ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Τεχνικές συμπίεσης βαθέων νευρωνικών δικτύων σε πολυπύρηνο επεξεργαστικό περιβάλλον

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Παπαγεωργίου

Επιβλέπων: Γεώργιος Γκούμας
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2020

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Τεχνικές Συμπίεσης Βαθέων Νευρωνικών Δικτύων σε πολυπύρηνο επεξεργαστικό περιβάλλον

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Παπαγεωργίου

Επιβλέπων: Γεώργιος Γκούμας
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 22η Οκτωβρίου 2020 .

....	....	....
Γ.Γκούμας	Ν.Κοζύρης	Δ.Πνευματικάτος
Επίκουρος Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π	Καθηγητής Ε.Μ.Π

Αθήνα, Οκτώβριος 2020.

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

Περίληψη

Στη σημερινή εποχή έχει προκύψει η ανάγκη τα βαθιά νευρωνικά δίκτυα (DNN) να χρησιμοποιηθούν σε μια πληθώρα ενσωματωμένων συσκευών λόγω της πολύ καλής τους ακρίβειας. Τα DNN κατάφεραν να κυριαρχήσουν με την πολύ καλή τους ακρίβεια σε πολλούς τομείς της μηχανικής μάθησης, μεταξύ άλλων και στον τομέα της όρασης υπολογιστών. Το μειονέκτημά τους είναι πως έχουν μεγάλες ανάγκες από υπολογιστικούς πόρους. Ως αποτέλεσμα οι υπολογιστικές απαιτήσεις των DNN ξεπερνούν κατά πολύ τις δυνατότητες των συσκευών αυτών σε επίπεδο μνήμης, υπολογιστικής ικανότητας και ενεργειακής αυτονομίας. Έτσι ένα κομμάτι της έρευνας επικεντρώθηκε σε τεχνικές ώστε να μπορούν τα DNN να χρησιμοποιηθούν στις παραπάνω συσκευές.

Η διπλωματική αυτή επικεντρώνεται στη μελέτη της κβαντοποίησης (quantization) με ομαδοποίηση (clustering) ως μεθόδου συμπίεσης των DNN μοντέλων. Αρχικά μελετάμε πώς και σε ποιο βαθμό η κβαντοποίηση των συνελικτικών στρωμάτων επιτυγχάνει συμπίεση διατηρώντας την ακρίβεια των μοντέλων. Έπειτα εξετάζουμε πώς επηρεάζεται η υπολογιστική απόδοσή τους. Βελτιστοποιούμε την απόδοση βασιζόμενοι τόσο τεχνικές που υπάρχουν ήδη στην έρευνα όσο και τεχνικές που προτείνουμε εμείς. Η προσπάθεια αυτή αρχικά επικεντρώνεται στο να κρύψουμε τις καθυστερήσεις που εισάγουν τα μη κανονικά μοτίβα πρόσβασης στη μνήμη που εισάγει η κβαντοποίηση. Για να το πετύχουμε αυτό εξετάζουμε τεχνικές όπως loop optimizations καθώς και Just In Time compilation. Για να αυξήσουμε περαιτέρω την απόδοση, αναπτύσσουμε μια δική μας βιβλιοθήκη Just In Time compilation. Χρησιμοποιώντας τη βιβλιοθήκη αυτή προτείνουμε επίσης μια μέθοδο για την εξάλειψη των μη κανονικών μοτίβων. Τέλος συγκρίνοντας τις υλοποιήσεις μας με σύγχρονες βελτιστοποιημένες συνελίξεις παρατηρούμε πως πετυχαίνουν παρόμοια ή μερικές φορές καλύτερη απόδοση.

Λέξεις-Κλειδιά: Βαθιά Μάθηση, Συνέλιξη, Κβαντοποίηση, k-means, Just in Time compilation

Abstract

Nowadays the need has emerged to deploy deep neural networks (DNN) on a variety of embedded devices due to their high performance. DNNs have dominated, with their state-of-the-art accuracy, a variety of Machine Learning domains and among others, computer vision tasks. Their drawback is however their computational intensity. As a result their computational demands far surpass the capabilities of edge devices in terms of memory, computational power and energy autonomy. Therefore extensive research is being conducted in developing techniques to make DNNs deployable in such devices.

