, την παραγωγή των ετικετών πρόβλεψης και, τέλος, το καλύτερο μοντέλο μηχανικής εκμάθησης από την προαναφερθείσα έρευνα αρχιτεκτονικής νευρωνικών δικτύων στους πυρήνες OpenCL. Ως εκ τούτου, η προτεινόμενη μεθοδολογία μας λαμβάνει έναν χειρόγραφο πυρήνα CUDA και προβλέπει το βέλτιστο coarsening transformation factor του. Από όσο γνωρίζουμε, αυτή είναι η πρώτη εργασία που επιχειρεί να εφαρμόσει τεχνικές NLP σε γραπτές εφαρμογές CUDA για τις συγκεκριμένες βελτιστοποιήσεις.

Αξιολογούμε τη μεθοδολογία μας στο σύνολο δεδομένων LS-CAT [2] για πέντε διαφορετικούς coarsening factors στη GPU υψηλής τεχνολογίας NVIDIA V100S, ανακαλύπτουμε τα τρωτά σημεία της και εξετάζουμε τη δυνατότητα εφαρμογής μηχανικής μάθησης για τρία διαφορετικά προβλήματα πρόβλεψης: δυαδική ταξινόμηση thread coarsening, thread και block coarsening ταξινόμηση πέντε κλάσεων. Το μοντέλο μας επιτυγχάνει ακρίβεια 84% στη δυαδική ταξινόμηση, ενώ έχει κακή απόδοση όταν πρόκειται για ταξινόμηση πέντε κλάσεων.

Λέξεις Κλειδιά — Νευρωνικά Δίκτυα, Μηχανική Μάθηση, Βαθειά Μηχανική Μάθηση, Επεξεργασία Φυσικής Γλώσσας, LLVM, Μεταγλωττιστές, GPU, CUDA, OpenCL, Αυτόματη Βελτιστοποίηση, Μηχανισμός Προσοχής

Abstract

Setting program parameters is challenging due to the abstract relationship between hardware and software. Automatic optimization algorithms that are accurate are required to cope with the complexity and variety of current hardware and software. Autotuning has always relied on time-consuming trial and error approaches. Machine learning (ML) and Natural Language Processing (NLP) has flourished over the last decade with research focusing on deep architectures. In this context, the use of natural language processing techniques to source code in order to conduct autotuning tasks is an emerging field of study. While previous research has successfully completed a variety of different autotuning tasks using a variety of different source code languages, the majority of source code data is CPU-centric, with relatively little GPU code.

In this work, we make two contributions. We first utilize the dataset of OpenCL kernels from the work of Cummins et al. [1] to evaluate and compare our proposed six different deep neural networks to the state-of-the-art network. Our best model surpasses that of Cummins et al. work, providing, in total, a 2.65% improvement in prediction accuracy.

In our second contribution, we extend our research to CUDA kernels and we create an end-to-end pipeline that incorporates a source-to-source compiler for thread and block coarsening transformations of CUDA kernels, a source rewriter that removes semantically irrelevant information from the kernels, creating train ready sequences, a profiling tool for measuring the performance of the transformed kernels, producing the prediction labels and, finally, our best machine learning model from our aforementioned neural network architecture research on OpenCL kernels. Hence, the pipeline receives a hand-written CUDA kernel and predicts its optimal configuration. To the best of our knowledge, this is the first work that attempts to apply NLP techniques on CUDA written applications for the specific optimizations.

We evaluate our methodology on the LS-CAT dataset [2] for five different coarsening factors on NVIDIA V100S high-end GPU, we discover its vulnerabilities and examine the applicability of machine learning for three different prediction problems: thread coarsening binary classification, thread and block coarsening five-class classification. Our model achieves 84% accuracy on the binary classification, while it performs poorly when it comes to five-class classification.

Keywords — Neural Networks, Machine Learning, Deep Learning, Natural Language Processing, LLVM, Compilers, GPU, CUDA, OpenCL, Autotuning, Attention

Ευχαριστίες

Θα ήθελα, καταρχάς, να ευχαριστήσω τον καθηγητή μου κ. Δημήτριο Σούντρη για την επίβλεψη αυτής της διπλωματικής εργασίας και την ευκαιρία που μου έδωσε να την εκπονήσω στο εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων. Επίσης, ευχαριστώ ιδιαίτερα τους υποψήφιους διδάκτορες Δημοσθένη Μασούρο και Ιωάννη Ορούτζογλου για την καθοδήγησή τους και την εξαιρετική συνεργασία που είχαμε. Ακόμα, θα ήθελα να ευχαριστήσω όλους τους φίλους μου, που με στήριξαν καθ' όλη τη διάρκεια της προσπάθειας μου, καθώς γέμισαν τα φοιτητικά μου χρονιά με υπέροχες στιγμές, που θα μείνουν για πάντα χαραγμένες στη μνήμη μου. Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου για την καθοδήγηση και την ηθική συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια.

Βαβαρούτσος Πέτρος,
Αθήνα, 25 Φεβρουαρίου 2022

Contents

Figure List

Table List

Εκτεταμένη Ελληνική Περίληψη

0.1 Εισαγωγή

Μεγάλοι όγκοι δεδομένων που συλλέγονται και ανταλλάσσονται από επιχειρήσεις, κυβερνήσεις, βιομηχανίες, μη κερδοσκοπικούς οργανισμούς και επιστημονική έρευνα έχουν αυξήσει την κλίμακα και τον αριθμό των εργασιών διαχείρησης μεγάλων δεδομένων. Για να ικανοποιηθούν οι ανάγκες των μεγάλων δεδομένων, τόσο το υλικό όσο και το λογισμικό πρέπει να ενημερωθούν. Η μηχανική μάθηση είναι ένας κλάδος της τεχνητής νοημοσύνης και της επιστήμης των υπολογιστών που χρησιμοποιεί δεδομένα και αλγόριθμους για να μιμηθεί την ανθρώπινη μάθηση και να βελτιώσει την ακρίβεια. Υπάρχουν πολυάριθμες εργασίες σε γλώσσες προγραμματισμού όπου η μηχανική εκμάθηση μπορεί να είναι επωφελής. Ένα συχνό παράδειγμα του πώς η μηχανική μάθηση βοηθά στην ανάπτυξη καλύτερων προγραμμάτων είναι η χρονοβόρα διαδικασία δημιουργίας ευρετικών τρόπων βελτιστοποίησης, όπως είναι το loop unrolling. Αυτού του είδους τα προβλήματα είναι πολύπλοκα όταν η ευρετική επιλογή πρέπει να λαμβάνει υπόψη όχι μόνο το λογισμικό, αλλά και το υλικό-στόχο, όπως η κρυφή μνήμη εντολών. Αντί να συγκεντρώνει όλη αυτή τη γνώση, ένας προγραμματιστής μπορεί να επιλέξει τη διαδικασία της μηχανικής μάθησης, δηλαδή να χρησιμοποιήσει έναν αλγόριθμο μάθησης για να εξαγάγει μια ευρετική από εμπειρικά δεδομένα σχετικά με την απόδοση βρόχου σε διάφορες διαμορφώσεις (loop unrolling factors).

Σε αυτή τη διπλωματική εργασία, διερευνούμε προσεγγίσεις για την ενσωμάτωση δεδομένων κώδικα προγραμματισμού στην επεξεργασία φυσικής γλώσσας (NLP). Στόχος μας είναι να ανακαλύψουμε τεχνικές για την εκμετάλλευση σημαντικών ποσοτικοποιήσιμων ιδιοτήτων ή χαρακτηριστικών από προγραμμάτα, προκειμένου να προβλεφθούν οι βέλτιστες ευρετικές βελτιστοποίησης για παράλληλες εφαρμογές που θα μεγιστοποιήσουν την απόδοσή τους μέσω της διαδικασίας της μηχανικής εκμάθησης.

Δύο από τις πιο σημαντικές βελτιστοποιήσεις σε κώδικα GPU είναι το thread coarsening και το block coarsening. Το thread coarsening είναι μια τεχνική μετασχηματισμού κώδικα που συνδυάζει δύο ή περισσότερα γειτονικά νήματα στο ίδιο μπλοκ σε ένα μόνο νήμα. Έχει αποδειχθεί ότι είναι αποτελεσματικό για μια ποικιλία παράλληλων προγραμμάτων και αρχιτεκτονικών. Σε αυτήν την περίπτωση, μειώνει τον αριθμό των νημάτων ανά μπλοκ ενώ διατηρεί τον συνολικό αριθμό των μπλοκ που αρχικοποιήθηκαν. Είναι σε θέση να αυξήσει τον όγκο της εργασίας που εκτελείται από κάθε πυρήνα αναπαράγοντας τις οδηγίες στο σώμα του, ενώ ταυτόχρονα μειώνει τον αριθμό των νημάτων που δημιουργούνται κατά το χρόνο εκτέλεσης. Είναι ανάλογο με το loop unrolling εκτός από το ότι εφαρμόζεται σε παράλληλες εργασίες παρά σε επαναλήψεις σειριακού βρόχου. Το block coarsening, από την άλλη, είναι μια τεχνική μετασχηματισμού πυρήνα που συνδυάζει τους φόρτους εργασίας δύο ή περισσότερων μπλοκ νημάτων και επομένως μειώνει τον συνολικό αριθμό τους διατηρώντας τον αριθμό νημάτων ανά μπλοκ. Ως αποτέλεσμα, συγχωνεύει πολυάριθμα περιβάλλοντα μπλοκ για να αντιμετωπίσει τα ζητήματα που ενυπάρχουν στον εκτεταμένο fine-grained παραλληλισμό.

Ωστόσο, και οι δύο μετασχηματισμοί μπορούν να έχουν μια σειρά από ανεπιθύμητες παρενέργειες, συμπεριλαμβανομένης της μείωσης του ολικού παραλληλισμού και της αύξησης της πίεσης του καταχωρητή, τα οποία και τα δύο μπορεί να έχουν ως αποτέλεσμα την υποβάθμιση της απόδοσης. Ο προσδιορισμός του πότε και του τρόπου εφαρμογής του coarsening δεν είναι εύκολος, καθώς ο βέλτιστος coarsening factor ποικίλλει ανάλογα με το πρόγραμμα-στόχο, το μέγεθος εισόδου του και την αρχιτεκτονική υλικού στην οποία εκτελείται το πρόγραμμα.

Η παρούσα εργασία κάνει δύο συνεισφορές. Αρχικά, εξετάζουμε τους τρέχοντες αλγόριθμους βαθιάς εκμάθησης για το NLP εφαρμοσμένους σε πυρήνες OpenCL. Συγκεκριμένα, κατασκευάζουμε και δοκιμάζουμε έξι βαθιάς μάθησης αρχιτεκτονικές νευρωνικών δικτύων για να βελτιστοποιήσουμε τον χρόνο εκτέλεσης κάθε πυρήνα στο βέλτιστο υλικό (CPU ή GPU) και αποδεικνύουμε ότι οι προτεινόμενες αρχιτεκτονικές DNN ξεπερνούν τις σύγχρονες τεχνικές NLP έως και 4%. Δεύτερον, εξετάζουμε τη δυνατότητα εφαρμογής NLP σε CUDA πυρηνες στην καλύτερη αρχιτεκτονική DNN από το προηγούμενο βήμα. Προτείνουμε μία end-to-end μεθοδολογία που ενσωματώνει τη διαδικασία από CUDA κώδικα γραμμένο από άνθρωπο σε βελτιστοποίηση της απόδοσης του, χωρίς τη χρήση compiler ή ανθρώπινη επίβλεψη.

Οι σημαντικότερες συνεισφορές είναι: (1) προτείνουμε και πειραματιζόμαστε πάνω σε 6 διαφορετικές αρχιτεκτονικές νευρωνικών δικτύων (2) μια από τις προτεινόμενες αρχιτεκτονικές ξεπερνά μέχρι και κατά 4% την state-of-the-art προσέγγιση (3) αναπτύσσουμε έναν source-to-source compiler για αυτόματο μετασχηματισμό CUDA πυρήνων σε thread και block coarsening (4) αναπτύσσουμε έναν source rewriter που δέχεται έναν πυρήνα CUDA και τον επιστρέφει αφαιρώντας του την συντατικτά ασήμαντη πληροφορία (5) προτείνουμε μία μεθοδολογία tokenization για πυρήνες CUDA, που χωρίζει την είσοδο σε tokens, τα αντιστοιχίζει στα αντίστοιχα ακέραια identifiers και επιστρέφει ακολουθίες με αυτά (6) εφαρμόζουμε ένα μεγάλης κλίμακας profiling σε πυρήνες CUDA, μετρώντας τον χρόνο εκτέλεσης τους για μία λίστα από coarsening factors (7) προτείνουμε μία μέθοδο επαύξησης δεδομένων από πυρήνες CUDA (8) εφαρμόζουμε το profiling στο σύνολο δεδομένων LS-CAT [2] και πάνω σε αυτά τα αποτελέσματα εφαρμόζουμε machine learning για τρία διαφορετικά προβλήματα (9) δείχνουμε ότι η προτεινόμενη μεθοδολογία επιφέρει 84% ακρίβεια πρόβλεψης στο δυαδικό πρόβλημα και ένα μέσο speedup 1.6 στα προβληματα με τις 5 κλάσεις.

0.2 Σχετική Βιβλιογραφία

Οι Magni κ.α. [3] επιδεικνύουν πώς μπορούν να χρησιμοποιηθούν προσεγγίσεις μηχανικής μάθησης για την αυτόματη ανάπτυξη κατάλληλων στρατηγικών thread coarsening για μια ποικιλία συστημάτων GPU. Η προσέγγισή τους λαμβάνει υπόψη έξι παραμέτρους coarsening (1, 2, 4, 8, 16, 32). Ο στόχος είναι να κατασκευαστεί ένα μοντέλο μηχανικής μάθησης που μπορεί να προσδιορίσει εάν ένας πυρήνας OpenCL θα πρέπει να γίνει coarsened σε μια συγκεκριμένη αρχιτεκτονική GPU και, εάν ναι, ποιος coarsening factor θα πρέπει να χρησιμοποιηθεί. Μεταξύ των πολυάριθμων αλγορίθμων μηχανικής μάθησης που είναι διαθέσιμοι, επέλεξαν να μοντελοποιήσουν το πρόβλημα χρησιμοποιώντας ένα τεχνητό νευρωνικό δίκτυο.

Οι Grewe κ.α. [4] προτείνουν μια προβλεπτική προσέγγιση για την αντιστοίχιση των πυρήνων OpenCL στην καλύτερη συσκευή σε ετερογενή συστήματα CPU/GPU. Δημιουργούν δέντρα αποφάσεων χρησιμοποιώντας εποπτευόμενη μάθηση και έναν συνδυασμό στατικών και δυναμικών χαρακτηριστικών του πυρήνα. Τα στατικά χαρακτηριστικά της εφαρμογής εξάγονται χρησιμοποιώντας ένα προσαρμοσμένο LLVM πέρασμα του συντακτικού δέντρου, ενώ τα δυναμικά χαρακτηριστικά εξάγονται από το χρόνο εκτέλεσης OpenCL.

Οι Cummins et al. [1] πρωτοπόρησε στη χρήση νευρωνικών δικτύων για τη βελτιστοποίηση των μεταγλωττιστών. Η τεχνολογία DeepTune εξάγει αυτόματα σχετικά δεδομένα από τον ακατέργαστο πηγαίο κώδικα. Σε αντίθεση με προηγούμενες εργασίες, αυτή η προσέγγιση ενσωματώνει τον κώδικα προγράμματος απευθείας στα δεδομένα εκπαίδευσης. Χρησιμοποίησαν ένα βαθύ νευρωνικό δίκτυο για να εκτιμήσουν την κατάλληλη ευρετική τιμή για ορισμένες βελτιστοποιήσεις OpenCL απευθείας από προγράμματα εισόδου. Στην επεξεργασία φυσικής γλώσσας, τα δίκτυα μακροπρόθεσμης βραχυπρόθεσμης μνήμης (LSTM) είναι μια καλά εδραιωμένη αρχιτεκτονική. Αυτά τα δίκτυα μπορούν να ερμηνεύσουν ροές δεδομένων και να ανακαλέσουν προηγούμενα συμβάντα. Μπορούν να κατανοήσουν τη δομή του προγράμματος και εάν μια μεταβλητή είχε οριστεί προηγουμένως. Τα αποτελέσματα ξεπέρασαν προηγούμενες εργασίες που χρησιμοποιούσαν χειροποίητη εξαγωγή χαρακτηριστικών κώδικα. Η μελέτη τους δείχνει επίσης ότι τα ανώτερα στρώματα του νευρωνικού δικτύου αφαιρούν πτυχές ακατέργαστου κώδικα που είναι κυρίως ανεξάρτητες από την πρόκληση βελτιστοποίησης. Κατά συνέπεια, το DeepTune αυτοματοποιεί πλήρως τη αυτόματη εξαγωγή χαρακτηριστικών κώδικα.

Οι Ben-Nun κ.α. [5] το πήγαν ένα βήμα παραπέρα. Αναγνώρισαν ότι, ενώ μια token-based αναπαράσταση λειτουργεί καλά για διφορούμενη φυσική γλώσσα, μια αναπαράσταση που βασίζεται σε γράφημα λειτουργεί καλύτερα για γλώσσες προγραμματισμού. Αντιπροσωπεύουν τις οδηγίες ενός προγράμματος ως ακμές σε ένα γράφημα που απεικονίζει τις σχέσεις μεταξύ των μεταβλητών. Στη συνέχεια μαθαίνουν διανυσματικές αναπαραστάσεις για κάθε εντολή με βάση το περιβάλλον του γραφήματος. Μετά από αυτό, τα LSTM μπορούν να επεξεργαστούν ένα πρόγραμμα ως ακολουθία αυτών των διανυσμάτων. Παρόλο που αυτή είναι η πιο περίπλοκη αναπαράσταση, παραλείπει μεταβλητές, τύπους τελεστών και σειρά τελεστών. Ως αποτέλεσμα, δεν είναι σε θέση να αναπαράγει την ανάλυση ροής δεδομένων που χρησιμοποιείται συνήθως σε οποιονδήποτε σύγχρονο μεταγλωττιστή. Οι Brauckmann κ.α. [6] διευρύνουν αυτή την έννοια προσδιορίζοντας όχι μόνο τα διανύσματα που χρησιμοποιούνται για την αναπαράσταση των εντολών, αλλά και τον τρόπο με τον οποίο επεξεργάζονται από τον learner. Χρησιμοποιούν νευρωνικά δίκτυα που περνούν μηνύματα με μια κατάσταση για κάθε κόμβο. Αυτή η κατάσταση αποστέλλεται κατά μήκος των άκρων σε κάθε γείτονα, ο οποίος χρησιμοποιεί μια συνάρτηση εκμάθησης για να τη συγχωνεύσει στη δική του κατάσταση. Μετά από μερικούς γύρους μετάδοσης μηνυμάτων, η ευρετική τιμή καθορίζεται από μια συνάρτηση συνάθροισης εκμάθησης.

0.3 Εφαρμογή NLP σε OpenCL πυρήνες

0.3.1 Ορισμός Προβληματος

Βασίζουμε την έρευνά μας στο έργο των Cummins κ.α. [1]. Θα εστιάσουμε σε μία από τις περιπτωσιολογικές μελέτες τους, την Ετερογενή Χαρτογράφηση Συσκευών. Η OpenCL παρέχει μια βάση για ετερογενή παραλληλισμό που είναι ανεξάρτητος από την πλατφόρμα. Αυτό επιτρέπει στα προγράμματα OpenCL να εκτελούνται ομαλά σε μια μεγάλη ποικιλία συσκευών, από CPU έως GPU και FPGA. Δεδομένου ενός προγράμματος και μιας επιλογής πιθανών συσκευών εκτέλεσης, το ερώτημα είναι ποια συσκευή πρέπει να χρησιμοποιηθεί για τη μεγιστοποίηση της απόδοσης. Για ένα ετερογενές σύστημα CPU/GPU, αυτό αποτελεί μια δυαδική επιλογή.

0.3.2 Προτεινόμενες Αρχιτεκτονικές

Η αρχιτεκτονική που προτείνεται στο [1], δηλαδή το DeepTune, απεικονίζεται στην εικόνα 0.3.1. Αποτελείται από ένα στάδιο προεπεξεργασίας που ξαναγράφει τον πηγαίο κώδικα προκειμένου να αφαιρέσει σημασιολογικά άσχετες πληροφορίες και να τις μετατρέψει σε tokens, ένα μοντέλο γλώσσας που λαμβάνει ως είσοδο αυτά τα tokens και τα μετατρέπει σε embedding διανύσματα, διάστασης $D = 64$, περνώντας ένα δίκτυο LSTM δύο επιπέδων. Το τελικό στοιχείο του DeepTune είναι δύο πλήρως συνδεδεμένα επίπεδα τεχνητού νευρωνικού δικτύου. Το αρχικό επίπεδο αποτελείται από 32 νευρώνες. Κάθε πιθανή ευρετική απόφαση αντιπροσωπεύεται από έναν μόνο νευρώνα στο δεύτερο επίπεδο. Η αυτοπεποίθηση του μοντέλου ότι η σχετική επιλογή είναι σωστή αντιπροσωπεύεται από την ενεργοποίηση κάθε νευρώνα στο επίπεδο εξόδου.

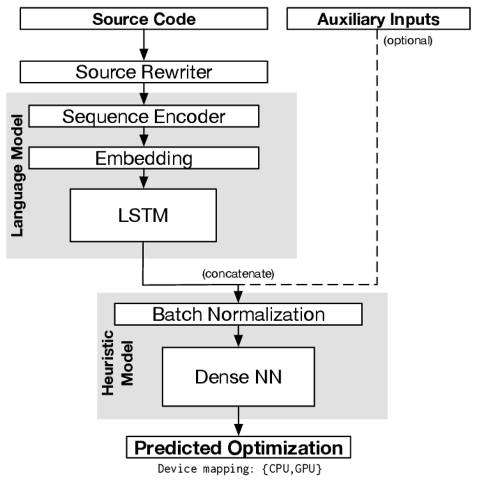


Figure 0.3.1: Επισκόπηση της αρχιτεκτονικής DeepTune [1]

Επεκτείνουμε το DeepTune προσθέτοντας ένα επίπεδο CNN, το οποίο φαίνεται στο σχήμα 0.3.2 και δέχεται ως είσοδο την έξοδο του embedding επιπέδου. Αποτελείται από 2 επίπεδα: ένα 1-d συνελικτικό επίπεδο, με 64 φίλτρα, μέγεθος πυρήνα 3 και βήμα βηματισμού 1, και ένα max pooling επίπεδο μεγέθους 4. Αυτή η μέθοδος εκμεταλλεύεται την εγγενή δομή μιας ετικέτας κειμένου με την αναγνώριση κοινών χαρακτηριστικών που μοιράζονται πολλές λέξεις, χωρίζοντας τες σε τρία ίσα μέρη. Το embedding επίπεδο μεταδίδει τις λέξεις στα συνελικτικά επίπεδα με τη μορφή προτάσεων (στην περίπτωσή μας οδηγίες κώδικα). Το επίπεδο συνέλιξης εφατμόζει την πράξη της συνέλιξης στην είσοδο χρησιμοποιώντας pooling επίπεδα. Τα επίπεδα pooling βοηθούν στη μείωση της αναπαράστασης των φράσεων εισόδου, των παραμέτρων εισόδου, των πράξεων υπολογισμού και της υπερπροσαρμογής στο δίκτυο (overfitting).

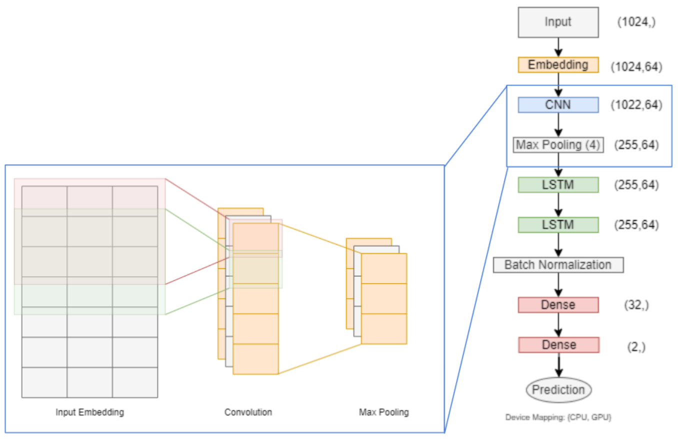


Figure 0.3.2: Αριστερά: Επισκόπηση της αρχιτεκτονικής του επιπέδου CNN. Δεξιά: Επισκόπηση της συνολικής αρχιτεκτονικής CNN-DeepTune

Στην εικόνα 0.3.3 βλέπουμε ομαδοποιημένα τις υπόλοιπες 5 αρχιτεκτονικές που προτείνουμε. Στο όνομα κάθε αρχιτεκτονικής αναφέρεται και η αντιστοιχη επέκταση στο DeepTune. Η αρχιτεκτονική CNN-BiLSTM-Attention αποτελεί μια αρχιτεκτονική που συνδυάζει τις προηγούμενες.

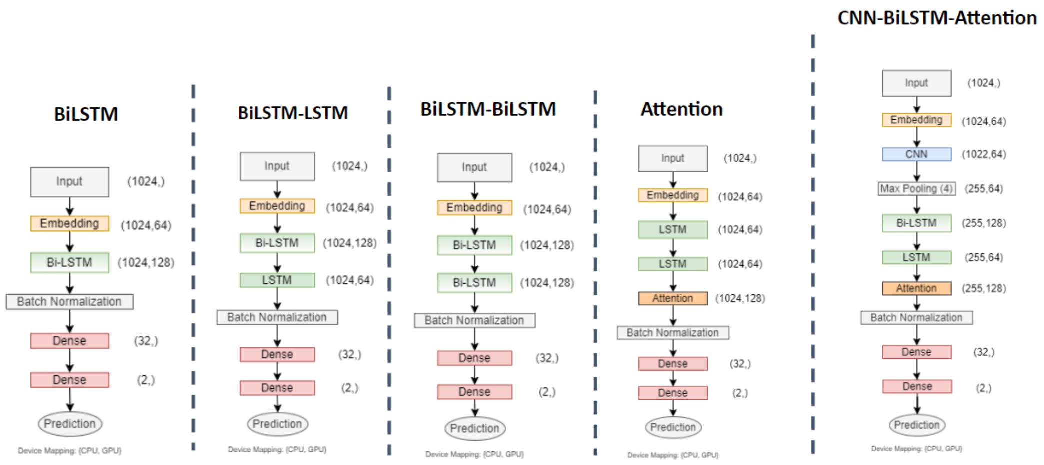


Figure 0.3.3: Αριστερά προς δεξιά: BiLSTM-DeepTune, BiLSTM-LSTM-DeepTune, BiLSTM-BiLSTM-DeepTune, Attention-DeepTune, CNN-BiLSTM-Attention

0.3.3 Πειράματα

Σύνολο Δεδομένων

Βασίζουμε την αξιολόγησή μας στο σύνολο δεδομένων που χρησιμοποιείται στο [1] προκειμένου να έχουμε ακριβή σύγκριση μεταξύ των διαφορετικών αρχιτεκτονικών μοντέλων. Το σύνολο δεδομένων περιέχει 680 προεπεξεργασμένους και tokenized πυρήνες OpenCL. Επιπλέον, περιέχει τους χρόνους εκτέλεσης για κάθε πυρήνα σε δύο συσκευές GPU, την AMD Tahiti 7970 και την NVIDIA GTX 970, καθώς και σε έναν επεξεργαστή Intel Core i7-3820. Το prediction target είναι η πλατφόρμα στην οποία ο χρόνος εκτέλεσης είναι μικρότερος. Πιο συγκεκριμένα, όταν εξετάζουμε τις περιπτώσεις AMD GPU και Intel CPU, ο στόχος είναι [1,0] εάν ο πυρήνας εκτελείται πιο γρήγορα στη GPU και [0,1] εάν ο πυρήνας τρέχει πιο γρήγορα στην CPU. Αυτό αναφέρεται επίσης ως κωδικοποίηση one-hot. Ομοίως, για την περίπτωση της NVIDIA GPU και της CPU της Intel.

Λεπτομέρειες Υλοποίησης

Χρησιμοποιούμε 10-fold cross validation για να αξιολογήσουμε την προβλεπτική ποιότητα κάθε μοντέλου. Κάθε πρόγραμμα εκχωρείται τυχαία σε ένα από τα δέκα σετ ίσου μεγέθους τα οποία είναι ισορροπημένα για να διασφαλιστεί μια σταθερή αναλογία δειγμάτων από κάθε κλάση σε ολόκληρο το σύνολο. Ένα μοντέλο εκπαιδεύεται σε όλα τα σετ εκτός από ένα και στη συνέχεια δοκιμάζεται στα προγράμματα από το σύνολο που δεν έχε δει. Αυτή η διαδικασία γίνεται για καθένα από τα δέκα σύνολα προκειμένου να παρέχεται μια ολοκληρωμένη πρόβλεψη για ολόκληρο το σύνολο δεδομένων.

Όλα τα μοντέλα υλοποιήθηκαν στην Python χρησιμοποιώντας τα backends Tensorflow[1] και Keras[2]. Για να εξασφαλίσουμε την πιο ακριβή σύγκριση, αρχικοποιούμε όλα τα επίπεδα με τα ίδια βάρη. Το μέγιστο μήκος ακολουθίας ορίστηκε σε 1024 και ο ρυθμός εκμάθησης στο 10^{-3}. Χρησιμοποιήσαμε categorical cross entropy ως συνάρτηση απώλειας, μέγεθος batch 64 και εκπαιδεύουμε για 50 εποχές. Ως εργαλείο βελτιστοποίησης, χρησιμοποιήσαμε τον προσαρμοστικό αλγόριθμο βελτιστοποίησης ρυθμού εκμάθησης, Adam. Τα πειράματα πραγματοποιήθηκαν σε GPU NVIDIA Tesla V100. Τέλος, χρησιμοποιήσαμε το TensorBoard[3] για να μετρήσουμε και να απεικονίσουμε παραμέτρους όπως η απώλεια και η ακρίβεια.

Αποτελέσματα

Στον πίνακα 1 απεικονίζονται συγκεντρωτικά τα αποτελέσματα για κάθε μοντέλο. Το DeepTune, το μοντέλο που προτείνεται στο [1], πέτυχε μέση ακρίβεια 81, 76%, με καλύτερα αποτελέσματα στην πλατφόρμα AMD, τα οποία είναι συνεπή σε όλες τις αρχιτεκτονικές. Το μοντέλο CNN-DeepTune ξεπερνά ήδη το baseline DeepTune μοντέλο κατά 2.65% στην πλατφόρμα της AMD και κατά 1.18% κατά μέσο όρο, αλλά έχει ελαφρώς χειρότερη απόδοση στην πλατφόρμα NVIDIA. Τα δύο επίπεδα bi-LSTM στο BiLSTM-BiLSTM-DeepTune μειώνουν την ακρίβεια του μοντέλου. Το BiLSTM-DeepTune επιτυγχάνει παρόμοια αποτελέσματα με το BiLSTM-LSTM-DeepTune στην πλατφόρμα NVIDIA, με το τελευταίο να έχει καλύτερες επιδόσεις στην πλατφόρμα AMD. Η προσθήκη ενός επιπέδου προσοχής στο μοντέλο Attention-DeepTune φαίνεται να δίνει τα καλύτερα αποτελέσματα, μέχρι στιγμής, στην πλατφόρμα NVIDIA. Ως αποτέλεσμα των προηγούμενων ευρημάτων, υποτέθηκε ότι η συγχώνευση των καλύτερων μοντέλων θα απέφερε τα καλύτερα αποτελέσματα. Το **προτεινόμενο** μοντέλο μας CNN-BiLSTM-Attention αποδεικνύει αυτήν την υπόθεση υπερτερώντας από το βασικό μοντέλο και όλα τα άλλα μοντέλα. Σημείωσε 4, 12% υψηλότερη σκορ από τη βασική γραμμή στην πλατφόρμα AMD και 1, 18% υψηλότερο στην πλατφόρμα NVIDIA, με αποτέλεσμα μέση αύξηση ακρίβειας 2, 65% τοις εκατό.

[1]Tensorflow: https://www.tensorflow.org/
[2]Keras: https://keras.io/
[3]TensorBoard: https://www.tensorflow.org/tensorboard/

	AMD Tahiti 7970	**NVIDIA GTX 970**	**Average**
DeepTune (baseline)	83.23%	80.29%	81.76%
CNN-DeepTune	85.88%	80.00%	82.94%
BiLSTM-DeepTune	82.80%	80.44%	81.62%
BiLSTM-LSTM-DeepTune	84.85%	80.88%	82.87%
BiLSTM-BiLSTM-DeepTune	83.38%	76.97%	80.14%
Attention-DeepTune	83.9%	81.03%	82.50%
CNN-BiLSTM-Attention	**87.35%**	**81.47%**	**84.41%**

Table 1: Accuracy of every model; Best results are highlighted in **bold** fold

0.4 Εφαρμογή NLP σε CUDA πυρήνες

0.4.1 Ορισμός Προβληματος

Σε αυτό το κεφάλαιο εστιάζουμε την έρευνα μας στην εφαρμογή των μετασχηματισμών βελτιστοποίσης που αναφέραμε παραπάνω, το thread coarsening και το block coarsening, πάνω σε πυρήνες CUDA. Θα εφαρμόσουμε αυτούς τους μετασχηματισμούς σε ένα σύνολο δεδομένων μεγάλης κλίμακας πυρήνων CUDA, που παρέχεται από την εργασία των Bjertnes κ.α. [2], θα μετρήσουμε τους χρόνους εκτέλεσης τους και θα εφαρμόσουμε μηχανική μάθηση σε αυτούς. Εξ όσων γνωρίζουμε, η εργασία αυτή είναι η πρώτη που θα μελετήσει τους μετασχηματισμούς αυτούς και θα εφαρμόσει μηχανική μάθησησ σε μεγάλης κλίμακας δεδομένα από CUDA kernels.

0.4.2 Προτεινόμενη Μεδοδολογία

Δεδομένου ενός συνόλου δεδομένων πυρήνων CUDA, πρέπει να είμαστε σε θέση να τους διαμορφώσουμε στις εργασίες μας, δηλαδή να δημιουργήσουμε ένα μοντέλο πρόβλεψης του factor για thread ή block coarsening. Για να γίνει αυτό πρέπει να μετατραπούν σε μια κατανοητή αναπαράσταση από υπολογιστή. Το σχήμα 0.4.1 απεικονίζει τη ροή ολόκληρης της μεθόδου που συζητείται στα ακόλουθα βήματα.

CUDA kernel transformation

Ο χειρόγραφος κώδικας μπορεί να περιλαμβάνει σχόλια που αποτελούνται από σημασιολογικά άσχετες πληροφορίες, μακροεντολές και μεταγλώττιση υπό όρους (π.χ. #define, #ifdef, κ.λπ.), κακή μορφή (ασυνέπεια σε εσοχές, κενά και νέες γραμμές, κ.λπ.) και αυθαίρετα ονόματα μεταβλητών και συναρτήσεων. Για το λόγο αυτό, αναπτύσσουμε μία μεθοδολογία, με τη χρήση του compiler toolchain Clang [7] με την οποία αφιαρούμε όλοι αυτή την πληροφορία. Επίσης, για να εξαλείψουμε αυθαίρετα ονόματα μεταβλητών και συναρτήσεων, αναπτύσσουμε ένα πρόγραμμα clang που ονομάζεται cuda-rewriter που αναλύει το AST και ξαναγράφει τον παρεχόμενο πηγαίο κώδικα με ομοιόμορφο τρόπο και με συνεπή στρατηγική ονομασίας για αναγνωριστικά. Χρησιμοποιούμε ένα σύστημα μετονομασίας αλφαβήτου με πεζά γράμματα $[a, b, \dots, z, aa, ab, \dots]$ για μεταβλητές που δηλώνονται εντός του πεδίου του πυρήνα και ένα σχήμα μετονομασίας αλφαβήτου με κεφαλαία γράμματα $[A, B, \dots, Z, AA, AB, \dots]$ για ονόματα συναρτήσεων που δηλώνονται εντός του πεδίου του πυρήνα.

Ο σκοπός της τυποποίησης των kernel είναι να βελτιστοποιήσει τη μοντελοποίηση του πηγαίου κώδικα διασφαλίζοντας ότι ασήμαντες σημασιολογικές αλλαγές στα προγράμματα, όπως η επιλογή ονόματος μεταβλητής ή η συμπερίληψη σχολίων, δεν έχουν καμία επίδραση στο διδασκόμενο μοντέλο.

Tokenization

Tokenization είναι η διαδικασία που μετατρέπει μια εισαγωγή κειμένου σε μια ακολουθία διακριτών ακεραίων, που αναφέρονται ως token ids. Έτσι, για να τροφοδοτηθεί το μοντέλο μηχανικής εκμάθησης με είσοδο, η κειμενική αναπαράσταση των CUDA kernels πρέπει να αναπαρασταθεί ως αριθμητικές ακολουθίες. Για αυτό χρησιμοποιούμε τον συντακτικό αναλυτή flex [8, 9]. Χρησιμοποιώντας σύνθετες κανονικές εκφράσεις μεταφράζουμε μεταβλητές, συναρτήσεις, αριθμητικές τιμές, τελεστές και χαρακτήρες σε μοναδικά

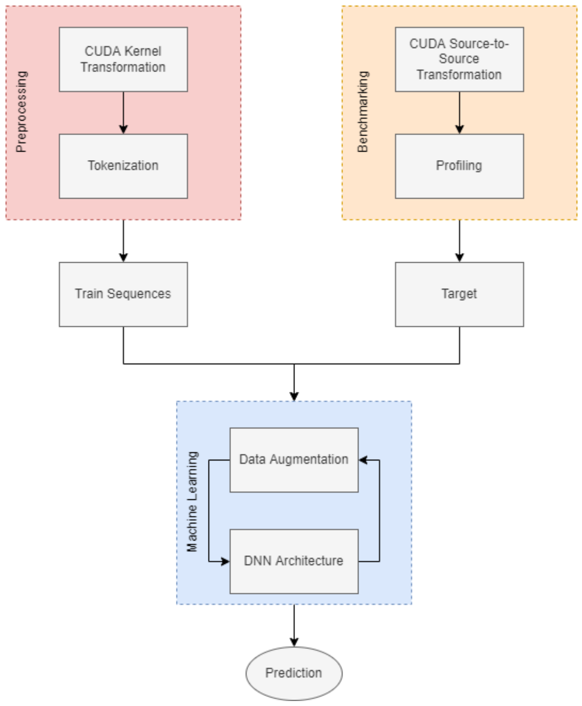


Figure 0.4.1: Προτεινόμενη μεθοδολογία για τη βέλτιστη αντιστοίχιση thread/block coarsening factor των CUDA kernels

tokens (π.χ. "int", "-", "++") και εκχωρούμε ένα μοναδικό ακέραιο αναγνωριστικό, σχηματίζοντας ένα λεξιλόγιο και αριθμητικές ακολουθίες. Υποθέτουμε ότι τα κενά, οι νέες γραμμές και η εσοχή είναι άσχετα με το νευρωνικό μοντέλο και ως εκ τούτου τα αγνοούμε, συντομεύοντας έτσι την ακολουθία και το λεξιλόγιο. Οι ακολουθίες συμπληρώνονται στην αρχή σε ένα προκαθορισμένο μήκος N με το ειδικό token '<pad>' και το αντίστοιχο αναγνωριστικό του, δηλαδή έναν αριθμό εκτός λεξιλογίου (π.χ. 0, -1). Εάν η ακολουθία υπερβαίνει τα N tokens, περικόπτεται στο τέλος, διατηρώντας τα πρώτα N. Η καταχώριση 1 παρέχει ένα παράδειγμα ενός πυρήνα CUDA. Ο πίνακας 2 περιέχει τα tokens που δημιουργούνται με την αντίστοιχη αντιστοίχιση ακέραιου αριθμού και ο πίνακας 3 περιέχει την αριθμητική ακολουθία που δημιουργείται, η οποία θα είναι η είσοδος του νευρωνικού μοντέλου.

```
__global__ void A(float *a, int *b) {
    int c = threadIdx.x + blockIdx.x * blockDim.x;
    float d = .1;
}
```

Listing 1: Απλό παράδειγμα ενός CUDA kernel

token	id	token	id	token	id
'__global__'	1	'int'	9	'x'	17
'void'	2	'b'	10	'+'	18
'A'	3	')'	11	'blockId'	19
'('	4	'{'	12	'blockDim'	20
'float'	5	'gid'	13	';'	21
'*'	6	'='	14	'c'	22
'a'	7	'threadId'	15	'0'	23
','	8	'.'	16	'}'	24

Table 2: Η αντιστοίχιση λεξιλογίου των tokens στα αντίστοιχα ακέραια αναγνωριστικά τους

<pad>	<pad>	<pad>	<pad>	<pad>	1	2	3
4	5	6	7	8	9	6	10
11	12	9	13	14	15	16	17
18	19	16	17	6	20	16	17
21	5	22	14	16	23	21	24

Table 3: Παραγόμενη αριθμητική ακολουθία

Αυτόματος Source to Source Compiler

Προκειμένου να δημιουργήσουμε έναν μεταγλωττιστή από πηγή σε πηγή που παίρνει ως είσοδο έναν πυρήνα και παράγει ένα thread ή ένα block coarsened πυρήνα, χρησιμοποιούμε τη γλώσσα Perl. Αρχικά, διασχίζουμε τον πυρήνα για να προσδιορίσουμε αν είναι μονοδιάστατος ή δισδιάστατος. Κατόπιν αυτού, αυξάνουμε τις παραμέτρους του προσθέτοντας τον coarsening factor για τους άξονες x και y (αν είναι δισδιάστατος), για τον οποίο κάνουμε έναν εμφωλευμένο βρόχο, που φαίνεται στο Listing 2. Τέλος, εφαρμόζουμε τις κατάλληλες τροποποιήσεις σε κάθε αντίστοιχο μετασχηματισμό, όπως φαίνεται στον πίνακα 4.

	Thread Coarsening	**Block Coarsening**
threadIdx.{x,y}	threadIdx.{x,y}*tc_{x,y}+index_{x,y}	threadIdx.{x,y}
blockDim.{x,y}	blockDim.{x,y}*tc_{x,y}	blockDim.{x,y}
blockIdx.{x,y}	blockIdx.{x,y}	blockIdx.{x,y}*bc_{x,y}+index_{x,y}
gridDim.{x,y}	gridDim.{x,y}	gridDim.{x,y}*bc_{x,y}

Table 4: Μετασχηματισμοί για thread και block coarsening

```
for (int indexx = 0; indexx < tc_x; indexx++)
    // If the kernel is one-dimensional the following line is not printed
    for (int indexy = 0; indexy < tc_y; indexy++)
```

Listing 2: Folded for loop

Large Scale Profiling

Για να επιτύχουμε το profiling σε μεγάλη κλίμακα πυρήνων CUDA, αναπτύξαμε δύο pipelines, ένα για thread coarsening και ένα για block coarsening, που μετασχηματίζει πυρήνες για μια σειρά coarsening factors, τους μεταγλωττίζει και τους εκτελεί και αποθηκεύει τον χρόνο εκτέλεσής τους. Η γλώσσα προγραματισμού Python χρησιμοποιήθηκε για την ανάπτυξη της διαδικασίας λόγω της απλότητας της αλληλεπίδρασης με καταλόγους, αρχεία και κείμενο.

0.4.3 Step 3: Proposed DNN Models

Για να αναπτύξουμε ένα μοντέλο μηχανικής μάθησης που προβλέπει τον βέλτιστο παράγοντα thread ή block coarsening, χρησιμοποιούμε το μοντέλο CNN-BiLSTM-Attention. Επιπλέον, προτείνουμε μια μεθοδολογία συνθετικής αύξησης δεδομένων, η οποία αποτελείται από δύο μέρη: μεταθέσεις των παραμέτρων του kernel και μετονομασία μεταβλητών.

0.4.4 Πειράματα

Σύνολο Δεδομένων

Bjertnes κ.α. [2] ανέκτησαν δημόσια προσβάσιμο πηγαίο κώδικα CUDA από το GitHub. Κάθε έργο αναδιαρθρώθηκε ως μια συλλογή εκτελέσιμων πυρήνων CUDA, αποδίδοντας 20251 μεταγλωττιζόμενους πυρήνες. Για να φτιάξουμε ένα μοντέλο πρόβλεψης με βάση τους συντελεστές coarsening, μας ενδιαφέρει ο χρόνος εκτέλεσης και ως εκ τούτου πρέπει να εφαρμόσουμε τη μεθοδολογία που περιγράφεται στην προηγούμενη ενότητα σε κάθε ένα kernel.

Λεπτομέρειες Υλοποίησης

Οι μετρήσεις μεταγλώττισης, εκτέλεσης και χρόνου εκτέλεσης πραγματοποιήθηκαν όλες σε ένα σύστημα GPU NVIDIA Tesla V100S 32 GB. Επιλέξαμε να δημιουργήσουμε τεράστια benchmarks για όλους τους πυρήνες, επομένως ορίσαμε το μέγεθος εισόδου σε περίπου 256 MB ανά πίνακα εισόδου. Ορίζουμε όλες τις παραμέτρους εισόδου του πυρήνα (μεταβλητές και πίνακες) σε ουδέτερες τιμές (π.χ. 0 ή 1). Το μέγεθος του πλέγματος ορίστηκε σε 262144 blocks για μονοδιάστατους πυρήνες και σε 512 σε κάθε άξονα για δισδιάστατους πυρήνες. Ο συνολικός αριθμός νημάτων ανά μπλοκ ορίστηκε σε 1024 για μονοδιάστατους πυρήνες και σε 32 σε κάθε άξονα για δισδιάστατους πυρήνες. Οι coarsening factors που διερευνήθηκαν ήταν στην περιοχή {1, 2, 4, 8, 16} και στους δύο άξονες x και y (εάν υπάρχει ο άξονας y).

Βασίζουμε τη δημιουργία του host κώδικα κάθε πυρήνα στο profiling tool του έργου LS-CAT [2]. Επεκτείναμε τους host κώδικες synchronization barriers, για μέτρηση χρόνου εκτέλεσης και ρητό έλεγχο σφαλμάτων. Η εκχώρηση μνήμης και η κλήση πυρήνα παραμένουν αμετάβλητα.

Για τη διαμόρφωση του μοντέλου μηχανικής εκμάθησης χρησιμοποιήσαμε την ίδια ρύθμιση όπως πριν. Η μόνη διαφορά είναι στον στόχο πρόβλεψης και ως εκ τούτου στην έξοδο του μοντέλου. Εξετάζουμε την αποτελεσματικότητα του μοντέλου και των δεδομένων μας σε διαφορετικά προβλήματα πρόβλεψης:

1. Binary Classification
   - One-dimensional thread coarsened kernels
2. Multi-class Classification
   - One-dimensional thread coarsened kernels
   - One-dimensional block coarsened kernels

Αποτελέσματα

Εφαρμόζοντας την προτεινόμενη μεθολογία που περιγράγραψε νωρίτερα ανακαλύψαμε ότι το LS-CAT dataset περιέχει μόνο 3477 αξιοποίσημους CUDA kernels, από τους συνολικά 20251. Αυτό συμβαίνει επείδη περιέχονται πολλοί διπλότυποι kernels και κάποιοι έχουν λάθη κατά το runtime. Υποθέτουμε ότι η μεθοδολογία της δημιουργίας του dataset στο [2] δεν ήταν ικανή να αναγνωρίσει αυτές τις περιπτώσεις.

Για τους υπόλοιπους πυρήνες, λοιπόν, τα αποτελέσματα του profiling του speedup καθε factor σε σχέση με το να μην εφαρμόσουμε coarsening, για τους thread και block coarsening μετασχηματισμούς φαίνονται στις εικόνες 0.4.2, 0.4.3 και 0.4.4, 0.4.5 αντιστοιχα.