This thesis focuses on studying clustering quantization as a DNN compression technique. We first study the compression achieved with clustering convolution layers while retaining model accuracy. Then we study the effects of clustering on computational performance. We optimize the performance of the DNN models, with methods inspired from existing research as well as with methods we propose. First we focus on improving performance by hiding the latency from irregular memory access patterns that quantization introduces. To achieve that, we investigate loop optimization techniques, as well as Just In Time compilation. To further increase performance, we also develop a Just In Time compilation library. Using the above library we also propose a method to eliminate the irregular access patterns altogether. Finally by comparing our implementations with contemporary optimized convolutions we observe that they achieve similar and sometimes better levels of performance.

Keywords: Deep learning, Convolution, Quantization, k-means, Just in Time compilation

Ευχαριστίες

Αρχικά θέλω να ευχαριστήσω τον Επίκουρο Καθηγητή Ε.Μ.Π. Γεώργιο Γκούμα που μου έδωσε την ευκαιρία να εκπονήσω την παρούσα διπλωματική εργασία στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου.

Θα ήθελα επίσης να ευχαριστήσω τους καθηγητές μου κ. Κοζύρη και κ. Πνευματικάτο για τη διδασκαλία τους και τις γνώσεις που μου προσέφεραν. Ευχαριστώ επίσης την υποψήφια διδάκτορα Αθηνά Ελαφρού για τη συννεισφορά της, την καθοδήγηση, την βοήθεια και τις συμβουλές που μου έδωσε σε όλα τα στάδια της εκπόνησης της διπλωματικής αυτής.

Αυτή η διπλωματική εργασία σηματοδοτεί την ολοκλήρωση των σπουδών μου. Είναι το τέλος ενός θεματικού κύκλου της ζωής μου και αποτελεί έναν επίλογο στις πολλές σελίδες των φοιτητικών μου χρόνων. Γι αυτό θέλω να ευχαριστήσω όλους όσους έκαναν αυτά τα χρόνια αξέχαστα. Όλους εκείνους που κάποτε περπατήσαμε στα ίδια μονοπάτια, τους φίλους και γνωστούς.

Ευχαριστώ όμως ξεχωριστά εκείνα τα παιδιά στα τραπεζάκια, τους ΑΝεξάρτητους Αριστερούς Φοιτητές Ηλεκτρολόγους. Μαζί ανακαλύψαμε τη συντροφικότητα, την αλληλεγγύη, την συλλογικότητα. Μάθαμε την αξία του να αγωνόμαστε για να φτιάχνουμε έναν κόσμο καλύτερο. Για να λεγόμαστε λίγο παραπάνω άνθρωποι.

Ευχαριστώ τους γονείς μου που παντα ήταν δίπλα μου, με βοηθούσαν και με στήριζαν. Χωρίς αυτούς δε θα είχα φτάσει ως εδώ.

Τέλος ευχαριστώ την Έλενα που με στήριξε και άντεξε τη γκρίνια μου όλο αυτό το διάστημα.

Παναγιώτης Παπαγεωργίου,
Οκτώβριος 2020

Contents

List of Figures

List of Algorithms

List of Tables

Κεφάλαιο 1

Εισαγωγή

Στη συνέχεια παρουσιάζουμε τα δύο πειράματα, τη διαδικασία που ακολουθούμε, τα αποτελέσματα καθώς και μελλοντικές επεκτάσεις.

1.1 Συνελικτικά Νευρωνικά Δίκτυα

Τα Συνελικτικά Νευρωνικά Δίκτυα (CNN) είναι μια υποκατηγορία DNN που εξειδικεύονται στην αναγνώριση εικόνας. Όπως τα κανονικά νευρωνικά δίκτυα οι νευρώνες τους λαμβάνουν εισόδους, εκτελούν μια μαθηματική πράξη με τα βάρη και στη συνέχεια εφαρμόζουν μια συνάρτηση ενεργοποίησης. Τα βάρη αλλάζουν με τον ίδιο τρόπο κατά τη διάρκεια της εκπαίδευσης. Η διαφορά είναι ότι τα CNN αποδίδουν καλύτερα με εικόνες ως εισόδους, επειδή οι συνδέσεις μεταξύ δύο στρωμάτων (layers) περιορίζονται σε τοπικές περιοχές. Αυτό μειώνει δραστικά τον αριθμό των παραμέτρων.

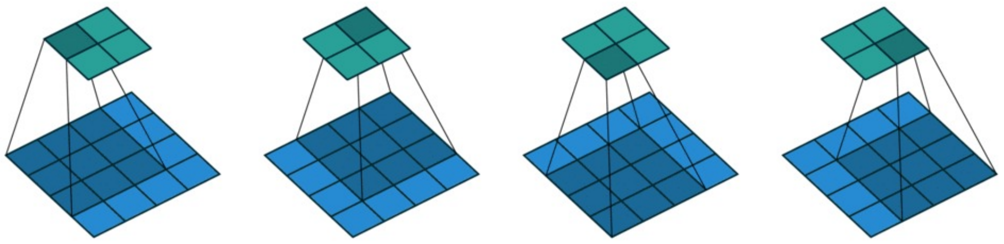
Η θεμελιώδης μαθηματική πράξη των CNN είναι η συνέλιξη. Οι συνελίξεις εφαρμόζονται μεταξύ μιας εισόδου και του πίνακα των βαρών. Τα βάρη κάθε συνελικτικού στρώματος είναι οργανωμένα σε πίνακες που περιέχουν 3-D φίλτρα. Κάθε φίλτρο αποτελείται από 2-D kernels κάθε ένα από τα οποία πολλαπλασιάζεται διαδοχικά με τμήμα του εκάστοτε καναλιού της εικόνας για να δώσει την έξοδο (Σχ. 1.1). Έπειτα οι επί μέρους έξοδοι των καναλιών αθροίζονται και το αποτέλεσμα είναι ένα κανάλι εξόδου. Αυτό επαναλαμβάνεται για όλα τα φίλτρα και έτσι σχηματίζεται η έξοδος. Ο αριθμός των φίλτρων είναι ίσος με τον αριθμό των επιθυμητών καναλιών της εξόδου.

Συγκεντρωτικά η διαδικασία της συνέλιξης φαίνεται στον αλγόριθμο 1. Όπου οι διαστάσεις της εξόδου υπολογίζονται ως εξής:

$$H_o = \frac{(H_i - H_k + 2 * pad)}{stride} + 1 \tag{1.1}$$

$$W_o = \frac{(W_i - W_k + 2 * pad)}{stride} + 1 \tag{1.2}$$

Όπου C_{out}, H_o, W_o είναι το κανάλι, το ύψος και το πλάτος της εξόδου, C_{in} H_i, W_i είναι το κανάλι, το ύψος και το πλάτος της εισόδου και H_k, W_k είναι το ύψος και το πλάτος του kernel. Stride ονομάζεται η απόσταση κατά την οποία μετακινείται το φίλτρο πάνω στην είσοδο σε κάθε επανάληψη του αλγορίθμου. Pad είναι μια τεχνική με την οποία η είσοδος ”γεμίζει” με μηδενικά στις άκρες του ύψους και πλάτους της ώστε η είσοδος και η έξοδος να έχουν τις ίδιες διαστάσεις ύψους πλάτους.



Σχήμα 1.1: Συνέλιξη ενός kernel με μια είσοδο [11]

Algorithm 1 Ο αλγόριθμος της συνέλιξης

```
1: for $i = 0$ to $Cin - 1$ do
2:  for $j = 0$ to $Cout - 1$ do
3:   for $k = 0$ to $H_o - 1$ do
4:    for $l = 0$ to $W_o - 1$ do
5:     for $m = 0$ to $H_k - 1$ do
6:      for $n = 0$ to $W_k - 1$ do
7:       $k_{in} = stride \times k + m - pad$
8:       $l_{in} = stride \times l + n - pad$
9:       $Output[j][k][l] += Input[i][k_{in}][l_{in}] \times Kernel[i][j][m][n]$
```

1.2 Αλγόριθμος k-means

Ο αλγόριθμος k-means είναι ένας επαναληπτικός αλγόριθμος μη επιβλεπόμενης μηχανικής μάθησης. Ομαδοποιεί ένα σύνολο παρατηρήσεων ($X = \{x_1, x_2, ..., x_n\}$) σε διακριτές μη εφαπτόμενες ομάδες (clusters) $C = \{C_1, C_2, ..., C_k\}$. Κάθε cluster αντιπροσωπεύεται από μια τιμή που ονομάζεται centroid. Όλα τα centroids αποτελούν ένα codebook. Ο αλγόριθμός ελαχιστοποιεί το τετράγωνο της Ευκλείδειας απόστασης των στοιχείων κάθε cluster από τη μέση τιμή τους μ_i.