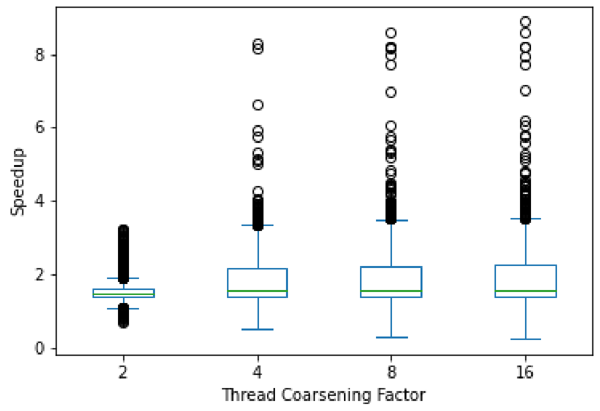
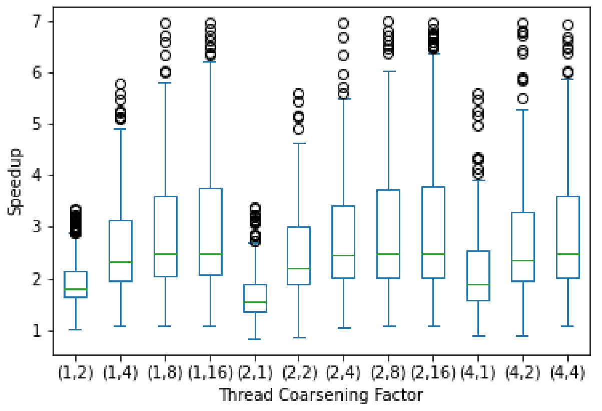
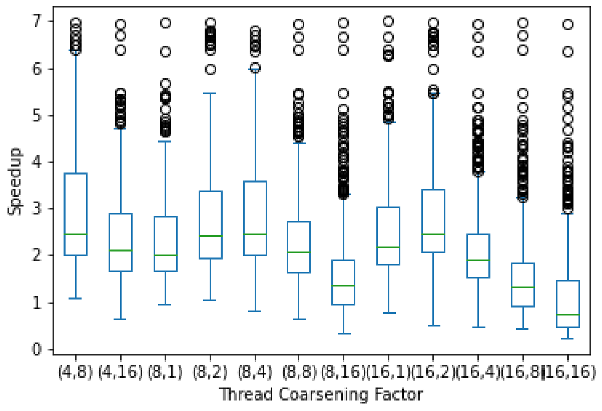


Figure 0.4.2: **One-dimensional kernels:** Distribution of speedup for thread coarsening factors {2, 4, 8, 16}



(a)



(b)

Figure 0.4.3: **Two-dimensional kernels:** (a) Distribution of speedup for thread coarsening factors {(1, 2) ... (4, 4)} (b) Distribution of speedup for thread coarsening factors {(4, 8) ... (16, 16)}

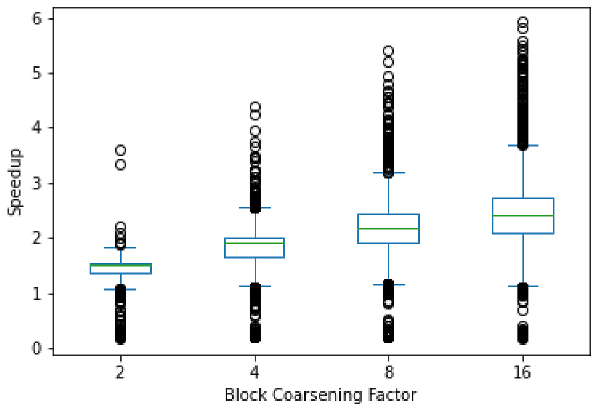
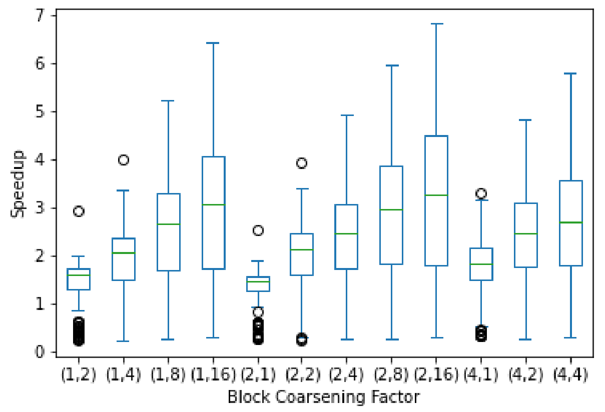
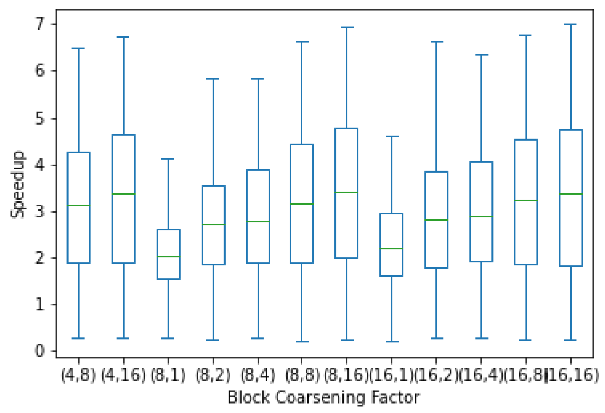


Figure 0.4.4: **One-dimensional kernels:** Distribution of speedup for block coarsening factors {2, 4, 8, 16}



(a)



(b)

Figure 0.4.5: **Two-dimensional kernels:** (a) Distribution of speedup for block coarsening factors {(1, 2) ... (4, 4)} (b) Distribution of speedup for block coarsening factors {(4, 8) ... (16, 16)}

Ακολουθώντας τα προηγούμενα αποτελέσματα, εκτελούμε τη μεθοδολογία μηχανικής μάθησης στα προβλήματα προβλέψεων που περιγράφονται παραπάνω. Δεν θα εξετάσουμε κανέναν δισδιάστατο πυρήνα ούτε το πρόβλημα δυαδικής ταξινόμησης για block coarsening, λόγω των εξαιρετικά ανισόρροπων κλάσεων που προέκυψαν από τα προηγούμενα αποτελέσματα.

1. **Binary classification of 1-dimensional thread coarsened kernels**

   Για αυτό το πρόβλημα, οι κλάσεις μας 0 και 1 αποτελούνται από βέλτιστους πυρήνες χωρίς χονδροποίηση (tcf: 1) και βέλτιστους πυρήνες με χονδροποίηση (tcf: 2, 4, 8, 16). We randomly choose 153 kernels for each class.

   Επειδή το σύνολο δεδομένων είναι αρκετά μικρό και η αρχιτεκτονική του νευρωνικού μας δικτύου είναι βαθιά, η διαδικασία εκπαίδευσης είναι επιρρεπής σε overfitting. Για να ξεπεράσουμε αυτό το πρόβλημα, πραγματοποιήσαμε αναζήτηση πλέγματος στις υπερπαραμέτρους του μοντέλου και επιλέξαμε να μειώσουμε τον ρυθμό εκμάθησης του μοντέλου στο 0,0001, να αυξήσουμε τις εποχές εκπαίδευσης σε 250, να αυξήσουμε το μέγεθος batch σε 256 και να προσθέσουμε ένα αυστηρό dropout επίπεδο μετά το τελευταίο dense επίπεδο. Το μοντέλο επιτυγχάνει **84%** ακρίβεια και μέση επιτάχυνση $1,7$.

2. **Five-class classification of 1-dimensional thread coarsened kernels**

   Για αυτό το πρόβλημα, οι κλάσεις μας 0, 1, 2, 3 και 4 αντιστοιχούν στον βέλτιστο συντελεστή coarsening κάθε πυρήνα, tcf: 1, 2, 4, 8 και 16 αντίστοιχα. Επιλέγουμε να εξισορροπήσουμε το σύνολο δεδομένων επιλέγοντας τυχαία 153 πυρήνες, το πολύ, από κάθε κλάση. Το μοντέλο επιτυγχάνει 42% ακρίβεια.

   Για να ξεπεράσουμε το πρόβλημα της υπερπροσαρμογής χρησιμοποιήσαμε την προτεινόμενη μεθοδολογία αύξησης συνθετικών δεδομένων, που περιγράψαμε παραπάνω, η οποία αποτελείται από δύο στρατηγικές. Συνδυάζοντας αυτές τις δύο στρατηγικές δημιουργούμε 5000 συνθετικούς πυρήνες ανά κλάση, αλλά ακόμη και με την αύξηση δεδομένων το πρόβλημα της υπερπροσαρμογής παραμένει και η ακρίβεια στο validation σύνολο δεν βελτιώνεται. Ως εκ τούτου, είναι ασφαλές να υποθέσουμε ότι το πρόβλημα οφείλεται στη χαμηλή ποιότητα των δεδομένων.

   Είναι ενδιαφέρον να ρίξουμε μια πιο προσεκτική ματιά στους πυρήνες που δεν έχουν ταξινομηθεί σωστά από το μοντέλο. Ο πίνακας σύγχυσης για τις προβλεπόμενες κλάσεις πυρήνων που υπάρχουν στο σύνολο δοκιμής φαίνεται στο σχήμα 0.4.6. Μπορούμε να δούμε ότι για τους πυρήνες που το μοντέλο αποτυγχάνει να ταξινομήσει σωστά, η προβλεπόμενη κλάση τους είναι δίπλα στη βέλτιστη. Τέλος, το σχήμα 0.4.7 δείχνει ένα πλαίσιο που απεικονίζει την επιτάχυνση αυτών των πυρήνων. Παρατηρούμε ότι ακόμα κι αν το μοντέλο αποτύχει να ταξινομήσει σωστά αυτούς τους πυρήνες, οι προβλεπόμενες κλάσεις επιτυγχάνουν μέση αύξηση στην ταχύτητα 1.5.

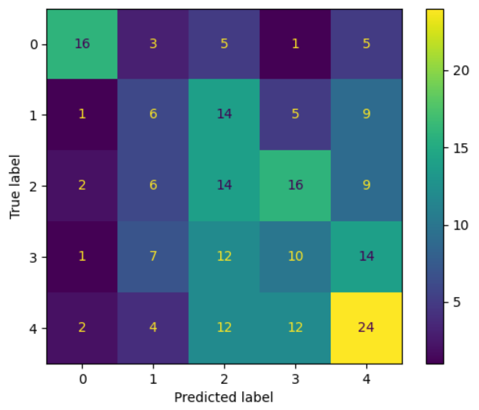


Figure 0.4.6: Confusion matrix of the classified kernels

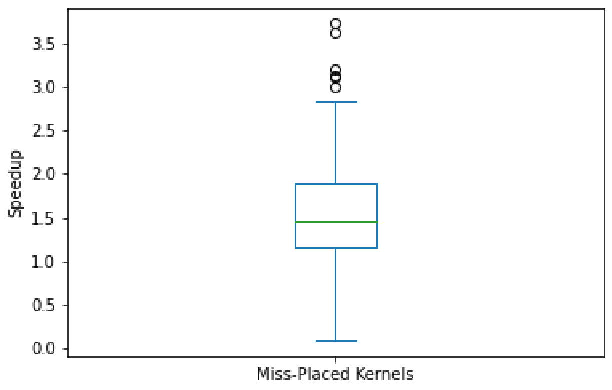


Figure 0.4.7: Speedup of miss-classified kernels as a boxplot

3. **Five-class classification of 1-dimensional block coarsened kernels**

   Για αυτό το πρόβλημα, οι κλάσεις μας είναι ίδιες με πριν και αντιστοιχούν στον βέλτιστο συντελεστή block coarsening κάθε πυρήνα, bcf: 1, 2, 4, 8 και 16. Λόγω της μεγάλης ανισορροπίας του συνόλου δεδομένων επιλέγουμε να το εξισορροπήσουμε επιλέγοντας τυχαία 100 πυρήνες από την τελευταία κλάση που αναφέρεται στο bcf 16, και για τις υπόλοιπες κλάσεις κρατάμε όλους τους πυρήνες. Το μοντέλο επιτυγχάνει ακρίβεια 33%. Αν και το ποσοστό ακρίβειας είναι χαμηλό, η μέση επιτάχυνση των ταξινομημένων πυρήνων είναι 1.4.

0.5 Συμπεράσματα και Μελλοντικές Προεκτάσεις

Σε αυτή τη διπλωματική εργασία, μελετάμε τρόπους ενσωμάτωσης δεδομένων κώδικα προγραμματισμού στην Επεξεργασία Φυσικής Γλώσσας (NLP). Στόχος μας είναι να μάθουμε πώς να χρησιμοποιούμε τη μηχανική εκμάθηση για να προβλέψουμε τα καλύτερα ευρετικά βελτιστοποίησης για παράλληλες εφαρμογές GPU με βάση τα ποσοτικά και τα ποιοτικά χαρακτηριστικά τους. Προτείνουμε αρχιτεκτονικές μηχανικής μάθησης που βελτιώνουν το state-of-the-art στο OpenCL dataset. Επίσης, προτείνουμε μια end-to-end μεθολογία για CUDA kernels, που μετασχηματίζει, εκτελεί, μετρά την απόδοσή τους και, με τη χρήση machine learning, προβλέπει τον βέλτιστο παράγοντα μετασχηματισμού, για μεγιστοποίηση της απόδοσης. Την αξιολογούμε πάνω στο νεοσύστατο LS-CAT dataset [2], ανακαλύπτοντας τα τρωτά του σημεία και βγάζουμε χρήσιμα αποτελέσματα.

Στο μέλλον δύναται να προταθούν διάφορες επεκτάσεις και παραλλαγές της δουλειάς μας. Κατά τη διερεύνηση των δυνατοτήτων των πυρήνων CUDA για Επεξεργασία Φυσικής Γλώσσας, ανακαλύψαμε αρκετούς ενδιαφέροντες μελλοντικούς δρόμους που μπορεί να κρύβουν ένα αναξιοποίητο δυναμικό. Αυτοί συνοψίζονται παρακάτω:

- Εξερεύνηση και αξιολόγηση transformer μοντέλων, τα οποία μπορούν να προεκπαιδευτούν σε μεγάλα unlabeled συνολα δεδομένων C/C++ και να χρησιμοποιηθούν για την πρόβλεψη εργασιών κατάντη.
- Πρόσθετη αξιολόγηση της ποιότητας των CUDA πυρήνων του συνόλου δεδομένων LS-CAT.
- Εκπαίδευση σε πρόβλημα παλινδρόμησης για πρόβλεψη χρόνου εκτέλεσης
- Χρήση Transfer Learning
- Εξερεύνηση και αξιολόγηση διαφορετικών τεχνικών tokenization σε πυρήνες CUDA σχετικά με την προγνωστική ικανότητα του μοντέλου.
- Επέκταση της μεθοδολογίας μας για να βελτιστοποιήσουμε διαφορετικές μετρήσεις απόδοσης, εκτός του χρόνου εκτέλεσης, όπως την κατανάλωση ενέργειας.

- Αξιολόγηση του αντίκτυπου στην πρόβλεψη κάθε χαρακτηριστικού κειμένου.
- Δημιουργία ένός σύνολο δεδομένων μεγάλης κλίμακας από πυρήνες CUDA υψηλής ποιότητας.

Chapter 1

Introduction

The massive amount of data generated and shared by enterprises, governments, different industrial and non-profit sectors, and scientific research has resulted in an unprecedented increase in the size and volume of data-intensive jobs. [10] To meet the demands of the migration to big data, both hardware and software must undergo major changes. [11]

Compilers are crucial for software efficiency because they convert input code to a set of machine instructions that take advantage of the capabilities of the target architecture. Modern compilers are simply too complex for a single developer to comprehend completely, and optimizing a single component requires knowledge of the entire compiler and how its components interact. Due to the high development costs of GPUs, FPGAs, and ASICs [12, 13], compilers are unable to adapt to the rapidly changing landscape of material. This failure to adjust results in increased energy usage and performance degradation.

Machine learning is a field of artificial intelligence (AI) and computer science that focuses on the use of data and algorithms to mimic how humans learn, gradually increasing its accuracy. There are numerous tasks in programming languages where machine learning might be beneficial.

A frequent example of how machine learning aids in the development of better programs is the time-consuming process of heuristic optimization creation. For instance, assume a developer is entrusted with modifying a compiler's loop unrolling heuristic, which is a technique for optimizing a program's execution speed at the expense of its binary size, a technique known as space–time trade-off. [14] Numerous factors, including the size of the loop body, the number of loop iterations, and the number of registers necessary to execute the loop body, might impact the decision to unroll the loop. [15] The problem becomes more complicated when the heuristic selection must take into account not only the software, but also the target hardware, such as the instruction cache.

Rather than amassing all of this knowledge, a developer may choose the process of machine learning. The concept is to utilize a learning algorithm to extract a heuristic from empirical data on loop performance in various configurations (e.g. loop unrolling factors). This can be accomplished by repeatedly analyzing benchmark programs with various unrolling factors in order to discover which is the best in each scenario (labeling). A machine learning system then models the association between these representations and the unrolling decisions that must be made by converting each program code to numerical data. This enables reliable predictions to be generated for unseen data that has a similar structure to the training data; all without the need for expert domain knowledge.

In this Diploma Thesis, we investigate approaches for incorporating programming code data into natural language processing (NLP). Our goal is to discover techniques to exploit significant quantifiable properties, or features, of sample programs in order to anticipate the optimum optimization heuristics for parallel applications that will maximize their performance through the process of machine learning.

1.1 Contributions

The contribution of this thesis is twofold. First, we examine state-of-the-art deep learning approaches for natural language processing of OpenCL kernels. Specifically:

- We develop and evaluate six different deep neural network architectures, with the goal of optimizing the execution time of each kernel by mapping it to the best hardware, CPU or GPU.
- By leveraging previous scientific efforts and open datasets in the field, we show that our proposed DNN architectures outperform state-of-the-art NLP approaches by providing up to 4% higher prediction accuracy.

Second, we examine the applicability of the overall best DNN architecture from the previous step, for natural language processing of CUDA kernels. We propose an end-to-end optimization methodology that encompasses the flow from human-written code to kernel runtime optimization without the usage of a compiler. To the best of our knowledge, this is the first work that attempts to apply NLP techniques for the CUDA programming model. In detail:

- We develop and deliver a source-to-source compiler for automatic thread and block coarsening transformations of CUDA kernels.
- We provide a source rewriter for CUDA kernels, that removes semantically irrelevant information and rewrites the variable and function names by parsing the abstract syntax tree (AST).
- We propose a tokenization methodology for CUDA kernels, that creates their respective vocabulary and integer mappings.
- We apply an execution pipeline for large scale profiling of CUDA kernels, measuring their performance, for a given list of coarsening factors.
- We propose a synthetic data augmentation approach of CUDA kernels.
- We perform an extensive profiling of the state-of-art LS-CAT dataset [2] for different coarsening factors on NVIDIA V100S high-end GPU, we discover its vulnerabilities and examine the applicability of machine learning for three different prediction problems.
- We perform grid search on the model hyperparameters to tackle overfitting.
- We show that our proposed methodology can achieve 84% accuracy for the binary problem of thread coarsening, with an average speedup of 2.1, 41% and 31% accuracy for the 5 classification problem on thread and block coarsening, respectively, with average speedups of 3.1 and 1.7 respectively.

1.2 Thesis Structure

The thesis is divided into 7 chapters. The following is an overview of the contents of each chapter:

Chapter 2 contains a detailed survey of the research on deep learning models and their application to source code modeling and production. We investigate how machine learning techniques may be utilized to build appropriate optimizations automatically on GPU systems and on program comprehension tasks in general.

In Chapter 3 we discuss the underlying technologies that support this research in detail, including graphics processing units (GPUs), their architecture and programming model, neural networks (NNs), their core architectures, and methods to natural language processing.

Our main research work is included in Chapters 4 and 5. Specifically, Chapter 4 includes the motivation behind our approach on NLP Techniques of device mapping of OpenCL kernels. We compare to the state-of-the-art approach and we evaluate our proposed methodologies.

Chapter 5 outlines our proposed methodology and the preprocessing conducted in order to develop an automated pipeline for transforming and profiling CUDA kernels, given a list of thread/block coarsening factors, and using machine learning to predict their optimal coarsening factor that minimizes their execution time. We discuss the data we used, to evaluate our methodology, and their vulnerabilities, and our efforts to overcome these problems.

Finally, Chapter 6 summarizes the thesis's findings and presents our future directions.

Chapter 2

Related Work

This chapter covers the state-of-the-art deep learning models and their extension to source code modeling and creation in detail.

Magni et al. demonstrate how machine learning approaches may be used to automatically develop suitable thread coarsening strategies for a variety of GPU systems [17]. Their approach takes six coarsening parameters into account (1, 2, 4, 8, 16, 32). The objective is to construct a machine-learning model that can determine whether an OpenCL kernel should be coarsened on a certain GPU architecture and, if so, what coarsening factor should be used. Among the numerous machine learning algorithms available, they selected to model the problem using an artificial neural network. Constructing such a model follows the standard three-step supervised learning procedure, which is illustrated in Figure 1 and described in further detail below.

(a) *Feature Extraction:* Magni et al. leverage static code properties taken from the compiler's intermediate representation to describe the input OpenCL kernel. They specifically built a compiler-based technique for extracting feature values from the LLVM bitcode of a program [16]. They began with 17 potential characteristics. These include the quantity and kind of instructions, as well as the amount of memory level parallelism (MLP) in an OpenCL kernel. Table 2.1 summarizes the candidate features mentioned in [3]. Candidate features are typically picked based on the developers' intuitions, ideas from previous works, or a combination of the two. Following the selection of candidate features, a statistical technique called principal component analysis (PCA) is used to map the 17 candidate features into seven aggregated features, each of which is a linear combination of the original features. This technique is referred to as "feature dimension reduction". Dimension reduction enables the learning algorithm to work more effectively by removing duplicated information among candidate characteristics.

(b) *Training the Model:* To create training data for the work given in [3], 16 OpenCL benchmarks were used. To determine which of the six coarsening factors performs the best for a given OpenCL kernel on a certain GPU architecture, we can apply each factor to an OpenCL kernel and time its execution. Due to the fact that the ideal thread coarsening factor differs between hardware architectures, this procedure must be repeated for each target architecture. Magni et al. retrieved the aggregated feature values for each kernel in addition to determining the optimal coarsening factor. When these two procedures are applied to the training benchmarks, a training data set is created in which each training example contains the optimal coarsening factor and feature values for a training kernel. After that, the training examples are fed into a learning algorithm that attempts to identify a set of model parameters (or weights) that minimizes the total prediction error on the training instances. The learning process produces an artificial neural network model, the weights of which are calculated using the training data.

(c) *Inference:* Following training, the learnt model can be used to forecast the best coarsening

Feature Name	Feature Description
BasicBlocks	basic blocks
Brances	branches
DivInsts	divergent instructions
DivRegionInsts	instructions in divergent regions
DivRegionInstsRatio	instr. in divergent regions / total instructions
DivRegions	divergent regions
TotInsts	instructions
FPInsts	floating point instructions
ILP	average ILP / basic block
Int/FP Inst Ratio	branches
IntInsts	integer instructions
MathFunctions	match builtin functions
MLP	average MLP / basic block
Loads	loads
Stores	stores
UniformLoads	loads that are not affected by coarsening
Barriers	barriers

Table 2.1: Candidate Code Features Used in [3]

factor for previously encountered OpenCL applications. To do this, static source code features from the target OpenCL kernel are retrieved; the extracted feature values are then fed into the model, which determines whether or not to coarsen and what coarsening factor to employ. The method described in [3] improves performance by an average of 1.11x to 1.33x across four GPU architectures without compromising performance on any one benchmark.

Grewe et al. [4] propose a predictive approach for mapping OpenCL kernels to the best device in heterogeneous CPU/GPU systems. They generate decision trees using supervised learning and a combination of static and dynamic kernel features. The static features of the application are extracted using a customized LLVM pass, while the dynamic features are extracted from the OpenCL runtime. Table 2.2a summarizes the characteristics of their work. Each feature is a mathematical expression constructed from the code and runtime measurements listed in Table 2.2b.

Cummins et al. [1] pioneered the use of neural networks to extract information from program source code for compiler optimization. Their technique, namely DeepTune, abstracts and picks appropriate information from the raw source code automatically. Unlike previous work, which incorporates a collection of human-crafted features into the predictive model, this work incorporates program code directly into the training data. They translated input programs into source token streams and then used a deep neural network to predict the optimal heuristic value for specific OpenCL optimizations directly from that. They drew on a well established architecture known as long short term memory networks (LSTM), which had achieved considerable success in natural language processing. These networks are capable of processing streams of inputs and remembering events that occurred in the distant past. This enables them to comprehend the program's structure, such as if a variable was declared previously. The results surpassed prior work using hand-built features. Additionally, their work demonstrates that the features of the raw code abstracted by the neural network's top layers are mostly independent of the optimization problem. While this is encouraging, it is worth noting that dynamic information such as the program's input size and performance counter values is frequently required for describing the target program's behavior. As a result, DeepTune does totally eliminate human involvement in feature engineering.

Ben-Nun et al. [5] took it a step further. They recognized that, while a token-based representation works well for ambiguous natural language, a graph-based representation works better for programming languages. They represent a program's instructions as edges in a graph that depicts the relationships between variables. They then learn vector representations for each instruction based on its graph

Feature Name	Feature Description
F1: data size/(comp+mem)	communication to computation ratio
F2: coalesced/mem	Percentage of coalesced memory accesses
F3: (localmem/mem)∗wgsize	ratio of local to global memory accesses times the number of work-items
F4: comp/mem	computation to memory ratio

(a) Features values

Feature Name	Type	Feature Description
comp	static	number of compute operations
mem	static	number of accesses to global memory
localmem	static	number of accesses in local memory
coalesced	static	number of merged memory accesses
data size	dynamic	size of data transfers
wgsize (work group size)	dynamic	number of work-items per kernel

(b) Features values used in feature computation

Table 2.2: Features used by Grewe et al. to predict heterogeneous device mappings for OpenCL kernels. [4]

context. After that, LSTMs can process a program as a sequence of these vectors. Even though this is the most sophisticated representation, it omits variables, operand types, and operand order. As a result, it is incapable of duplicating the commonly used dataflow analysis in any modern compiler.

Brauckmann et al. broaden this concept by determining not only the vectors used to represent instructions, but also how they are processed by the learner. They employ message-passing neural networks with a state for each node. This state is sent along edges to each neighbour, who uses a learned function to merge it into its own state. After a few rounds of message passing, the heuristic value is determined by a learned aggregation function. [6]

In 2018 Jacob Devlin et al. introduced a novel language representation model called BERT, which stands for Bidirectional Encoder Representations from Transformers. [17] [18] In contrast to prior language representation models, BERT is designed to pre-train deep bidirectional representations from unlabeled text by conditioning all layers on both left and right context simultaneously. As a result, with only one additional output layer, the pre-trained BERT model may be fine-tuned to generate state-of-the-art models for a wide variety of tasks, such as text classification and language inference, without requiring significant task-specific architecture modifications.

Kanade et al. made the first attempt to obtain a high-quality contextual embedding of source code and to evaluate it on a variety of program-understanding tasks. [19] To be specific, they curate a massive, deduplicated corpus of 7.4 million Python files from GitHub, which they use to pre-train CuBERT, an open-sourced BERT model for code understanding; and, second, they create an open-sourced benchmark consisting of five classification tasks and one program-repair task, which are similar to previously proposed code-understanding tasks in the literature. They optimize CuBERT using benchmark tasks and compare the resulting models to various variants of Word2Vec token embeddings, BiLSTM and Transformer models, as well as published state-of-the-art models, demonstrating that CuBERT outperforms them all, even with shorter training times and fewer labeled examples.

In comparison to previous generations of machine learning, deep learners require much larger data sets than are typically seen in compiler research. While this can be mitigated in some cases by synthetic data generation [20], the rate of innovation will accelerate significantly as the availability of large labelled data sets increases.

Chapter 3

Background on GPU Acceleration and Machine Learning

This chapter describes in depth the fundamental technologies that underpin this study, including graphics processing units (GPUs), their architecture and programming model, neural networks (NNs), their fundamental architectures, and natural language processing approaches.

3.1 GPU Architecture and Programming Models

A Graphics Processor Unit (GPU) is a type of computer hardware device that is commonly used for running applications that require a lot of graphics, such as 3D modeling software or virtual desktop infrastructures. In the consumer market, a GPU is mostly utilized to accelerate gaming visuals. However, GPGPUs (General Purpose Graphics Processing Units) constitute the hardware of choice for accelerating computational workloads in today's High Performance Computing (HPC) environments.

3.1.1 GPU Architecture

A high-level architecture overview of a GPU appears to indicate that the essence of a GPU is all about putting available cores to work and that it is less concerned with low latency cache memory access, in contrast to a CPU. A single GPU device is composed of numerous Processor Clusters (PCs) each containing a number of Streaming Multiprocessors (SM). Each SM contains a layer-1 instruction cache layer and the cores related with it. Typically, one SM utilizes a specialized layer-1 cache and a shared layer-2 cache before accessing global GDDR-5 (or GDDR-6 memory in recent GPU models) memory. Its architecture is shown in figure 3.1.1.

In comparison to a CPU, a GPU operates with fewer and smaller memory cache levels. The reason behind this is that a GPU has more transistors dedicated to computing, which means it is less concerned with the time required to get data from memory. The potential memory access 'delay' is concealed as long as the GPU is kept occupied with computations.

3.1.2 Programming Models

GPGPU programming requires partitioning many processes or a single process across multiple processors in order to reduce the time required to complete. GPGPUs leverage software frameworks such as OpenCL and CUDA to accelerate specific processes within a program, thereby speeding up and simplifying your work. GPUs enable parallel computing by utilizing hundreds of on-chip processing cores that communicate and cooperate in real time to tackle complicated computing tasks.

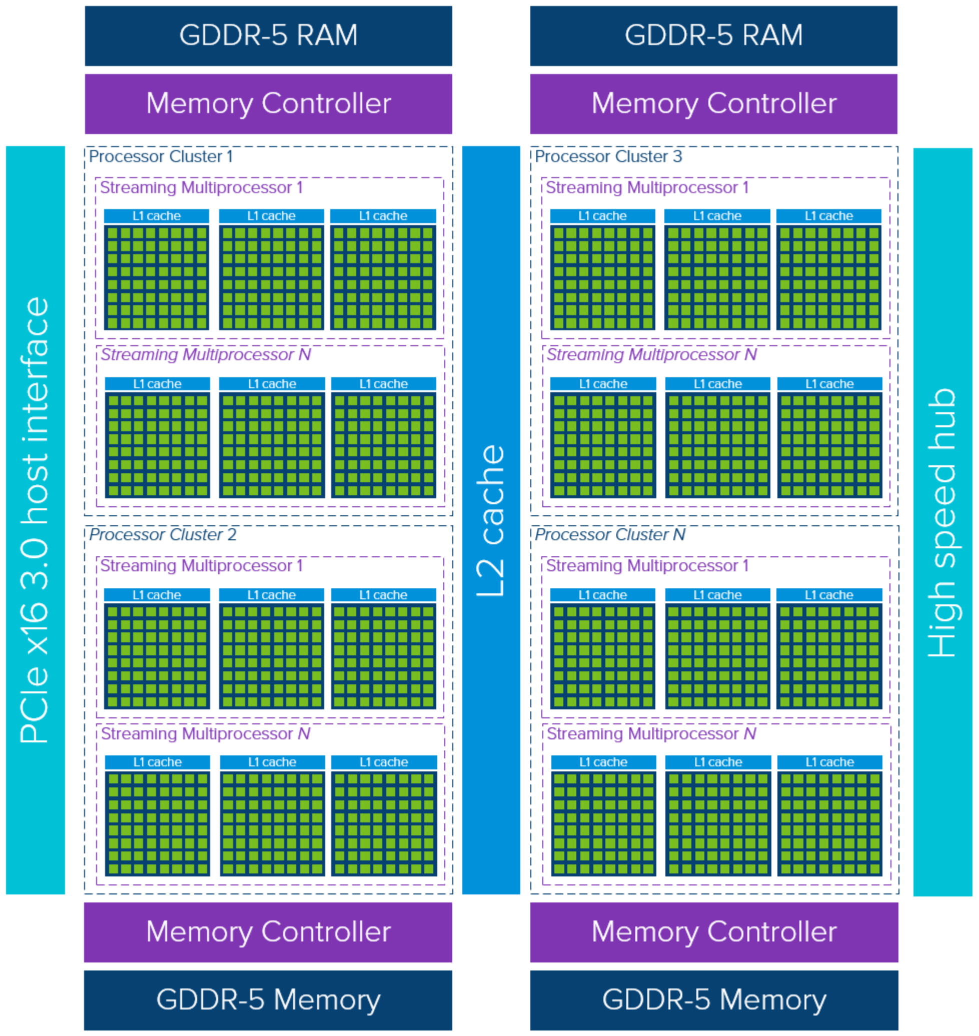


Figure 3.1.1: GPU Architecture [21]

Cuda

CUDA stands for Compute Unified Device Architecture. It is a parallel computing platform and programming model developed by Nvidia specifically for use with Nvidia GPUs. It simplifies and elegantly implements the use of a GPU for general purpose computing. CUDA C++ is just one of the many ways you can use CUDA to develop massively parallel applications. It enables you to develop high-performance algorithms using the powerful C++ programming language and thousands of parallel threads running on GPUs.

In CUDA programming model the parallel program executed on GPU called kernel, is organised in numerous threads, all organized in thread blocks. Additionally, from the hardware perspective, GPUs' SMs (Streaming Multiprocessors) are capable to simultaneously launch and execute blocks. When one of its thread blocks completes the execution, it proceeds to the serially following thread block. Thread blocks are made up of 'warps'. A warp [22] is a collection of 32 threads contained within a thread block that all execute the same instruction in the same cycle (Instruction-Level Parallelism, ILP).

Each thread in CUDA is assigned a unique index, which enables it to calculate and access memory locations within an array. Figure 3.1.2 shows an example of indexing a 1-dimensional block. If our block is 2-dimensional then the y-index is computed in the same manner as the x-index.

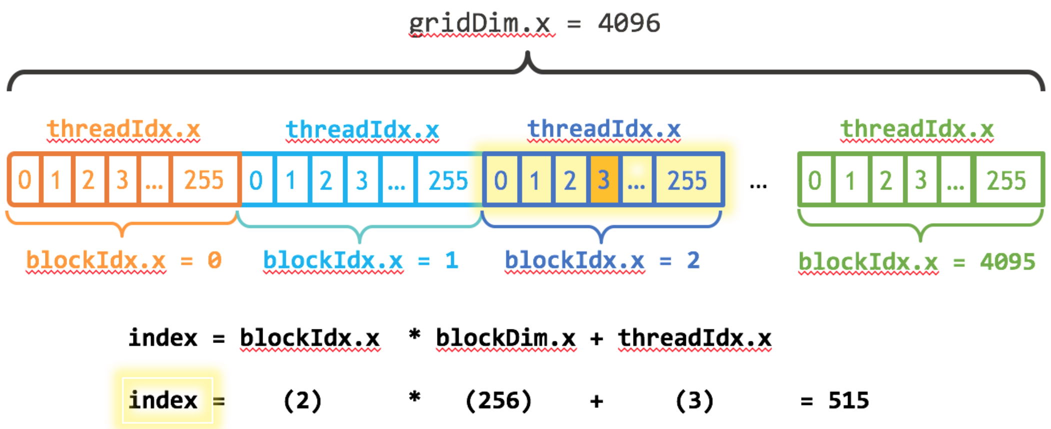


Figure 3.1.2: Example of indexing a 1-dimensional block. [23]

Listing 3.1 shows a simple kernel written in CUDA C++ that performs matrix-vector multiplication.

```
__global__ void mm_simple(float* A, float* B, int n, float* C)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int k = 0; k < n; k++) {
        sum += A[row*n+k] * B[k * n + col];
    }
    C[row*n+col] = sum;
}
```

Listing 3.1: Matrix-Vector Multiplication Algorithm in CUDA C++

OpenCL

OpenCL, which stands for Open Computing Language, was created to give a benchmark for heterogeneous computing that was not limited to NVIDIA GPUs. OpenCL is a portable language for programming GPUs that supports CPUs, GPUs, Digital signal processors, and other types of processors.

OpenCL C is a dialect of the C99 language that has been altered to meet the OpenCL device model. OpenCL C is expanded to support parallelism via vector types and operations, synchronization, and work-item and work-group-related functions. Rather than having a main function for a device program, OpenCL C functions are marked __kernel to indicate that they are entry points into the program that can be called from the host application. There are no function pointers, bit fields, or variable-length arrays, and recursion is prohibited. [24]

OpenCL C is extended to facilitate use of parallelism with vector types and operations, synchronization, and functions to work with work-items and work-groups. It is, also, expanded to support parallelism via vector types and operations, synchronization, and work-item and work-group-related functions. OpenCL, in particular, supports fixed-length vector types such as float4 (4-vector of single-precision floats); these vector types are available in lengths of two, three, four, eight, and sixteen for a variety of base types. Vectorized operations on these types are designed to map to SIMD instruction sets when OpenCL programs are executed on CPUs. [25]

Listing 3.2 shows a kernel written in OpenCL C that performs matrix-vector multiplication.

```
__kernel void matvec(__global const float *A, __global const float *x,
                     uint ncols, __global float *y)
{
    size_t i = get_global_id(0);              // Global id, used as the row index
    __global float const *a = &A[i*ncols];    // Pointer to the i'th row
    float sum = 0.f;                          // Accumulator for dot product
    for (size_t j = 0; j < ncols; j++) {
        sum += a[j] * x[j];
    }
    y[i] = sum;
}
```

Listing 3.2: Matrix-Vector Multiplication Algorithm in OpenCL C

3.1.3 Kernel Optimizations

The key to performance on GPGPU systems is huge multithreading, which makes use of the system's many cores and hides global memory latency. However, developers must strike a balance between the resource consumption of each thread and the number of concurrently active threads. The resources that must be managed include the number of registers and on-chip memory required by each thread, the number of threads per multiprocessor, and the global memory bandwidth. The next sections discuss two general effective software optimization strategies: thread coarsening and block coarsening.

Listing 3.3 demonstrates a simple CUDA kernel in which every thread performs operations on a single item at a time.

Thread Coarsening

Thread coarsening is a code transformation technique that combines two or more adjacent threads in the same block into a single thread. It has been demonstrated to be efficient for a variety of parallel programs and architectures. In that instance, it decreases the number of threads per block while maintaining the total number of blocks launched. It is capable of increasing the amount of work performed by each kernel by reproducing the instructions in its body, while simultaneously decreasing the number of threads instantiated at runtime. It is analogous to loop unrolling except that it is applied across parallel work items rather than serial loop iterations.

However, thread coarsening may have a number of undesirable side effects, including a reduction in total parallelism and an increase in register pressure, both of which can result in performance degradation. Determining when and how to apply coarsening is not easy, as the optimal coarsening factor varies according on the target program, its input size and the hardware architecture on which the program executes [17], [19].

Listing 3.4 depicts the squared kernel after the thread coarsening transformation has been applied to it.

Block Coarsening

In contrast to thread optimization, block coarsening is a kernel transformation technique that combines the workloads of two or more thread blocks and therefore reduces their total number while maintaining the thread count per block. As a result, it merges numerous surrounding blocks to address the issues inherent with extensive fine-grained parallelism. Similarly to thread coarsening may have the same undesirable side effects.

Listing 3.5 shows the squared kernel after the block coarsening transformation has been applied to it.

```
__global__ void square(float* in, float* out){
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    out[gid] = in[gid] * in[gid];
}
```

Listing 3.3: Original Cuda Kernel

```
__global__ void square(float* in, float* out, int tc){
    int gid = blockIdx.x * (blockDim.x * tc_x) + (threadIdx.x * tc_x);
    for (int index_x = 0; index_x < tc_x; index_x++)
        out[gid + index_x] = in[gid + index_x] * in[gid + index_x];
}
```

Listing 3.4: Thread Coarsened Cuda Kernel

```
__global__ void square(float* in, float* out, int bc){
    int gid = (blockIdx.x * bc_x) * blockDim.x + (threadIdx.x * bc_x);
    for (int index_x = 0; index_x < bc_x; index_x++)
        out[gid + index_x] = in[gid + index_x] * in[gid + index_x];
}
```

Listing 3.5: Block Coarsened Cuda Kernel

3.2 Machine Learning - Artificial Neural Networks

The term Artificial Intelligence, and consequently the scientific area which it covers, first appeared in 1943, from the pioneering work of McCulloch and Pitts [26]. The first was a psychiatrist and the second mathematician and their classical work developed within the University of Chicago for more than 5 years. Since then many similar works were developed, several of which form the basis of today neural networks, such as that of J. Hopfield [27], T. Kohonen on self organizing maps [28] etc. However, in recent years this field of science has shown great growth and has offered great achievements that helped us understand, not just how the world around us works, but also how human intelligence works. Rapid technological development in the field of information technology and especially in computer engineering has contributed to this progress, where with new technologies they have increased the processing power, resulting in the processing of a much larger volume of data in less time.

Another factor that has contributed to the evolution of Artificial Intelligence is the availability of large volumes of data, as in 2020 each of us generated more than about 1.7MB of data every second [29], which scientists in this field can now use in various ways.

In this chapter we will analyze some basic definitions and techniques of the artificial intelligence. Specifically, machine learning and deep learning will be analyzed, their techniques will be mentioned and the difference between them will be noted. Finally, the basic principles of natural language processing will be analyzed.

The model of an artificial neuron

In artificial neural networks there is a set of artificial neurons, which are their structural unit, and are connected to each other through synapses and interact. The degree to which neurons interact with each other is different and is determined by their synaptic weights. These weights are not fixed or predetermined. On the contrary, they change as the neural network receives stimuli from the environment and learns, so some bonds are strengthened and some are weakened. In other words, synaptic weights are those in which all empirical knowledge is encoded and consequently give the network the ability to learn and adapt to the environment. The inputs to each neuron can be either the primary information or the output signal from another neuron.

Figure 3.2.1 shows the model of a neuron, the key elements of which are:

1. A set of synapses or connecting links, each of which is characterized by a weight. More specifically, a signal x_j at the input of each artificial synapse j connected to the artificial neuron k, is multiplied by the synaptic weight w_{kj}
2. An adder for the addition of input signals, multiplied by the weight of the corresponding connection
3. An externally applied threshold b_k that has the effect of reducing the input to the applied activation function, known as bias
4. An activation function which is linear or non-linear

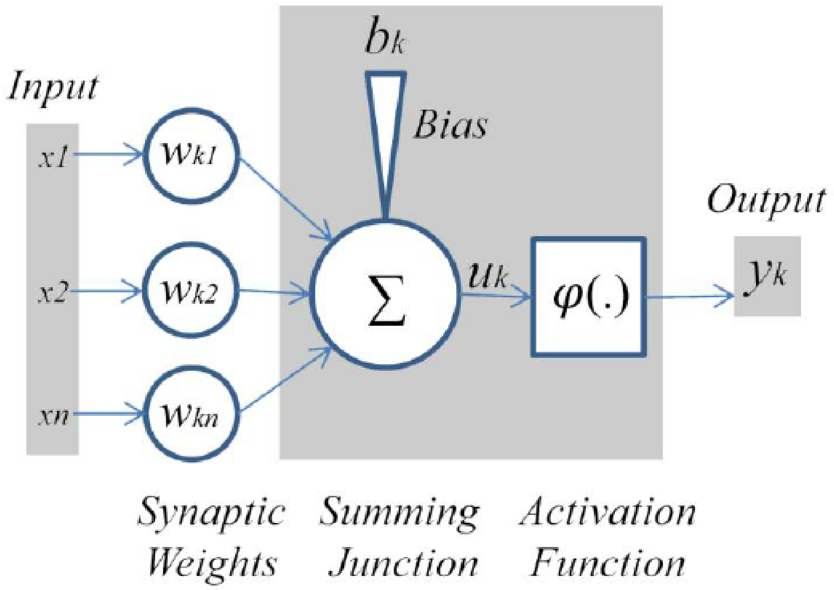


Figure 3.2.1: The non-linear model of an artificial neuron [30]

In mathematical terms a neuron j is described as follows:

$$u_k = \sum_{i=1}^{n} w_{ki} x_i \tag{3.2.1}$$

$$y_k = \phi(u_k - b_k) \tag{3.2.2}$$

where u_k is the sum of the combination of weigths and inputs of the neuron j, the b_k is the bias, the $\phi(.)$ is the activation function and the y_k is the output of the network.
The output of the network can also be written as follows:

$$u_k = W * x + b_k \tag{3.2.3}$$

The most common activation functions are:

1. Step Function
   We have

$$\phi(u) = \begin{cases} 1, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{3.2.4}$$

   So, the output of a neuron k, with the above activation function, is of the form:

$$y_k = \begin{cases} 1, & \text{if } u_k \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{3.2.5}$$

where u_k is the inner activation layer of the neuron.
The derivative of not defined and is therefore calculated by numerical analysis methods.

2. Sigmoid Function
   We have
   $$\phi(u) = \frac{1}{(1 + \exp^{-\alpha u})} \tag{3.2.6}$$
   It is one of the most common forms of activation function used in the construction of artificial neural networks. It is defined as a strictly ascending function, which exhibits smoothing and asymptotic properties. By changing the parameter α we get different functions, as shown in figure 3.2.2.
   The derivative is:
   $$\phi'(u) = \phi(u)(1 - \phi(u)) \tag{3.2.7}$$

3. Rectified Linear Unit
   We have
   $$\phi(u) = \begin{cases} u, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.2.8}$$
   The most popular activation function in the field of deep learning, overcoming the sigmoid function that was for years the first.

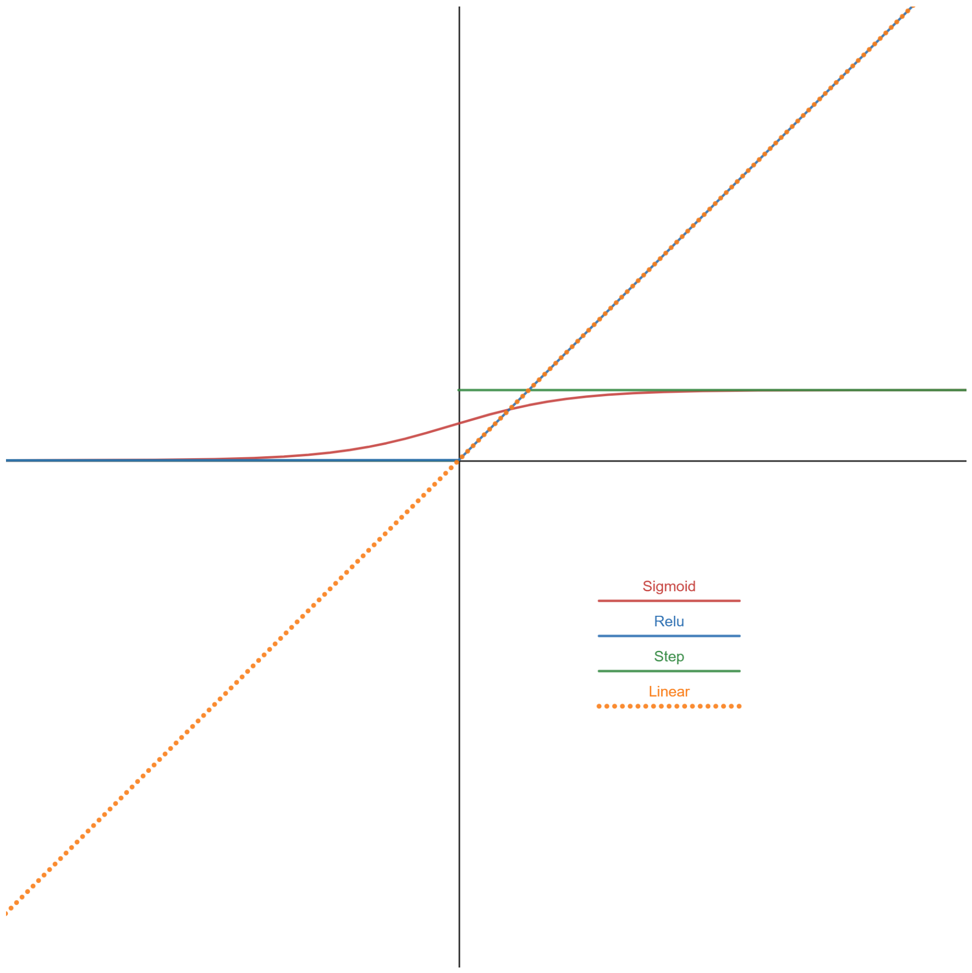


Figure 3.2.2: Activation Functions

Neural Network Architectures

There are many ways to categorize neural networks. The most common is the organization based on how the neurons are connected to each other. Basic ways are:

- The feedforward fully connected NNs
- The recurrent NNs
- The Convolutional NNs

There are, also, other ways of categorizing that usually involve a combination of the above, for example the partially connected feedforward NN [31]

3.2.1 Feedforward Fully connected NN

The feedforward fully connected neural networks, or the multilayer perceptrons (MLP) are the key models of deep learning. These models are so named because information flows through the activation function and through the intermediate calculations used to determine the y output. All the neurons of one level are connected to all the neurons of the next and there are no feedback links in which the model outputs are fed back to the input. Perceptrons are the simplest form of neural network, as they contain only the input level and the output level. Their training and optimization of synaptic weights are done with the technique of reverse error transmission, known as the back propagation algorithm.

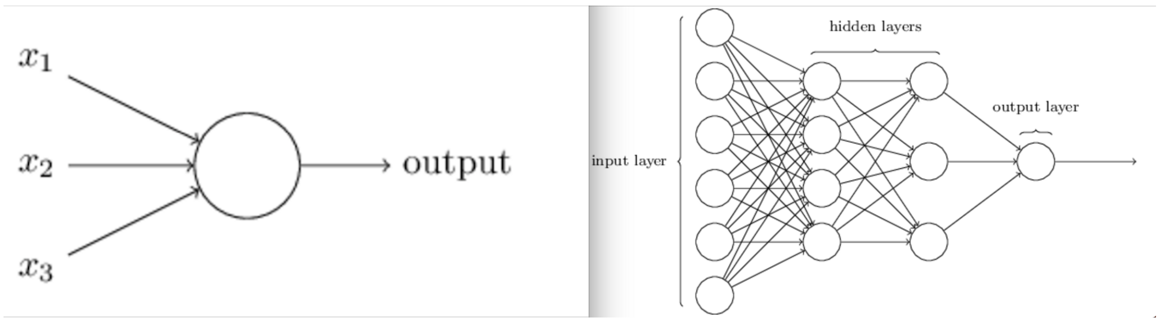


Figure 3.2.3: On the left we have a perceptron and on the right a multilayer NN [32]

Backpropagation and Gradient Descent

The most widely used technique to train artificial neural networks is backpropagation [33]. Typically, the artificial neuron parameters are initialised with small random values. During training, a mini-batch of B observations is propagated through the network in a feed-forward stage. The final outputs of the network $\hat{y}$ are then compared against the true values y and used to compute an error $L(\hat{y}, y)$. The appropriate error function depends on the task. For a classification task with K classes, categorical cross-entropy may be used:

$$L(\hat{y}, y) = -\sum_{i=1}^{K} y^i \log(\hat{y}^i) \tag{3.2.9}$$

For each layer l, the average error of the mini-batch J is backpropagated through the network to update the connection weight W^l and bias parameters b^l based on a learning rate α:

$$J = \frac{1}{B} \sum_{i=1}^{B} L^i(\hat{y}, y) \tag{3.2.10}$$

$$W^l = W^l - \alpha \frac{\partial J}{\partial W^l} \tag{3.2.11}$$

$$b^l = b^l - \alpha \frac{\partial J}{\partial b^l} \tag{3.2.12}$$

3.2.2 Convolutional NN

Convolutional Neural Networks (CNNs) are inspired by the field of Neuroscience and are front propagation networks that base their operation on the convolution process. They belong to the category of deep neural networks with the main use of image analysis and categorization. Unlike MLP networks, where each level connects to all of the next neurons and becomes vulnerable to overfitting [34], CNNs take advantage of the hierarchical pattern in the data and learn patterns of increasing complexity. They are also based on the fact that adjacent pixels are related to each other, forming patterns that can be easily identified. CNN's use the product of convergence, that is, the dot product of a matrix with sliding filters (kernels) [35]

Convolutional Neural Networks are made up of several layers. The first is an Input Layer where it contains as a parameter the dimensions of the data. Then follows a Convolutional Layer, which consists of a number of filters and hyperparameters, relating to the convolution, such as padding, stride. The total of the trainable parameters comes only from the product of the number and the dimensions of the kernels. The fact that the parameters are decreasing implies that very deep networks can be created, while at the same time problems like vanishing gradient and exploding gradient, that occur in classical MLP neural networks, are avoided. [36]

CNNs have proven to be effective in several Natural Language Processing tasks with remarkable results, such as sentence classification [37], search query retrieval [38], sentiment analysis [39] and other traditional NLP tasks. They take advantage of distributed representations of words in vector space, called embeddings. [40] [41]

3.2.3 Embedding Vectors

An embedding is a relatively low-dimensional space into which high-dimensional vectors can be translated. Embeddings make machine learning easier when dealing with huge inputs such as sparse vectors representing words. In an ideal world, an embedding captures some of the input's semantics by clustering semantically comparable inputs in the embedding space. A model's embedding can be learned and reused.

3.2.4 Recurrent NN

Recurrent neural networks (RNNs) are a type of neural networks that process sequences of data. Unlike Feedforward fully connected neural networks that assume that all training instances are independent, RNNs produce their output taking into consideration information previously presented to them. All or some of the neurons of one level are connected to those of the previous level, so that they receive information from the next levels. These networks, with their retrospective relationships, have the ability to recognize both temporal and spatial correlations in the input data, as their calculations take into account historical information.

The core idea of an RNN is to infer the temporal dynamics of the data sequence by keeping an internal state, also known as hidden layer, and updating it on each time-step, for each new datum of the sequence. As shown in 3.2.4 the RNN first take x_0 from the input sequence and extracts h_0 (hidden state). The h_0 along with the x_1 are the input for the next step and so on.

For each time point t, the equations that describe the function of an RNN are:

$$h_t = f(W_{hh}h_{t-1} + W_{hx}x_t) \tag{3.2.13}$$

$$y_t = W_s h_t \tag{3.2.14}$$

where:

- h_t: the hidden state at time t
- $x_t \in \mathbb{R}^d$: input vector at time-step t

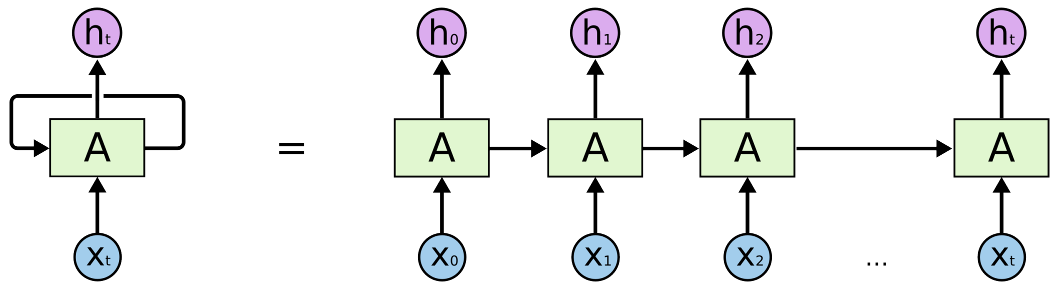


Figure 3.2.4: Recurrent Neural Network [42]

- $W_{hx} \in \mathbb{R}^{d_h \times d}$: weights matrix used to condition the input vector x_t
- $W_{hh} \in \mathbb{R}^{d_h \times d_h}$: weights matrix used to condition the output of the previous time-step h_{t-1}.
- $f()$: non-linear activation function
- $y_t \in \mathbb{R}^{l}$: the output at time-step t.
- $W_s \in \mathbb{R}^{d_h \times l}$: weights matrix used to take an output y_t

Bi-directional RNNs

As mentioned above, RNNs collect sequential information that have received at moment t and encode them in their hidden state. However, they are also likely to get more information by reading a given sequence backwards, to make more accurate predictions. [43]

The core idea, as shown in 3.2.5, is to split the state neurons of a regular RNN in a part that is responsible for the positive time direction (forward states) and a part for the negative time direction (backward states). Outputs from forward states are not connected to inputs of backward states, and vice versa. Then the hidden states of the two RNNs are combined to find the hidden state for each time point. As a result, the hidden state at time t is simply the combination of the two vectors: $h_t = \overrightarrow{h}_t || \overleftarrow{h}_{T-t}$. The same also applies for all $T+1$ time points of the input sequence.

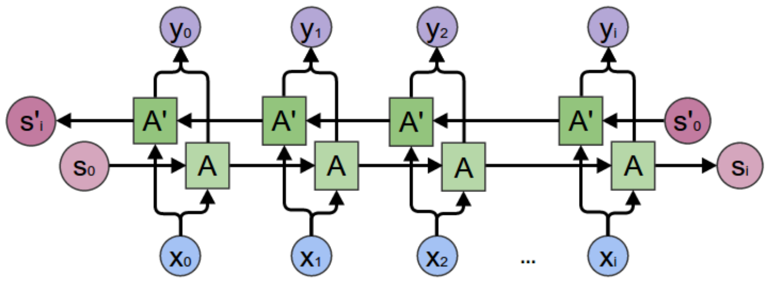


Figure 3.2.5: Bi-directional Recurrent Neural Network [44]

Problem of long-term dependencies

In the analysis of the sequential data type, a phenomenon emerged, which is called 'long-term dependence'. It occurs when two or more points in the data set are spatially spaced. Theoretically, it seems that Recurrent Neural Networks are able to handle this phenomenon and make good predictions when it happens. In practice, however, they do not seem to be able to cope with long-term dependencies. [45] [46]

This inability of Recurrent Neural Networks to provide good predictions when the dataset contains long-term dependencies is called the vanishing/exploding gradient problem. [47] [48] In order to train the network, gradients of the loss with respect to each parameter should be calculated. Computing the gradient with respect to h_0 involves many factors of W_{hh}.

If many of the intermediate values are greater than one, then the gradient value increases tremendously and causes an overflow at the time of the training. The issue is called the gradient explosion problem. To solve the problem, a simple heuristic solution is used, which clips the gradients into small numbers each time they explode. That is, whenever they reach a certain limit, they return to a small number.

On the contrary, when many values in the computation chain are between zero and one, multiplying many such small numbers together causes smaller gradients that tend to zero and makes further back time steps irrelevant to the hidden states. This is called the vanishing gradient problem, due to which we cannot distinguish if there is no dependence between two steps in the data or if the dependency is not recorded. A usual solution to the problem is the usage of ReLU as activation function. ReLU's derivative prevents the gradients from shrinking, as it is either 0 or 1. Another technique to face vanishing gradients is that, instead of initializing W_{hh} and biases randomly, setting W_{hh} equal to the identity matrix and biases to zero.

3.2.5 Long Short Term Memory Networks (LSTM)

To cope with the aforementioned problems, in 1997, Sepp Hochreiter et al. [49] proposed an architecture called Long Short-term Memory (LSTM). LSTM's allow older information to reappear at a later time and for this reason they can remember information for long periods of time. This ability to contain information other than the normal flow of a simple recurrent neural network is achieved through the use of a gate-protected cell. This cell looks like a data conveyor belt (See Figure 3.2.6). It's job is to make decisions about when and what data is going to be stored, read or deleted, through the portals that open and close.

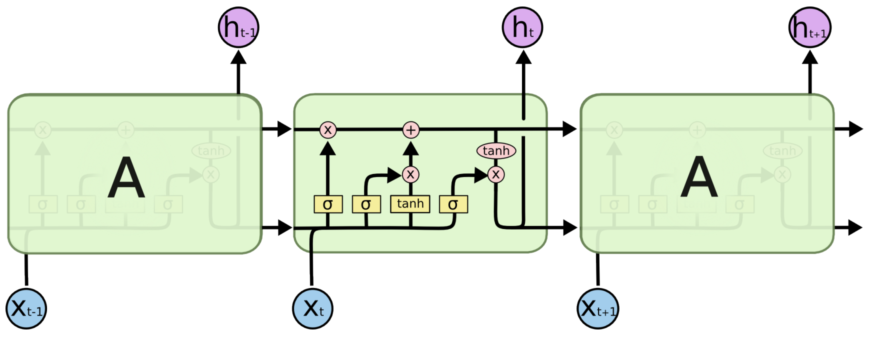
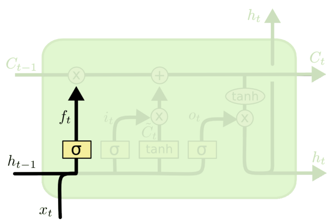


Figure 3.2.6: The repeating module in an LSTM contains four interacting layers. [42]

The analysis of an LSTM "cell" is presented below step by step:

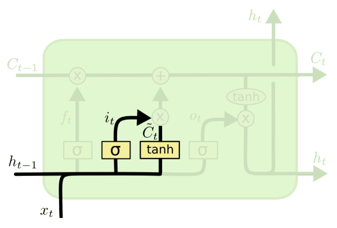
- **Forget gate**: select the junk information contained in the previous hidden state h_{t-1} and the new input x_t, and "forget" it through the portal. A number is produced between [0, 1] which is multiplied by the internal values of the state C_{t-1}, choosing what information will remain.



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

Figure 3.2.7: Forget Gate of LSTM [42]

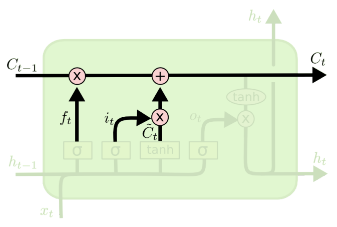
- **Update gate**: The next step is to decide what new information are going to be stored in the cell state. This has two parts. First, a sigmoid layer called the “input gate layer” decides which values will be updated and then a tanh layer creates a vector of new candidate values, C_t, that could be added to the state. Next updates the old cell state, C_{t-1}, into the new cell state C_t,



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Figure 3.2.8: Store Gate of LSTM [42]

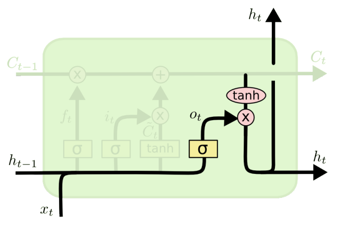
forgetting the useless information.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 3.2.9: Update Gate of LSTM [42]

- **Output gate**: Finally, output will be based on our cell state, but will be a filtered version. The state calculated from the previous steps C_t is passed through a tanh function and multiplied by the output of a sigmoid gate.



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

Figure 3.2.10: Output Gate of LSTM [42]

This LSTM network architecture makes it one of the most useful and popular types of Recurrent Neural Networks, with many applications. It is also one of the most common RNN option for sequential data problems, such as the one presented in this paper.

3.2.6 Attention Mechanisms

The core idea behind attention techniques is to mimic human behavior by focusing on the important parts of a sentence (words) for context understanding. That is, the model does not use all the vectors equally to make a prediction, but focuses on the parts of the input that contain the most relevant information about a particular problem. Originally, attention was used in Neural Machine Translation to dynamically attend over the representation of the input sequence and predict the next word in the translated sequence. Attention can be interpreted as focusing on the most relevant elements, e.g. words, of the input by computing weights of importance for their representations. Which part of the data will be considered more important than the others depends on the content of the input and is learned through gradient descent training.

The importance of each element is typically measured by their alignment with a query vector q. Given a sequence of N input vectors $h_1, \dots, h_N$ the attention module uses an alignment function to score the relevance of each h_i to the query q. The alignment scores s_i are then normalized using the softmax function to produce attention weights $\alpha_1, \dots, \alpha_N$ that sum to 1. The final representation x is the weighted average of the inputs.

$$s_i = align(q, x_i) \tag{3.2.15}$$

$$a_i = softmax(s_i) = \frac{\epsilon^{s_i}}{\sum_{t=1}^{T} \epsilon^{s_t}}, \sum_{i=1}^{T} a_i = 1 \tag{3.2.16}$$

$$x = \sum_{i=1}^{T} a_i h_i \tag{3.2.17}$$

The most important align functions are:

General Attention

$$align(q, x_i) = s^{\intercal} W_h h_i \tag{3.2.18}$$

Additive Attention

$$align(q, x_i) = \upsilon_h^{\intercal} tanh(W_h h_i) \tag{3.2.19}$$

Dot-Product Attention

$$align(q, x_i) = s^{\intercal} h_i \tag{3.2.20}$$

Scaled Dot-Product Attention

$$align(q, x_i) = \frac{s^{\intercal} W_h h_i}{\sqrt{N}} \tag{3.2.21}$$

3.2.7 Transformers

The transformer is a deep learning model that was firstly introduced by Ashish Vaswani et al. [18] and it is used primarily in Natural Language Processing (NLP) and in computer vision. It's function is based on attention, described in 3.2.6, giving weight to the effect of the different parts of the input.

Transformers are designed to handle sequential input data, but it is not necessary to process them in the same order. The attention function can be framed in any position of the input sequence. Because of this feature, transformers provide greater parallelism than similar sequential models and enabled training on larger datasets and led to the development of pretrained systems such as BERT (Bidirectional Encoder Representations from Transformers) [17] and GPT (Generative Pre-trained Transformer) [50], which have been trained with huge general language datasets, and can be fine-tuned to specific language tasks.

It is a model with encoder-decoder architecture, in which it consists of identical subsystems, that have different parameter values. All the encoders as well as the set of the decoders consists of layers with similar structure but of course have different weight values. The encoders and decoders accept inputs from the lower layers and carry them to the higher. As shown in figure 3.2.11, each encoder consists of two subsystems. The first is a Self-Attention Layer, that allows the encoder to match the dependencies of each word, with the other words in the sentence. The second subsystem is a Feed Forward Neural Network and layer normalization units. Similar is the decoders architecture with an addition of a intermediate encoder-decoder attention mechanism. This subsystem is responsible to identify and focus on specific elements of the input that has already been encoded by the encoder.

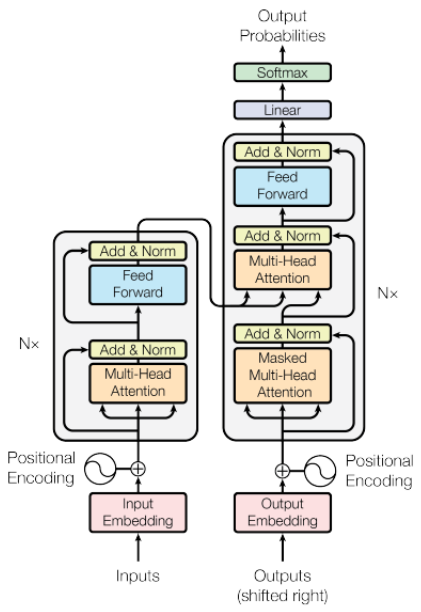


Figure 3.2.11: The Transformer architecture [18]

The system which is responsible for the correlations between the input units is the self-attention. The calculation of the outputs of the Self-Attention system is essentially done in two basic steps. The first step is to create three vectors from the input. Specifically, Query, Key and Value vectors need to be created. This vectors resulting from the multiplication of inputs with specific arrays, the parameters of which are optimized during the training process. Given the three vectors that make up the self-attention mechanism, Query (Q), Key (K) and Value (V), we can define self-attention by

$$Attention(Q, V, K) = softmax(\frac{QK^{\intercal}}{\sqrt{d_k}})V \tag{3.2.22}$$

where d_k^{-1} is a scaling factor, dependant to the dimension of K vector. One set of W_Q, W_V, W_K is called attention head. So in paper [18] multi-head attention is used, specifically with 8 heads. These heads are multiplied with another learnable vector W_O before it is fed in the feed forward subsystem.

Transformers are usually trained on a semi-supervised manner, involving unsupervised pretraining followed by supervised fine-tuning.

3.2.8 Overfitting and Regularization

When the number of feasible situations is broad, we must be cautious not to use the resulting freedom to discover meaningless normality in the data. This is referred to as overfitting. So, overfitting is the process of producing an analysis that is too near or exact to a single set of data, and thus fails to fit additional data or accurately anticipate future observations. Overfitting is a widespread occurrence that occurs even when the goal function is not random and affects all sorts of learning methods.

On the other hand, underfitting happens when a model is unable to adequately capture the structure of the data or when the model lacks the capacity to fully learn the data. When a model is underfitted, several parameters that would exist in a properly trained model are omitted, resulting in a model

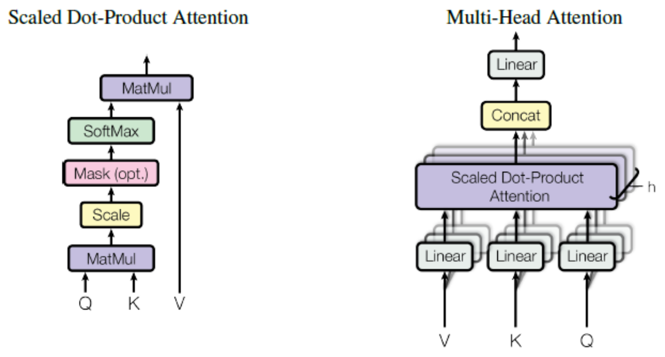


Figure 3.2.12: Left: Scaled Dot-Product Attention. Right: Multi-Head Attention consists of several attention layers in parallel. [18]

with poor predictive performance. An example of underfitting is shown by fitting a linear model to non-linear data. On Figure 3.2.13, we can see examples of under- and over-fitting in comparison to a perfect fit.

Numerous strategies have been devised to achieve the perfect fit, avoid overfitting, and improve the generalization of a model to unknown data. Regularization is a term that refers to a collection of approaches for preventing overfitting through the addition of information. The majority of regularization techniques try to strike a balance between bias and volatility. This entails reducing generalization error at the cost of increasing training error.

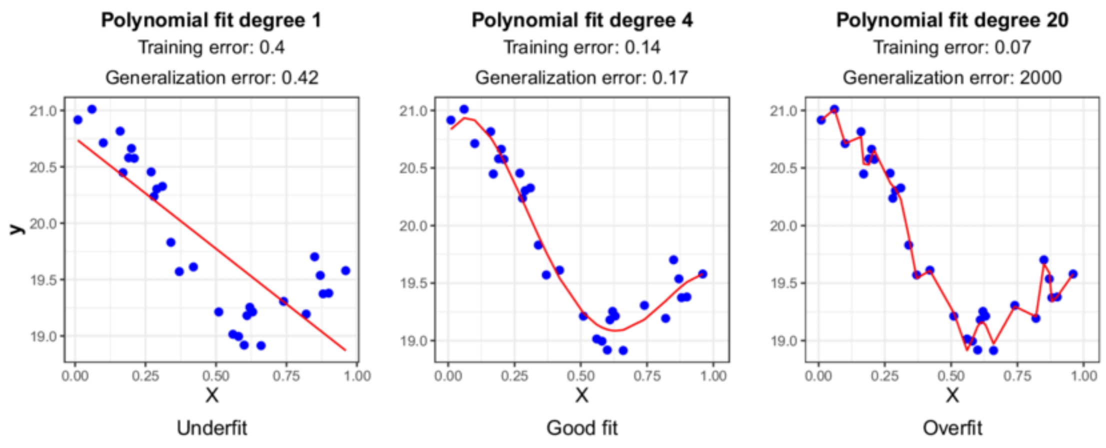


Figure 3.2.13: **Left**: Underfitting, the model is incapable of learning the data completely. **Center**: Ideal, the model acquires knowledge of the underlying data structure. **Right**: Overfitting, The model is overly complex and acquires noise as it learns from the training data. [51]

Dropout

Dropout is a basic yet effective regularization method for neural networks that is commonly utilized. Given a probability $p > 0$, each unit is dropped with that probability throughout training. When a unit is dropped, it is ignored during both forward and reverse passes, and its activations are set to zero. We can see the effect of dropout on a neural network during training in Figure 3.2.14. At test time, all units are used, but their values are scaled by a factor $1/p$ to account for the activations that were missed during training. Dropout forces the network to be self-sufficient and prevents units from developing co-dependencies on one another.

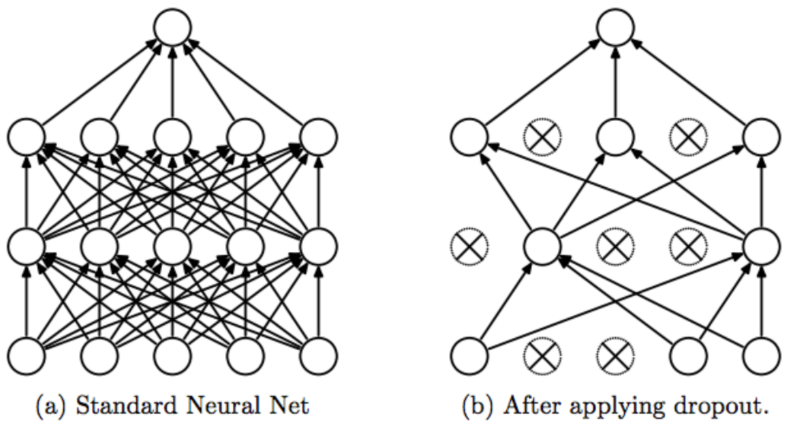


Figure 3.2.14: **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. [52]

Data Augmentation

Increased training data is a straightforward technique to lessen the danger of overfitting. Given the impossibility of collecting further data, one can build synthetic data from the existing dataset. This is referred to as data augmentation, and the method used to create these data points varies according to the domain and type of data. Small transforms such as flipping, rotating, rescaling, and cropping are frequently used with images. Although augmentation of natural language data is less prevalent, it is possible through synonym replacement. In general, data augmentation can be accomplished by injecting small amounts of random noise into the data to increase the model's robustness.

Chapter 4

NLP Techniques for Device-Aware Mapping of OpenCL Kernels

This chapter discusses the first contribution of this diploma thesis, our research into various deep neural network architectures for OpenCL kernels compiler optimization.

4.1 Problem Description

There are numerous scenarios during the compilation and execution of a program in which decisions must be made about how, or whether, to apply a particular optimization. Contemporary compilers and runtime environments are replete with hand-coded heuristics that perform this decision-making. Thus, program performance is contingent upon the quality of these heuristics.

Nonetheless, experts remain involved in the design process. Choosing the appropriate features is a manual process that requires a thorough knowledge of the system. In order to make heuristic construction efficient and inexpensive, humans must be removed from the process. In this regard, machine learning systems may be capable of extracting feature representations from source code automatically. Afterwards, other learning systems could employ these learnt feature representations as inputs to predict various down stream tasks.

We base our investigation on the work of Cummins et al. [1]. We will focus on one of their case studies, Heterogeneous Device Mapping, because the dataset is significantly larger than the dataset used in the second case study, selection of optimum Thread Coarsening Factor.

OpenCL provides a foundation for heterogeneous parallelism that is platform independent. This enables OpenCL programs to run smoothly across a wide variety of devices, from CPUs to GPUs to FPGAs. Given a program and a selection of possible execution devices, the question becomes which device should be used to maximize performance. For a CPU/GPU heterogeneous system, this presents a binary choice.

4.2 Examined DNN Architectures

This section will discuss the examined NN architectures, that expand our baseline model [1] and the underlying concept that inspired them. Each architecture is given a unique name for future evaluation.

DeepTune

The architecture proposed in [1], namely DeepTune, consists of a preprocessing stage that rewrites the source code in order to remove semantically irrelevant information and transforms it into tokens, a language model that takes as input those tokens and converts them to embedding vectors, of dimension $D = 64$, passing them a two layer LSTM network. DeepTune's final component is two fully connected artificial neural network layers. The initial layer is composed of 32 neurons. Each possible heuristic decision is represented by a single neuron in the second layer. The model's confidence that the associated choice is right is represented by the activation of each neuron in the output layer. Taking the output layer's argmax yields the decision with the highest activation. Figure 4.2.1 presents an overview of DeepTune, which serves as our **baseline** model for evaluating the exploration of our neural network architectures.

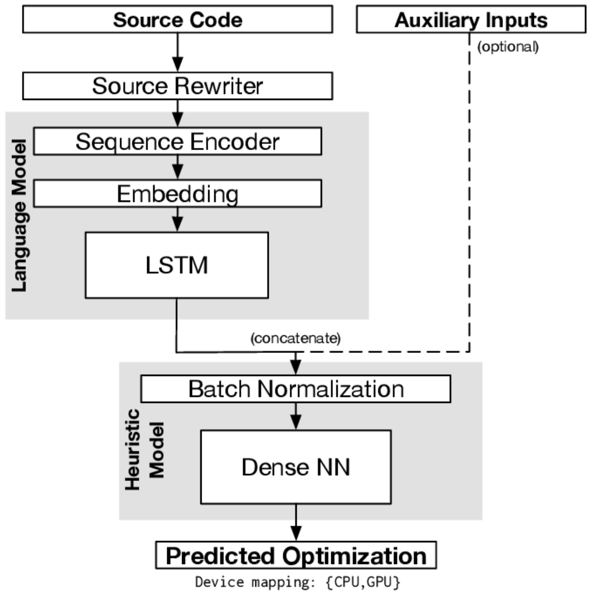
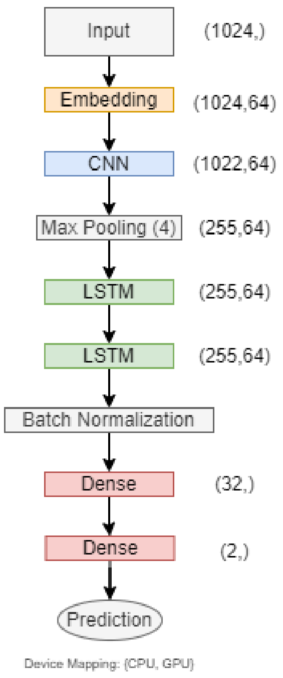


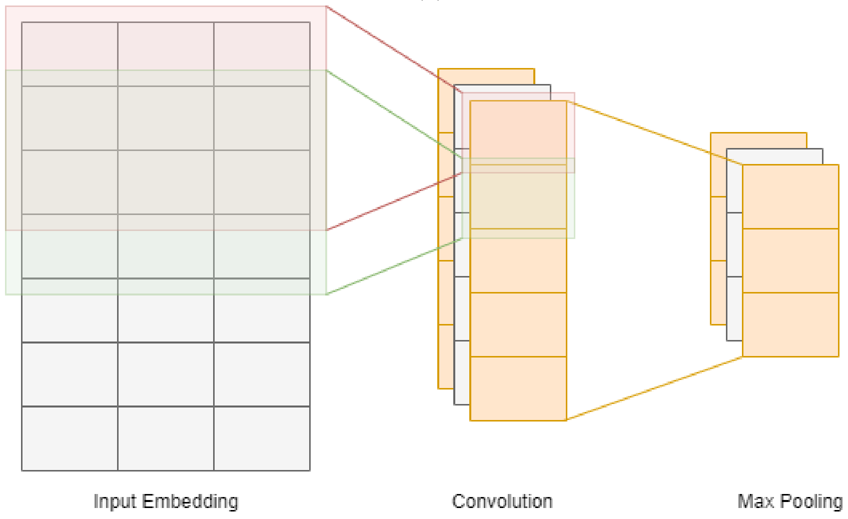
Figure 4.2.1: Overview of DeepTune architecture. [1]

CNN-DeepTune

Inspired from the work of Poznanski et al. [53], in which they are using CNNs on word images to extract n-grams for handwritten word recognition, we extended DeepTune by adding a CNN layer, that is showed in figure 4.2.2b and accepts the embedding layer's output as input. It is composed of 2 layers: one 1-d convolutional layer, with 64 filters, kernel size of 3 and striding step 1, and one max pooling layer of size 4. This method takes advantage of the inherent structure of a textual label by identifying common attributes shared by multiple words and dividing them into three equal parts. The embedding layer transmits the words to the convolutional layers in the form of sentences (in our case code instructions). Convolution layer convolve the input using pooling layers; pooling layers aid in reducing the representation of input phrases, input parameters, computation, and overfitting in the network. The architecture of CNN-DeepTune is depicted in Figure 4.2.2a, along with the output dimension of each layer.



(a)



(b)

Figure 4.2.2: (a) Overview of CNN-DeepTune architecture (b) The architecture of the CNN layer

BiLSTM-DeepTune

In Chapter 3 we presented the concept of bi-directional RNNs. Siami-Namini et al. [54] conducted a behavioral analysis and comparison of BiLSTM and LSTM models, concluding that BiLSTM models make more accurate predictions and LSTM models, but at a significant cost in terms of training time. Taking this into account, we replaced the DeepTune architecture's both LSTM layers with one Bi-LSTM layer. Figure 4.2.3 shows the architecture of BiLSTM-DeepTune.

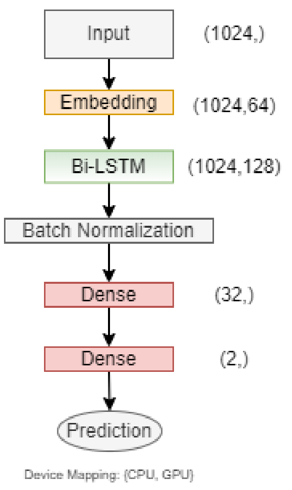


Figure 4.2.3: Overview of BiLSTM-DeepTune architecture

BiLSTM-LSTM-DeepTune

In order to investigate whether a combination of a Bi-LSTM and an LSTM layer will improve the model's prediction ability, we added an LSTM layer after the Bi-LSTM. Figure 4.2.4 shows the architecture of BiLSTM-LSTM-DeepTune.

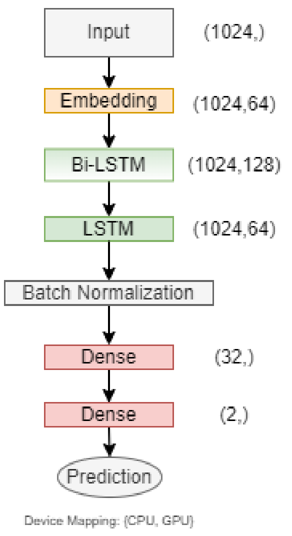


Figure 4.2.4: Overview of BiLSTM-LSTM architecture

BiLSTM-BiLSTM-DeepTune

This time, we added a second Bi-LSTM layer for the same purpose as before. Figure 4.2.5 shows the architecture of BiLSTM-BiLSTM-DeepTune.

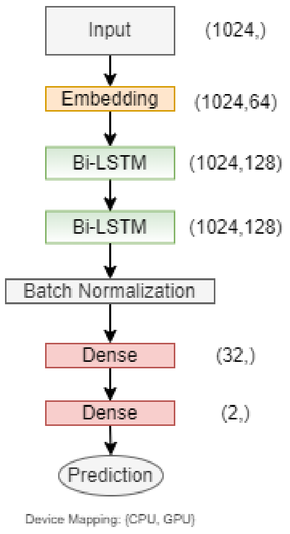


Figure 4.2.5: Overview of BiLSTM-BiLSTM architecture

Attention-DeepTune

In order to utilize the advantages of the attention mechanism, as discussed in 3.2.6, we added an attention layer after the output of the last LSTM output of the DeepTune model. The attention block highlights the importance of different features for model prediction by assigning weights to features. Figure 4.2.6 shows the architecture of Attention-DeepTune.

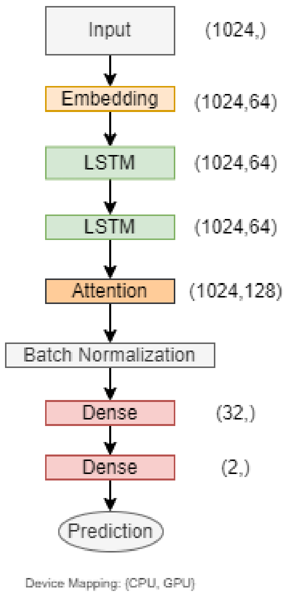


Figure 4.2.6: Overview of Attention-DeepTune architecture

CNN-BiLSTM-Attention

Figure 4.2.7 shows the architecture of CNN-BiLSTM-Attention, which integrates the ideas of the aforementioned architectures. It accepts a corpus as input and applies an embedding layer to it. The convolutional layer extracts high-level features, whereas the Bi-LSTM layer recognizes long-term relationships between words and the attention layer extracts the context information. Finally, we employ a classification layer based on the sigmoid function.

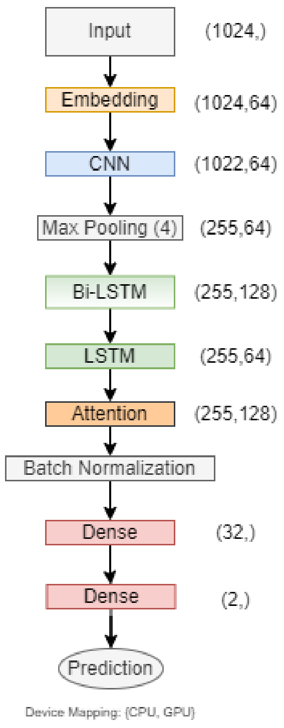


Figure 4.2.7: Overview of the proposed CNN-BiLSTM-Attention architecture

4.3 Evaluation

This section details the evaluation process we used to assess the models provided in Section 4.2.

4.3.1 Dataset

We base our evaluation on the dataset used in [1] in order to have accurate comparison between the different model architectures. The dataset contains 680 preprocessed and tokenized OpenCL kernels. Additionally, it contains the execution times for each kernel on two GPU devices, the AMD Tahiti 7970 and the NVIDIA GTX 970, as well as on an Intel Core i7-3820 CPU. Table 4.1 contains details about the CPU-GPU platforms.

The prediction target is the platform in which the execution time is lower. More precisely, when we examine the AMD GPU and Intel CPU cases, the target is [1,0] if the kernel runs faster on the GPU and [0,1] if the kernel runs faster on the CPU. This is also referred to as one-hot encoding. Likewise, for the NVIDIA GPU and the Intel CPU case.

4.3.2 Experimental Setup

We use stratified 10-fold cross-validation to evaluate the predictive quality of each model. Each program is randomly assigned to one of ten equal-sized sets; the sets are balanced to ensure a consistent

Platform	Frequency	Memory	Driver
Intel Core i7-3820	3.6 GHz	8 GB	AMD 1526.3
AMD Tahiti 7970	1000 MHz	3 GB	AMD 1526.3
NVIDIA GTX 970	1050 MHz	4 GB	NVIDIA 361.42

Table 4.1: Platform Details [1]

proportion of samples from each class across the whole set. A model is trained on all but one of the sets' programs and then tested on the programs from the unseen set. This procedure is done for each of the ten sets in order to provide a comprehensive prediction for the entire dataset.

All models were implemented in Python using Tensorflow[1] and Keras[2] backends. To ensure the most accurate comparison, we seeded our layers with the same number, in order to be initialized with the same weights. The maximum sequence length was set to 1024 and the learning rate at 10^{-3}. We used categorical cross entropy as loss function, a batch size of 64 and trained for 50 epochs. As optimizer, we used the adaptive learning rate optimization algorithm, Adam. The experiments were carried out on an NVIDIA Tesla V100 GPU. Finally, we used TensorBoard[3] to measure and visualize parameters like loss and accuracy.

4.3.3 Experimental Results

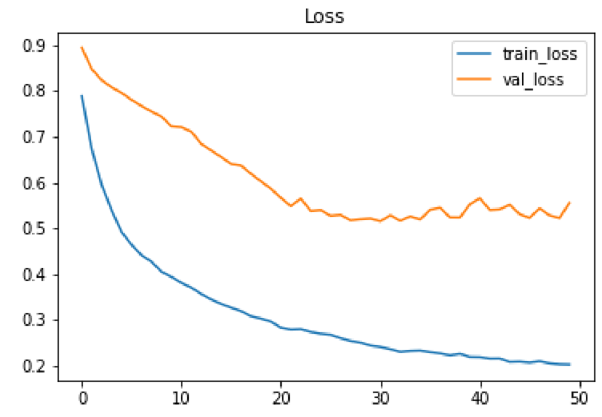
The average accuracy of each model, as measured in a 10-fold test set, is shown in Table 4.2. The best results are printed in **bold** font. DeepTune, the model suggested in [1], achieved an average accuracy of 81.76%, with better results on the AMD platform, which is consistent across all architectures. The CNN-DeepTune model already outperforms the baseline by 2.65% on AMD platform and 1.18% on average, but performs slightly worse on the NVIDIA platform. The two bi-LSTM layers in the BiLSTM-BiLSTM-DeepTune decrease the model's accuracy. The BiLSTM-DeepTune achieves similar scores with the BiLSTM-LSTM-DeepTune on the NVIDIA platform, with the latter performing better in the AMD platform. The addition of an attention layer in Attention-DeepTune model appears to give the best results, so far, on the NVIDIA platform. As a result of the preceding findings, it was assumed that merging the best models would yield the best outcomes. Our **proposed** CNN-BiLSTM-Attention model proves this assumption by outperforming the baseline model and all other models. It scored 4.12% higher than the baseline on the AMD platform and 1.18% higher on the NVIDIA platform, resulting in an average accuracy increase of 2.65 percent.

The average loss and accuracy graphs of the best CNN-BiLSTM-Attention model over the 10-fold cross train and validation sets are shown in Figures 4.3.1a 4.3.1b and 2.1b. As can be seen, the validation loss decreases until epoch 40, at which point it remains pretty constant. Because the training loss continues to decrease, we stop training until epoch 50 to avoid overfitting.

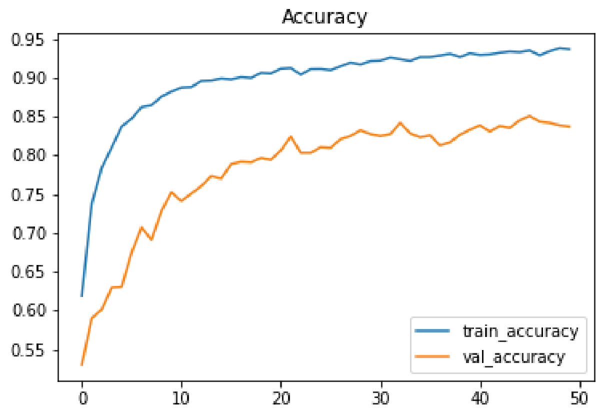
	AMD Tahiti 7970	**NVIDIA GTX 970**	**Average**
DeepTune (baseline)	83.23%	80.29%	81.76%
CNN-DeepTune	85.88%	80.00%	82.94%
BiLSTM-DeepTune	82.80%	80.44%	81.62%
BiLSTM-LSTM-DeepTune	84.85%	80.88%	82.87%
BiLSTM-BiLSTM-DeepTune	83.38%	76.97%	80.14%
Attention-DeepTune	83.9%	81.03%	82.50%
CNN-BiLSTM-Attention	**87.35%**	**81.47%**	**84.41%**

Table 4.2: Accuracy of every model; Best results are highlighted in **bold** fold

[1]Tensorflow: https://www.tensorflow.org/
[2]Keras: https://keras.io/
[3]TensorBoard: https://www.tensorflow.org/tensorboard/



(a)



(b)

Figure 4.3.1: (a) Train and validation loss of the best model (b) Train and validation accuracy of the best model

Chapter 5

NLP Techniques for Thread/Block Coarsening Optimization of CUDA Kernels

This chapter details our efforts toward developing a comprehensive approach for optimizing CUDA kernels.

5.1 Problem Description

As discussed in Chapter 3 in GPGPU high-performance computing, many factors have to be taken into account when one wants to optimize an application, and those factors differ when it comes to what are you trying to optimize. Below we will focus on two optimization techniques, thread coarsening and block coarsening.

Thread coarsening is a parallel programming optimization technique that combines the actions of two or more threads. This thread coarsening factor is then used to minimize the number of threads that must be performed. This optimization may be useful for specific combinations of programs and architectures, for example, programs with a high potential for Instruction-level Parallelism on architectures with very long instruction words.

Block Coarsening is a kernel optimization technique that combines the workloads of two or more thread blocks, reducing their overall number while retaining the thread count per block. As a consequence, it consolidates multiple adjacent blocks and may be effective in addressing the challenges inherent with large amounts of fine-grained parallelism.

We will apply those transformations on a large scale dataset of CUDA kernels, provided by the work of Bjertnes et al. [2], measure their runtimes and apply machine learning on them

5.2 Proposed Methodology for optimal Thread/Block coarsening factor prediction

Given a dataset of CUDA kernels we need to be able to profile them to our tasks, that is, build a thread or block coarsening factor prediction model. To do that they need to be transformed to a computer understandable representation. Figure 5.2.1 illustrate the flow of the whole method that is discussed in the following steps.

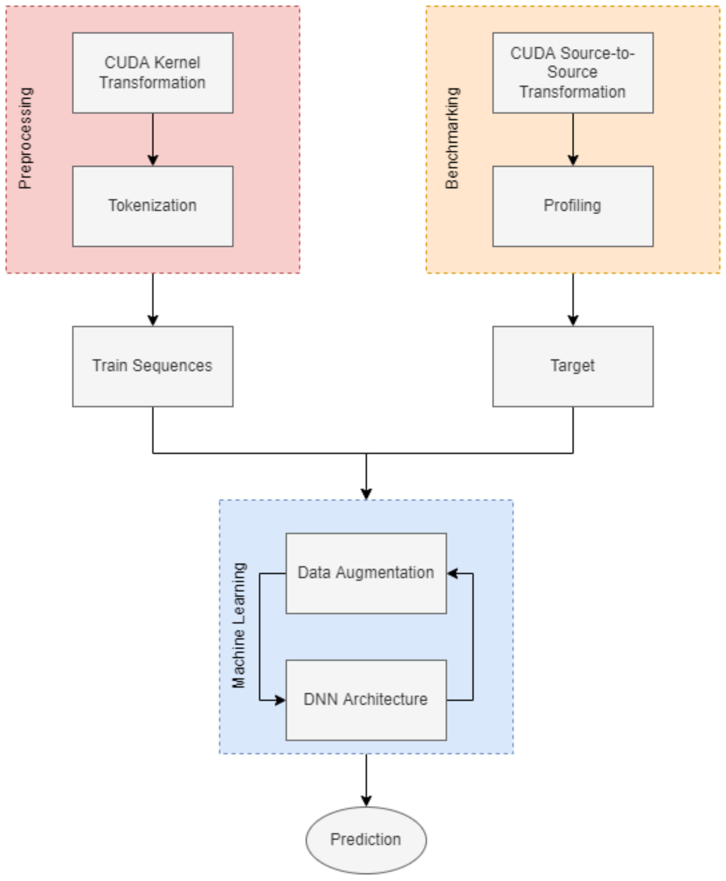


Figure 5.2.1: Proposed methodology for optimal thread/block coarsening factor mapping of CUDA kernels

5.2.1 Step 1: Preprocessing of CUDA Kernels

CUDA kernel transformation

As seen in Listing 5.1, handwritten code may include comments that consist semantically irrelevant information, macros and conditional compilation (e.g. #define, #ifdef, etc), bad format (inconsistency in indentation, spaces and newlines, etc.) and arbitrary variable and function names.

The Clang project [7] provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM project [16]. We use a special configuration of clang in order to extract the output of the preprocessor stage. This output, which comprises roughly 40.000 lines, is stripped and the cleaned kernel code is retained. Following the preprocessing pass, Listing 5.2 has the device code from Listing 5.1, without comments, macros and conditional compiling.

```
#define numBlock_x 32
#define numBlock_y 32
__global__ void
Kernel(float *Md, float *Nd, float *Pd, int Width)
{

  // Sanity Check
  assert(blockDim.x == numBlock_x);
  assert(blockDim.y == numBlock_y);

  // Calculate the column index of the Pd element, denote by x
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  // Calculate the row index of the Pd element, denote by y
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  float Pvalue = 0;
  // each thread computes one element of the output matrix Pd.
  for (int k = 0; k < Width; ++k) {
    Pvalue += Md[y*Width + k] * Nd[k*Width + x];
  }

  // write back to the global memory
  Pd[y*Width + x] = Pvalue;
}
```

Listing 5.1: Device Code [55]

```
__global__ void
Kernel(float *Md, float *Nd, float *Pd, int Width)
{

  assert(blockDim.x == 32);
  assert(blockDim.y == 32);

  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  float Pvalue = 0;
  for (int k = 0; k < Width; ++k)
  {
    Pvalue += Md[y*Width + k] * Nd[k*Width + x];
  }

  Pd[y*Width + x] = Pvalue;
}
```

Listing 5.2: Device Code after the Preprocessor Stage

In order to enforce code style, we utilize *clang-format*, a precompiled program that is a commonly used C++ code formatter. It provides an option to define code style options in YAML-formatted files. Listing 5.3 contains the device code that is style-enforced.

```
__global__ void Kernel(float *Md, float *Nd, float *Pd, int Width) {
    assert(blockDim.x == 32);
    assert(blockDim.y == 32);
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
        Pvalue += Md[y * Width + k] * Nd[k * Width + x];
    }
    Pd[y * Width + x] = Pvalue;
}
```

Listing 5.3: Device Code after the Style Enforcement

To eliminate arbitrary variable and function names, we develop a clang program called cuda-rewriter that parses the AST. It rewrites the supplied source code in a uniform manner and with a consistent naming strategy for identifiers. We employ a lower case letter alphabet renaming system for variables declared within the kernel scope $[a, b, \ldots, z, aa, ab, \ldots]$ and an upper case letter alphabet renaming scheme for function names declared within the kernel scope $[A, B, \ldots, Z, AA, AB, \ldots]$. Variables and functions declared outside kernel's scope are left unchanged (e.g. threadIdx.x). Listing 5.4 contains the device code following the transformation.

```
__global__ void A(float *a, float *b, float *c, int d) {
    assert(blockDim.x == 32);
    assert(blockDim.y == 32);
    int e = threadIdx.x + blockIdx.x * blockDim.x;
    int f = threadIdx.y + blockIdx.y * blockDim.y;
    float g = 0;
    for (int h = 0; h < d; ++h) {
        g += a[f * d + h] * b[h * d + e];
    }
    c[f * d + e] = g;
}
```

Listing 5.4: Device Code after the Variable/Function Renaming

The purpose of source standardization is to optimize the flow of modeling source code by guaranteeing that insignificant semantic changes in programs, such as variable name choice or comment inclusion, have no effect on the taught model.

Tokenization

Tokenization is the process that transforms a text input into a sequence of discrete integers, referred to as token ids. So in order to feed the machine learning model with input, the textual representation of program codes must be represented as numeric sequences. This is achieved by using the flex (Fast Lexical Analyzer Generator), a framework, written in C language, for generating lexical analyzers (scanners or lexers) [8, 9]. Using complex rules and patterns we translate variables, functions, numeric values, operators, and characters to unique tokens (e.g. "int", "-", "++") and assign a unique integer identifier, forming a vocabulary and numeric sequences. We assume that spaces, newlines, and indentation are irrelevant to the neural model and hence we do not tokenize them, therefore shortening the sequence and the vocabulary. Sequences are padded at the beginning to a predetermined length of N with the special token '<pad>' and its associated id, that is an out of vocabulary number (e.g. 0, -1). If the sequence exceeds the N tokens, it is truncated at the end, retaining the first N tokens. Listing 5.5 provides an example input of a CUDA kernel. Table 5.1 contains the generated tokens with their respective integer mapping and table 5.2 contains the generated numeric sequence, that will be the input of the neural model.

```
__global__ void A(float *a, int *b) {
    int c = threadIdx.x + blockIdx.x * blockDim.x;
    float d = .1;
}
```

Listing 5.5: Simple example of CUDA kernel

5.2.2 Step 2: Automatic Source-to-Source Compilation and Profiling

After compiling a dataset of kernels and their numeric representation, that computers can understand we need to profile every kernel and extract the execution time for each coarsen factor, in order to establish a prediction target for the machine learning model.

token	id	token	id	token	id
'__global__'	1	'int'	9	'x'	17
'void'	2	'b'	10	'+'	18
'A'	3	')'	11	'blockId'	19
'('	4	'{'	12	'blockDim'	20
'float'	5	'gid'	13	';'	21
'*'	6	'='	14	'c'	22
'a'	7	'threadId'	15	'0'	23
','	8	'.'	16	'}'	24

Table 5.1: The vocabulary mapping of tokens to their respective integer identifiers

<pad>	<pad>	<pad>	<pad>	<pad>	1	2	3
4	5	6	7	8	9	6	10
11	12	9	13	14	15	16	17
18	19	16	17	6	20	16	17
21	5	22	14	16	23	21	24

Table 5.2: Generated numeric sequence

Timing Kernel Execution with CPU Timers

Listings 5.6 and 5.1 present the host code and the device code, respectively, of a CUDA application. The host code is responsible for deploying each kernel with it's resources and measuring the time it takes to complete its execution. The CPU must wait until the kernel completes before accessing the results, due to the fact that CUDA kernel launches do not stop the calling CPU thread [56]. This is accomplished by using the explicit synchronization barrier cudaDeviceSynchronize() after the kernel launch and prior to calculating the end time. Without this barrier, this code would measure the kernel launch time and not the kernel execution time. The execution time is measured in microseconds.

In order to perform a warm-up phase to the GPU [57] we launch the kernel two times and measure the last launch for our profiling. Finally, to ensure that the kernel does not contain any errors, for example segmentation fault, division by zero, and so forth, we perform explicit error checking during the warm-up phase. If the kernel generates any form of error, it is excluded from our profiling. We use explicit error checking since the kernel runtime does not raise an exception in the CPU runtime when an error occurs in CUDA Programming.

```
int main(){
    // ...
    int size = {1024, 1024};
    int threads_block = {32, 32};
    dim3 dimGrid(size[0]/threads_block[0], size[1]/threads_block[1]);
    dim3 dimBlock(threads_block[0], threads_block[1]);
    // GPU Warm-up Stage
    Kernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
    cudaDeviceSynchronize();
    // Kernel Error Checking
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess){
        printf(`Kernel Error: %s`, cudaGetErrorString(err));
        exit(1);
    }
    // Measure the Execution Time of the Kernel
    auto start = steady_clock::now();
    Kernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
    cudaDeviceSynchronize();
    auto end = steady_clock::now();
    auto usecs = duration_cast<duration<float, microseconds::period>>(end - start);
    // ...
}
```

Listing 5.6: Host Code

Automatic Source to Source Compiler

In order to make a source to source compiler that takes as input a kernel and produces a thread or block coarsened kernel, we use the Perl language. Perl is a scripting language that performs string manipulations efficiently. To begin, we iterate the kernel in order to identify if it is one dimensional or two dimensional. Following that, we increase its parameters by adding the coarsen factor for the x- and y-axis (if it is two-dimensional), for which we make a folded for loop, shown in Listing 5.7. Finally, we apply the appropriate modifications to each case, as indicated in Table 5.3. Listings 5.9 and 5.9 show the transformed kernels for thread and block coarsening respectively. Similarly, listings 5.8 and 5.8 shows the host code to deploy and measure the aforementioned kernels.

	Thread Coarsening	**Block Coarsening**
threadIdx.{x,y}	threadIdx.{x,y}*tc_{x,y}+index_{x,y}	threadIdx.{x,y}
blockDim.{x,y}	blockDim.{x,y}*tc_{x,y}	blockDim.{x,y}
blockIdx.{x,y}	blockIdx.{x,y}	blockIdx.{x,y}*bc_{x,y}+index_{x,y}
gridDim.{x,y}	gridDim.{x,y}	gridDim.{x,y}*bc_{x,y}

Table 5.3: Transformation mapping for thread and block coarsening

```
for (int indexx = 0; indexx < tc_x; indexx++)
    // If the kernel is one-dimensional the following line is not printed
    for (int indexy = 0; indexy < tc_y; indexy++)
```

Listing 5.7: Folded for loop

```
int main(){
    // ...
    int tc_x = 2, tc_y = 2;
    int size = {1024, 1024};
    int threads_block = {32, 32};
    dim3 dimGrid(size[0]/threads_block[0], size[1]/threads_block[1]);
    dim3 dimBlock((threads_block[0]/tc_x), (threads_block[1]/tc_y));
    auto start = steady_clock::now();
    Kernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width, tc_x, tc_y);
    cudaDeviceSynchronize();
    auto end = steady_clock::now();
    auto usecs = duration_cast<duration<float, microseconds::period>>(end - start);
    // ...
}
```

Listing 5.8: Thread Coarsened Host Code

```
__global__ void Kernel(float *Md, float *Nd, float *Pd, int Width, int tc_x, int tc_y){
  for (int indexx = 0; indexx < tc_x; indexx++)
    for (int indexy = 0; indexy < tc_y; indexy++){
      // Calculate the column index of the Pd element, denote by x
      int x = (threadIdx.x*tc_x+indexx) + blockIdx.x * (blockDim.x*tc_x);
      // Calculate the row index of the Pd element, denote by y
      int y = (threadIdx.y*tc_y+indexy) + blockIdx.y * (blockDim.y*tc_y);

      float Pvalue = 0;
      // each thread computes one element of the output matrix Pd.
      for (int k = 0; k < Width; ++k) {
        Pvalue += Md[y*Width + k] * Nd[k*Width + x];
      }

      // write back to the global memory
      Pd[y*Width + x] = Pvalue;
    }
}
```

Listing 5.9: Thread Coarsened Device Code

```
int main(){
    // ...
    int bc_x = 2, bc_y = 2;
    int size = {1024, 1024};
    int threads_block = {32, 32};
    dim3 dimGrid((size[0]/threads_block[0])/bc_x, (size[1]/threads_block[1])/bc_y);
    dim3 dimBlock(threads_block[0], threads_block[1]);
    auto start = steady_clock::now();
    Kernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width, bc_x, bc_y);
    cudaDeviceSynchronize();
    auto end = steady_clock::now();
    auto usecs = duration_cast<duration<float, microseconds::period>>(end - start);
    // ...
}
```

Listing 5.10: Block Coarsened Host Code

```
__global__ void Kernel(float *Md, float *Nd, float *Pd, int Width, int bc_x, int bc_y){
  for (int indexx = 0; indexx < bc_x; indexx++)
    for (int indexy = 0; indexy < bc_y; indexy++){
      // Calculate the column index of the Pd element, denote by x
      int x = threadIdx.x + (blockIdx.x*bc_x+indexx) * blockDim.x;
      // Calculate the row index of the Pd element, denote by y
      int y = threadIdx.y + (blockIdx.y*bc_y+indexy) * blockDim.y;

      float Pvalue = 0;
      // each thread computes one element of the output matrix Pd.
      for (int k = 0; k < Width; ++k) {
        Pvalue += Md[y*Width + k] * Nd[k*Width + x];
      }

      // write back to the global memory
      Pd[y*Width + x] = Pvalue;
    }
}
```

Listing 5.11: Block Coarsened Device Code

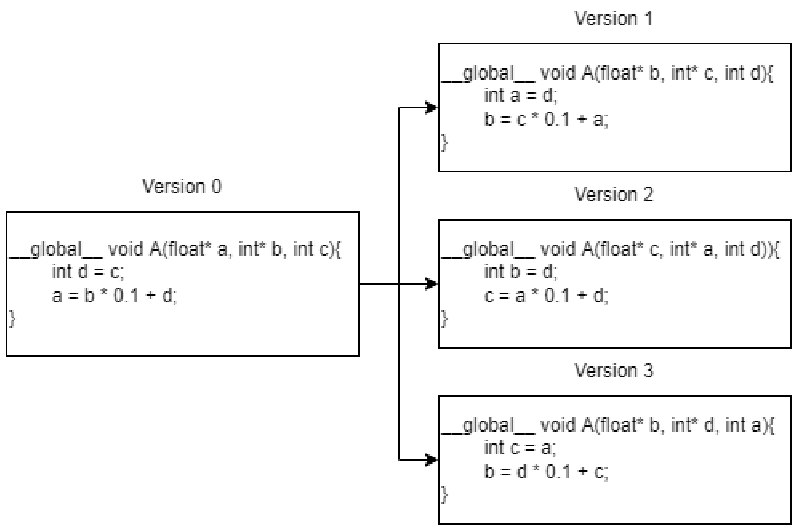
Large Scale Profiling

Finally, in order to accomplish this profiling on a large scale of CUDA kernels, we developed two scripts, one for thread coarsening and one for block coarsening, that transforms kernels for a range of coarsening factors, compiles and executes them, and stores their execution times to a file. Python was used to script this procedure owing of its simplicity of interaction with directories, files, and text.

5.2.3 Step 3: Proposed DNN Models

To develop a machine learning model that predicts the optimal Thread and Block Coarsening Factor, we use the CNN-BiLSTM-Attention model from Chapter 4 as our best architecture.

Additionally, we propose a synthetic data augmentation methodology, that comprises of two parts: function parameter permutations and variable renaming. An example of a permutation set of the kernel's parameters, which produces $N!$ versions of the kernel, where N is the number of the kernel's parameters, is shown in figure 5.2.3. An example of variable renaming is shown in figure 5.2.2 for three renaming schemes.



```
__global__ void A(float* a, int* b, int c){
      int d = c;
      a = b * 0.1 + d;
}
```

```
__global__ void A(float* b, int* c, int d){
      int a = d;
      b = c * 0.1 + a;
}
```

```
__global__ void A(float* c, int* a, int d)){
      int b = d;
      c = a * 0.1 + d;
}
```

```
__global__ void A(float* b, int* d, int a){
      int c = a;
      b = d * 0.1 + c;
}
```

Figure 5.2.2: Example of a kernel's variable renaming

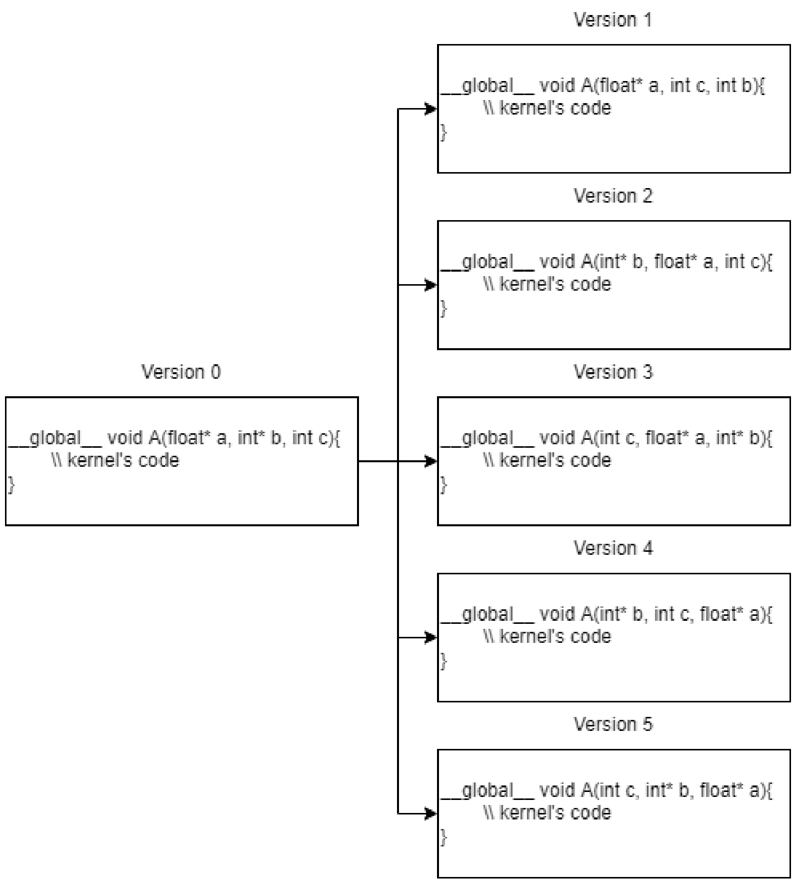

Version 0
__global__ void A(float* a, int* b, int c){
\\ kernel's code
}
Version 1
__global__ void A(float* a, int c, int b){
\\ kernel's code
}
Version 2
__global__ void A(int* b, float* a, int c){
\\ kernel's code
}
Version 3
__global__ void A(int c, float* a, int* b){
\\ kernel's code
}
Version 4
__global__ void A(int* b, int c, float* a){
\\ kernel's code
}
Version 5
__global__ void A(int c, int* b, float* a){
\\ kernel's code
}

Figure 5.2.3: Example of a kernel's parameter permutations

5.3 Evaluation

This section will cover the approach stated before and its application to a large scale and novel dataset of CUDA kernels.

5.3.1 Dataset

While prior research has concentrated on OpenCL GPU code, there seem to have been no efforts to create a significant CUDA-based dataset. Bjertnes et al. [2] attempt to mitigate this gap. They retrieved publicly accessible source code from GitHub. Each project was restructured as a collection of executable CUDA kernels, yielding 20251 compilable kernels; of these, 19683 yielded results that could be used as a database of executable CUDA kernels.

Along with the CUDA kernels, the LS-CAT dataset provides 5028536 related measured **launch** times, which include GTX 980 and T4 results with a variety of kernel, block, and matrix sizes. In CUDA Launch or Latency time refers to the time spend by the CPU in launching the GPU kernels on the device. It is different from the Execution time that is the time required for the kernel to complete. The execution time includes the launch time. In order to make a prediction model based on coarsening factors we are interested in the execution time, and hence we must apply the methodology described in the previous section.

5.3.2 Experimental Setup

The compilation, execution, and execution time measurements were all performed on a NVIDIA Tesla V100S 32 GB GPU system. The compilation architecture was set on NVIDIA Volta (sm_70) with no compiler optimization (flag -O0). We opted to build huge benchmarks for all kernels, therefore we set the input size to around 256 MB per input array. We set all kernel input parameters (variables and arrays) to neutral values (e.g. 0 or 1). The grid block size was set to 262144 for one-dimensional kernels and to 512 on each axis for two-dimensional kernels. The total number of threads per block was set to 1024 for one-dimensional kernels and 32 on each axis for two-dimensional kernels. The coarsening factors investigated were in the range {1, 2, 4, 8, 16} on both the x and y axes (if y-axis exists).

We base the generation of each kernel's host code on the LS-CAT project's profiling tool [2]. We extended their host codes by adding synchronization barriers, for execution time measurement, and explicit error checking (see 5.2.2). Memory allocation and kernel invocation are left unchanged.

We used clang 6 and llvm v11 to perform the CUDA kernel transformations. The CUDA version was 11.2 and the NVIDIA driver version was 460.8 at the time of writing. For the execution of the kernels we used Python subprocesses.

For the configuration of the machine learning model we utilized the same setup as outlined in Chapter 4. The only difference is in the prediction target, and hence the output of the model. We examine the efficiency of our model and data on different prediction problems:

1. Binary Classification
    - One-dimensional thread coarsened kernels
2. Multi-class Classification
    - One-dimensional thread coarsened kernels
    - One-dimensional block coarsened kernels

5.3.3 Experimental Results

Following the format described in Section 5.2, the experimental results are presented as a series of steps.

Step 1: Preprocessing of CUDA Kernels

When applied to the LS-CAT dataset, the CUDA kernel transformation resulted in 17331 rewritten kernels out of a total of 20251, or the 85.59%. This is most likely due to incompatible CUDA versions between the system and the kernel, and/or path issues.

We generate 17331 train-ready sequences and their associated vocabulary after tokenizing the rewritten kernels. Although, as a sanity check, we execute a sequence comparison and discovered that there were only 7529 unique kernels and the remaining 9802 were duplicates. This occurred, we assume, because Bjertnes's et al. [2] system was incapable of identifying duplicate kernels. This resulted in a 57% reduction in the size of the dataset.

Step 2: Source-to-Source Compiler and Profiling

Following the profiling stage, with the configuration described in 5.3.2, we compile and run 7529 kernels. Out of these, 723 kernels failed to compile and 3367 kernels, identified by our error checking system, failed in runtime with segmentation fault. In summary, out of 20251 total kernels, we ended up with 3477 kernels that we can use, that is the 17.17% of the initial LS-CAT dataset. We assume that in the creation pipeline of LS-CAT, Bjertnes et al. did not include these error checks.

For the remaining kernels, we profile them for the thread and block coarsening factors configuration described in 5.3.2. Specifically, below we present our profiling results for the thread and block coarsening transformations.

(a) **Thread Coarsening**

For the profiling process we split the kernels to one- and two-dimensional. There are 2112 one-dimensional kernels, while the remaining 1365 kernels are two-dimensional. The distribution of the speedup, of every coarsen factor in comparison with the uncoarsened kernel, of both splits are shown in figures 5.3.1 and 5.3.2, respectively. We observe that for the one-dimensional kernels we achieve an average speedup of 1.7, for all factors, except factor 2, that present lower speedup. For the two-dimensional kernels we note that increasing the coarsening factor of y-axis and keeping the coarsening factor of x-axis low increases performance. Additionally, we see that increasing the coarsening factor on the x-axis decreases the average performance. Table 5.4 maps each kernel to their best coarsening factor, sorted by the number of kernels. We can see that it is pretty balanced to the higher coarsening factors.

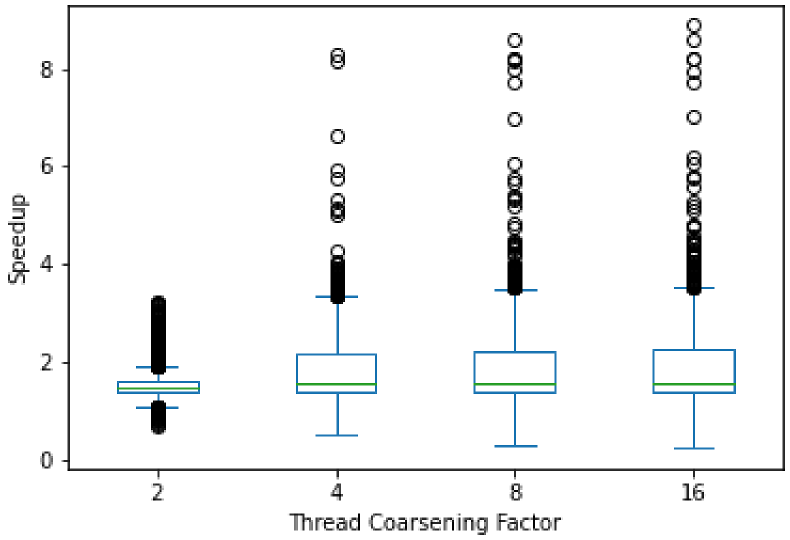
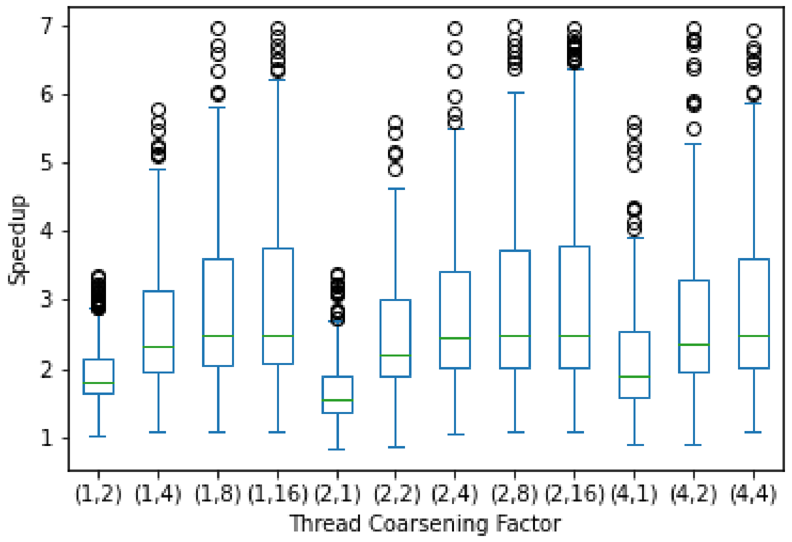


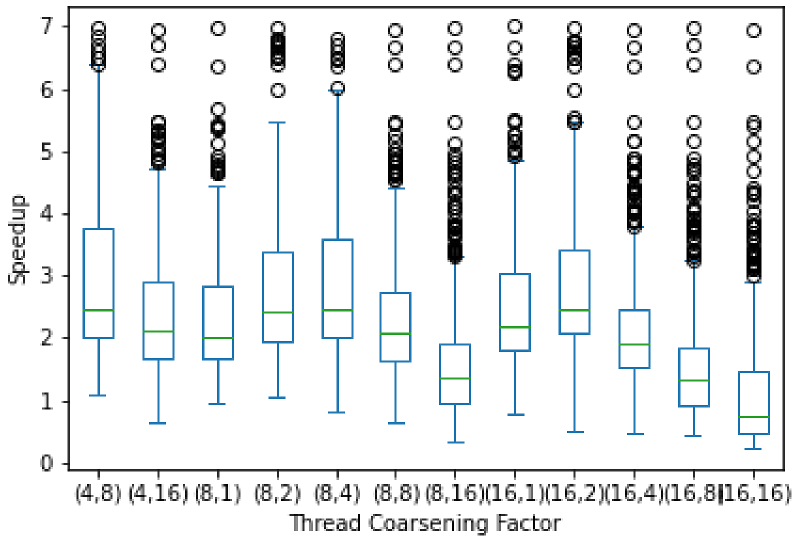
Figure 5.3.1: **One-dimensional kernels:** Distribution of speedup for thread coarsening factors {2, 4, 8, 16}

Thread Coarsening Factor {tc_x}	**# Kernels**
16	528
4	517
8	490
1	153
2	55

Table 5.4: One-dimensional kernel mapping to their optimal thread coarsening factor. Sorted by kernels.



(a)



(b)

Figure 5.3.2: **Two-dimensional kernels:** (a) Distribution of speedup for thread coarsening factors {(1, 2) ... (4, 4)} (b) Distribution of speedup for thread coarsening factors {(4, 8) ... (16, 16)}

(b) **Block Coarsening**

We perform the same splitting to one- and two-dimensional kernels as before. The distribution of the speedup, of every coarsen factor in comparison with the uncoarsened kernel, of both splits are shown in figures 5.3.3 and 5.3.4, respectively. We observe, immediately that performing block coarsening to the kernels improves radically their performance. This is happening, we assume, due to the large grid size that we chose to launch the kernels (see 5.3.2). Especially when it comes to the two-dimensional kernels, we observe that increasing the block coarsening factor on both axes achieves the best results, in contrast to thread coarsening. Table 5.5 depicts the mapping of each kernel to their best block coarsening factor. As we can see, the distribution is highly unbalanced to the largest coarsening factor.

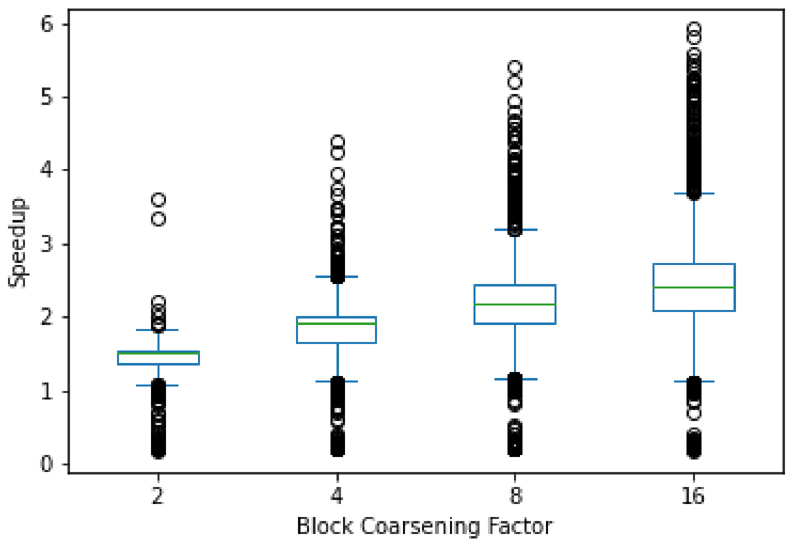
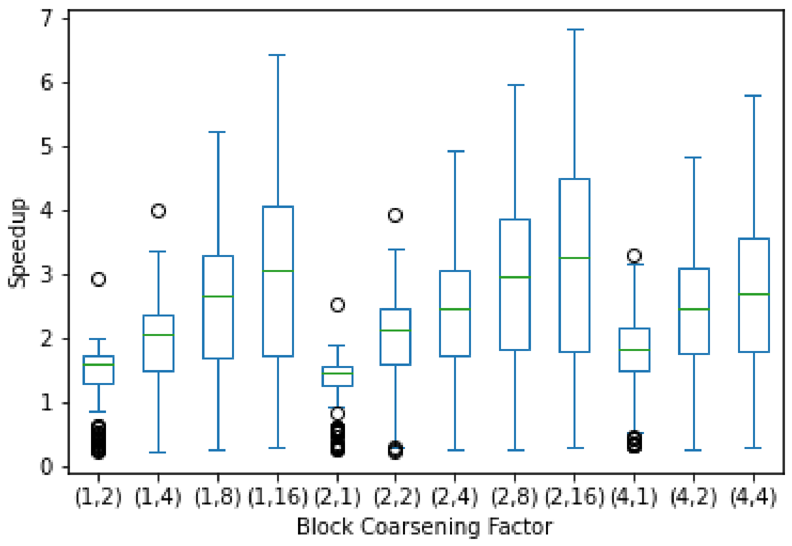


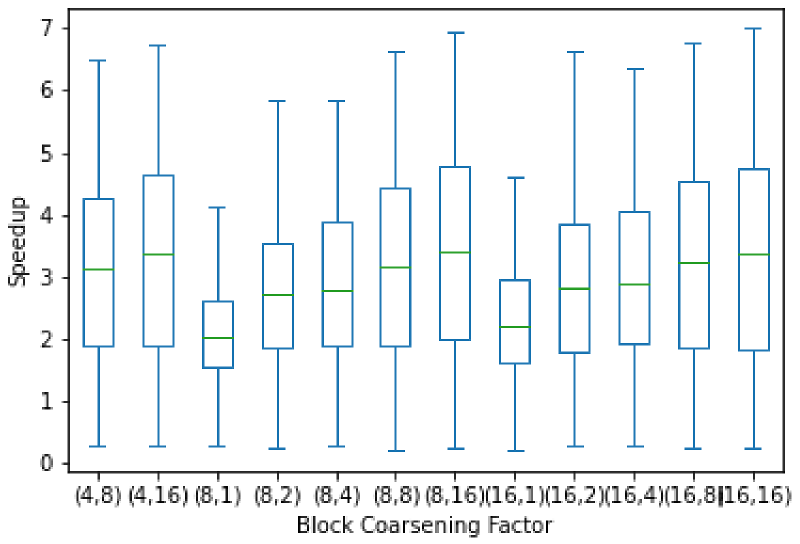
Figure 5.3.3: **One-dimensional kernels:** Distribution of speedup for block coarsening factors {2, 4, 8, 16}

Block Coarsening Factor {bc_x}	# **Kernels**
16	1369
8	68
4	30
1	41
2	20

Table 5.5: One-dimensional kernel mapping to their optimal block coarsening factor. Sorted by kernels.



(a)



(b)

Figure 5.3.4: **Two-dimensional kernels:** (a) Distribution of speedup for block coarsening factors {(1, 2) ... (4, 4)} (b) Distribution of speedup for block coarsening factors {(4, 8) ... (16, 16)}

Step 3: Machine Learning

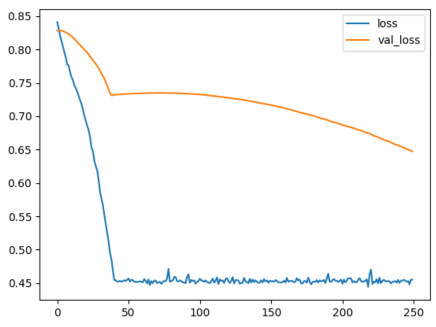
Following the results of the previous steps we perform our machine learning methodology on the predictions problems described in 5.3.2. We will not examine any two-dimensional kernels neither the binary classification problem for block coarsening, due to the highly imbalanced classes that derived from the results of the previous step.

1. **Binary classification of 1-dimensional thread coarsened kernels**

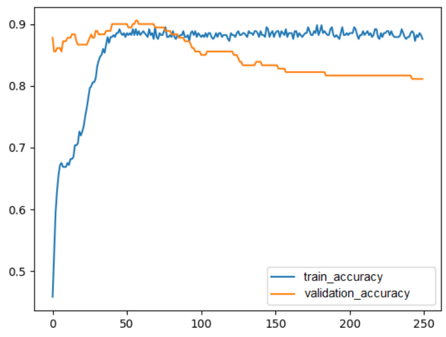
   For this problem, our classes 0 and 1 consist of optimum kernels without coarsening (tcf: 1) and optimal kernels with coarsening (tcf: 2, 4, 8, 16). As we can see on table 5.4 there are 153 kernels that operate optimally without thread coarsening, and 1590 kernels that perform optimally with thread coarsening transformation. Thus, we randomly choose 153 kernels from the 1590 in order to address this imbalance.

   Because the dataset is pretty small and our neural network architecture is deep, the training process is susceptible to overfitting. In order to overcome this problem we perform grid search and chose to reduce the learning rate of the model to 0.0001, increase the training epochs to 250, increase the batch size to 256 and add a strict dropout layer (see 3.2.8) after the last dense layer. The model achieves **84%** accuracy and an average speedup of 1.7.

   Figures 5.3.5a and 5.3.5a show the plots of the loss and accuracy, respectively, of the model. We can clearly see the overfitting problem in the loss plot, but with the regularization techniques we manage to decrease the validation loss.



(a)



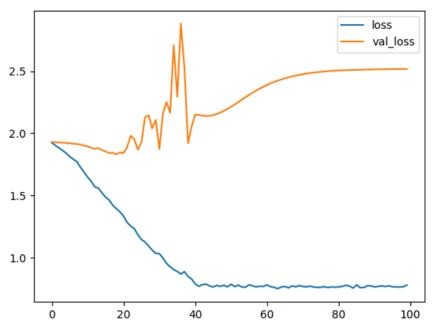
(b)

Figure 5.3.5: (a) Train and validation loss of the binary classification problem for thread coarsening (b) Train and validation accuracy of the binary classification problem for thread coarsening

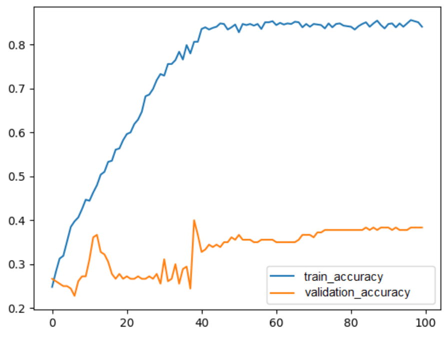
2. **Five-class classification of 1-dimensional thread coarsened kernels**

   For this problem, our classes 0, 1, 2, 3 and 4 correspond to the optimal coarsening factor of each kernel, tcf: 1, 2, 4, 8 and 16 respectively. We choose to balance the dataset by randomly selecting 153 kernels from each class, except from class 2 (see table 5.4 that we selected all of them, that is 55 kernels. The model achieves 42% accuracy. As we can see from the plots 5.3.6a and 5.3.6b, that depict the loss and accuracy plots of the model, respectively, the model suffers from overfitting. We performed the same grid search as before but the problem persists.

   In order to overcome the problem of overfitting we utilized our proposed synthetic data augmentation methodology, that comprises of two parts (see 5.2.3). By combining these two strategies we create 5000 synthetic kernels per each class, but even with data augmentation the problem of overfitting remains and the accuracy on the test set is not improving. Hence, it is safe to assume that the problem is due to the low quality of data.



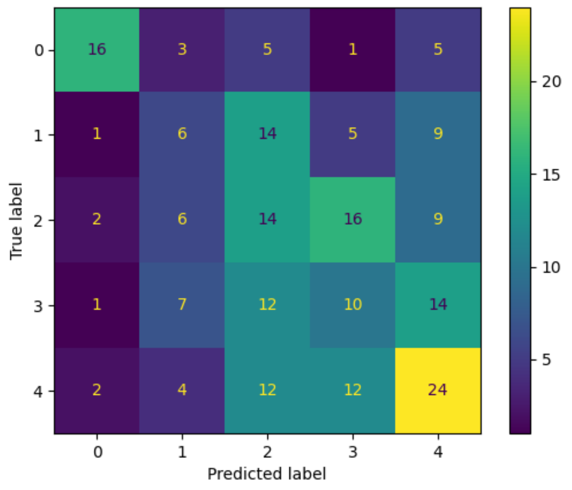
(a)



(b)

Figure 5.3.6: (a) Train and validation loss of the five-class classification problem for thread coarsening (b) Train and validation accuracy of the five-class classification problem for thread coarsening

It is of interest to take a closer look to the miss-classified kernels. The confusion matrix for the predicted classes of kernels present in the test set is shown in Figure 5.3.7. We can see that for the kernels that the model fails to classify correctly, their predicted class is adjacent to their optimal. Finally, Figure 5.3.8 shows a boxplot illustrating the speedup of those kernels. We observe that even if the model fails to correctly classify those kernels, the predicted classes achieve an average speedup of 1.5.



True \ Predicted	0	1	2	3	4
0	16	3	5	1	5
1	1	6	14	5	9
2	2	6	14	16	9
3	1	7	12	10	14
4	2	4	12	12	24

Figure 5.3.7: Confusion matrix of the classified kernels

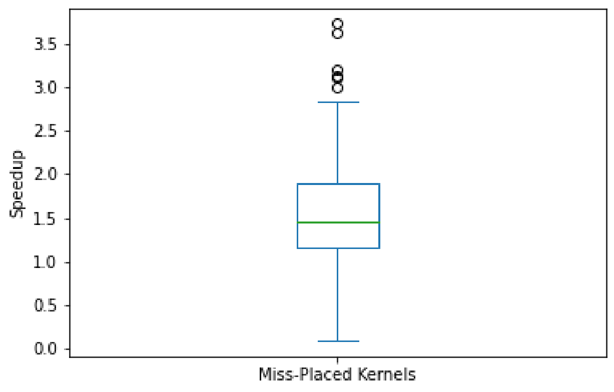


Figure 5.3.8: Speedup of miss-classified kernels as a boxplot

3. **Five-class classification of 1-dimensional block coarsened kernels**

   For this problem, our classes are the same as before and correspond to the optimal block coarsening factor of each kernel, bcf: 1, 2, 4, 8 and 16. Due to the highly imbalance of the dataset we choose to balance it by randomly selecting 100 kernels from the last class that refers to bcf 16, and for the remaining classes we keep all the kernels. The model achieves 33% accuracy.

   Figures 5.3.9a and 5.3.9b show the plots of the loss and accuracy, respectively, of the model. Same as before, the model suffers from overfitting and fails to generalize on unseen data. Here we do not apply any regularization technique.

   Although the low accuracy percentage, the average speedup of the classified kernels is 1.4.

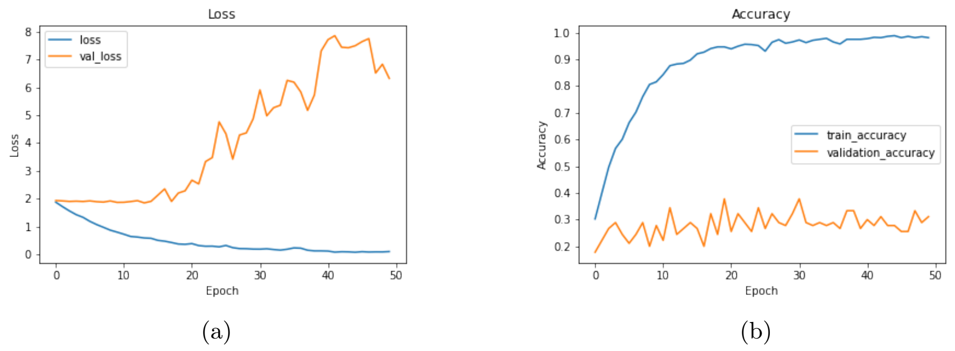


Figure 5.3.9: (a) Train and validation loss of the five-class classification problem for block coarsening (b) Train and validation accuracy of the five-class classification problem for block coarsening

Chapter 6

Conclusions

In this Diploma Thesis, we study ways to incorporate programming code data into Natural Language Processing (NLP). Our objective is to learn how to use machine learning to predict the best optimization heuristics for parallel applications based on their quantitative features.

6.1 Discussion

Our findings in Chapters 4 and 5 lead to conclusions that fall into two categories.

6.1.1 OpenCL Device-Aware Mapping with NLP

In Chapter 4, we extend the work of Cummins et al. [1] by utilizing their dataset of OpenCL kernels, that have been profiled for three hardware devices: one CPU and two GPUs. Two prediction tasks are formed from this dataset: CPU-GPU_1, CPU-GPU_2. We replicate their suggested neural network architecture and propose six different architectures that are evaluated and compared on the same data. On average, five of our suggested algorithms outperform the baseline model, and one of them, namely CNN-BiLSTM-Attention, which incorporates the basic parts of all others, surpasses them on both prediction challenges, achieving a 4% improvement compared to the results of the state-of-the-art model of Cummins et al.

6.1.2 CUDA Thread/Block Coarsening using NLP

In Chapter 5 we develop a general optimization pipeline that incorporates the flow from human-written CUDA code to kernel runtime optimization. Firstly, we perform a series of kernel transformations that include a source-to-source compiler for automatic thread and block coarsening transformations of CUDA kernels, a CUDA rewriter that removes semantically irrelevant information, given by the human factor and our proposed tokenization methodology for the transformed kernels. Afterwards, we perform a profiling to these kernels for a given thread or block coarsening factors, creating prediction classes for our machine learning model. Lastly, we employ our best model from Chapter 4 and, if needed, we suggest an augmentation methodology to improve the model's generalization ability.

We apply the aforementioned pipeline on the LS-CAT dataset from [2] that is comprised by 20251 CUDA kernels. From the results of our methodology we discovered that only the 3477 CUDA (17.17% of the initial dataset) can be used. The other 82.83% comprises of duplicate kernels (58.43%), kernels that raise runtime exception errors (24.17%) and kernels that our system failed to transform, due to version conflicts of our compiler toolchain and the kernel (17.40%).

From the results of the profiling process on the remaining 3477 kernels we observed that block coarsening optimization provides a speedup on the majority of the kernels, and as the coarsening is increased so the speedup of the kernel increases. The optimization of thread coarsening provides a more balanced picture, but again the majority of kernels gain a speedup when coarsened.

When it comes to machine learning we achieve 84% prediction accuracy for the binary classification problem of thread coarsening, with an average speedup of 1.7, and for the five-class classification problem of thread and block coarsening we achieve 42% and 33% accuracy, respectively. Additionally, we observe that for every problem that we studied, the model is susceptible to overfitting. Lastly, in the case of five-classification of thread coarsened kernels we study the speedup of the miss-classified kernels and we find that even when the model does not predict the optimal class, the predicted classes achieve an average speedup of 1.5.

6.2 Future Work

Various expansions and variants of our work may be proposed in the future. While investigating the possibilities of CUDA kernels for Natural Language Processing, we discovered several intriguing future avenues that may conceal an untapped potential. These are summarized below:

- Exploration and evaluation of transformer models, that can be pretrained on large unlabeled C/C++ corpora and be used to predict downstream tasks.
- Additional assessment of the LS-CAT dataset's kernel quality.
- Train on a regression problem for execution time prediction
- Use of Transfer Learning
- Exploration and evaluation of different tokenization techniques on CUDA kernels on the model's predictive capability.
- Extend our pipeline to optimize different performance metrics, besides execution time, such energy consumption.
- Evaluate the impact to the prediction of every textual feature that is fed as input, using local surrogate (LIME).
- Create a large scale dataset of high-quality CUDA kernels.

Bibliography

[1] Chris Cummins et al. "End-to-end deep learning of optimization heuristics". In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2017, pp. 219–232.

[2] Lars Bjertnes, Jacob O Tørring, and Anne C Elster. "LS-CAT: A Large-Scale CUDA AutoTuning Dataset". In: *2021 International Conference on Applied Artificial Intelligence (ICAPAI)*. IEEE. 2021, pp. 1–6.

[3] Alberto Magni, Christophe Dubach, and Michael O'Boyle. "Automatic optimization of thread-coarsening for graphics processors". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 455–466.

[4] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. "Portable mapping of data parallel programs to opencl for heterogeneous systems". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2013, pp. 1–10.

[5] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. "Neural code comprehension: A learnable representation of code semantics". In: *Advances in Neural Information Processing Systems* 31 (2018).

[6] Alexander Brauckmann et al. "Compiler-based graph representations for deep learning models of code". In: *Proceedings of the 29th International Conference on Compiler Construction*. 2020, pp. 201–211.

[7] *Clang: a C language family frontend for LLVM*. URL: https://clang.llvm.org.

[8] Vern Paxson, Will Estes, and John Millaway. "Lexical analysis with flex". In: *University of California* (2007), p. 28.

[9] John Levine and Levine John. *Flex & Bison*. 1st. O'Reilly Media, Inc., 2009. ISBN: 0596155972.

[10] Ritu Agarwal and Vasant Dhar. "Big data, data science, and analytics: The opportunity and challenge for IS research". In: *Information systems research* 25.3 (2014), pp. 443–448.

[11] Amir Gandomi and Murtaza Haider. "Beyond the hype: Big data concepts, methods, and analytics". In: *International journal of information management* 35.2 (2015), pp. 137–144.

[12] Philip HW Leong. "Recent trends in FPGA architectures and applications". In: *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*. IEEE. 2008, pp. 137–141. DOI: 10.1109/DELTA.2008.14.

[13] William Dally et al. *Structured Application-Specific Integrated Circuit (ASIC) Study*. Tech. rep. STANFORD UNIV CA COMPUTER SYSTEMS LAB, 2008.

[14] *Loop unrolling - Wikipedia*. URL: https://en.wikipedia.org/wiki/Loop_unrolling.

[15] Vivek Sarkar. "Optimized Unrolling of Nested Loops". In: *International Journal of Parallel Programming* 29 (2004), pp. 545–581.

[16] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE. 2004, pp. 75–86.

[17] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[18] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[19] Aditya Kanade et al. “Learning and evaluating contextual embedding of source code”. In: (2020), pp. 5110–5121.

[20] Chris Cummins et al. “Synthesizing benchmarks for predictive modeling”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 86–99.

[21] *Exploring the GPU Architecture*. URL: https://core.vmware.com/resource/exploring-gpu-architecture.

[22] *Using CUDA Warp-Level Primitives*. URL: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives.

[23] *An Even Easier Introduction to CUDA*. URL: https://developer.nvidia.com/blog/even-easier-introduction-cuda.

[24] Training Guide. “Introduction to OpenCL™ Programming”. In: (2010).

[25] *OpenCL (Open Computing Language)*. URL: https://en.wikipedia.org/wiki/OpenCL.

[26] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[27] John J Hopfield. “Neurons with graded response have collective computational properties like those of two-state neurons”. In: *Proceedings of the national academy of sciences* 81.10 (1984), pp. 3088–3092.

[28] Teuvo Kohonen. “Self-organized formation of topologically correct feature maps”. In: *Biological cybernetics* 43.1 (1982), pp. 59–69.

[29] *How Much Data Is Created Every Day in 2021?* URL: https://techjury.net/blog/how-much-data-is-created-every-day.

[30] Alireza Taravat et al. “Multilayer perceptron neural networks model for meteosat second generation SEVIRI daytime cloud masking”. In: *Remote Sensing* 7.2 (2015), pp. 1529–1539.

[31] Sanggil Kang and Can Isik. “Partially connected feedforward neural networks structured by input types”. In: *IEEE transactions on neural networks* 16.1 (2005), pp. 175–184.

[32] Tushar Gupta. *Deep Learning: Feedforward Neural Network*. URL: https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7.

[33] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[34] Rich Caruana, Steve Lawrence, and C Giles. “Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping”. In: *Advances in neural information processing systems* 13 (2000).

[35] Yann LeCun et al. “Handwritten digit recognition with a back-propagation network”. In: *Advances in neural information processing systems* 2 (1989).

[36] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. “Understanding of a convolutional neural network”. In: *2017 international conference on engineering and technology (ICET)*. Ieee. 2017, pp. 1–6.

[37] Yoon Kim. “Convolutional Neural Networks for Sentence Classification”. In: (2014). arXiv: 1408.5882 [cs.CL].

[38] Yelong Shen et al. “Learning semantic representations using convolutional neural networks for web search”. In: *Proceedings of the 23rd international conference on world wide web*. 2014, pp. 373–374.

[39] Aliaksei Severyn and Alessandro Moschitti. “Twitter sentiment analysis with deep convolutional neural networks”. In: *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*. 2015, pp. 959–962.

[40] Tomas Mikolov et al. “Recurrent neural network based language model.” In: *Interspeech*. Vol. 2. 3. Makuhari. 2010, pp. 1045–1048.

[41] Tomas Mikolov et al. “Distributed representations of words and phrases and their compositionality”. In: *Advances in neural information processing systems* 26 (2013).

[42] *Understanding LSTM Networks*. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[43] Mike Schuster and Kuldip K Paliwal. “Bidirectional recurrent neural networks”. In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[44] *Neural Networks, Types, and Functional Programming.* URL: http://colah.github.io/posts/2015-09-NN-Types-FP/.

[45] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning Long-Term Dependencies with Gradient Descent is Difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[46] Sepp Hochreiter et al. *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.* 2001.

[47] Sepp Hochreiter. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[48] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks". In: *International conference on machine learning.* PMLR. 2013, pp. 1310–1318.

[49] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[50] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[51] Solveig Badillo et al. "An Introduction to Machine Learning". In: *Clinical Pharmacology & Therapeutics* 107 (Mar. 2020). DOI: 10.1002/cpt.1796.

[52] *Dropout Neural Network Layer In Keras Explained.* URL: https://towardsdatascience.com/machine-learning-part-20-dropout-keras-layers-explained-8c9f6dc4c9ab.

[53] Arik Poznanski and Lior Wolf. "Cnn-n-gram for handwriting word recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 2305–2314.

[54] Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. "The performance of LSTM and BiLSTM in forecasting time series". In: *2019 IEEE International Conference on Big Data (Big Data).* IEEE. 2019, pp. 3285–3292.

[55] Jie Cheng. "Programming Massively Parallel Processors. A Hands-on Approach". In: *Scalable Computing: Practice and Experience* 11.3 (2010), pp. 327–327.

[56] *How to Implement Performance Metrics in CUDA C/C++.* URL: https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc.

[57] John Cheng, Max Grossman, and Ty McKercher. "Professional CUDA C Programming". In: 2014.