Τα centroids αρχικοποιούνται σε κάποιες τιμές και έπειτα ακολουθούνται εναλλάξ τα εξής βήματα(Σχ. 1.2:

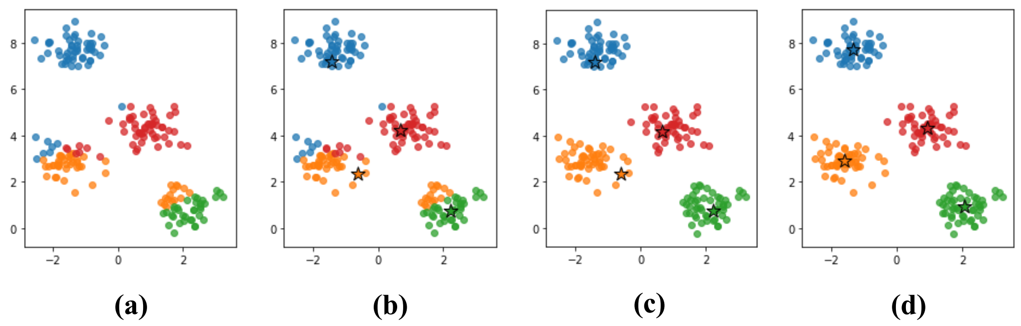
- Κάθε παρατήρηση ανατίθεται στο cluster με το centroid του οποίου έχει το ελάχιστο τετράγωνο της Ευκλείδειας απόστασης.

$$C_i = \{x_k \in X : \| x_k - m_j \|^2 \leq \| x_k - m_i \|^2 j, 1 \leq j \leq k\} \tag{1.3}$$

- Η τιμή του centroid κάθε cluster (m_i) επανυπολογίζεται ως η μέση τιμή όλων των παρατηρήσεων του cluster.

$$m_i = \frac{1}{n_i} \sum_{x_j \in C_i} x_j \tag{1.4}$$

Ο αλγόριθμος συνεχίζει μέχρι οι τιμές των centroids να μην αλλάζουν. Τότε έχει συγκλίνει. Ο αλγόριθμος δεν εγγυάται βέλτιστη λύση γιατί μπορεί να συγκλίνει σε τοπικά ελάχιστα. Γι αυτό και επαναλαμβάνεται πολλές φορές με τυχαίες αρχικοποιήσεις.



Σχήμα 1.2: Τα βήματα του k-means [24]. (a) Κάθε τιμή ανατίθεται σε ένα cluster. (b) Τα centroids επανυπολογίζονται. (c) Κάθε τιμή ανατίθεται σε ένα νέο cluster. (d) Τα centroids υπολογίζονται ξανά.

Κεφάλαιο 2

Κβαντοποίηση

2.1 Κβαντοποίηση

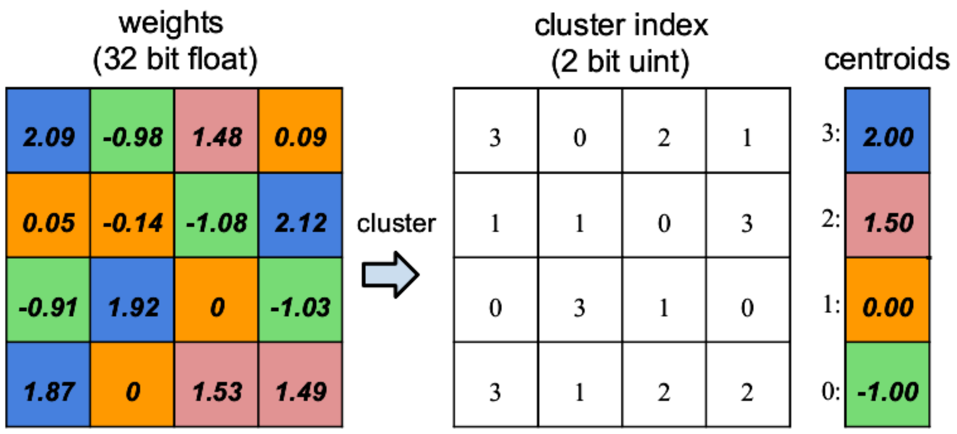
Η κβαντοποίηση είναι μια προσέγγιση για τη συμπίεση DNN. Η κβαντοποίηση χρησιμοποιεί λιγότερα bit για να να αναπαραστήσει τους αριθμούς. Αυτό προκαλεί απόκλιση από της αρχικές τιμές και έχει αντίκτυπο στην ακρίβεια του μοντέλου. Η κβαντοποίηση συνήθως επιτυγχάνεται με δύο τρόπους. Ο πρώτος είναι η μετατροπή των αριθμών κινητής υποδιαστολής σε αριθμούς σταθερής υποδιαστολής. Η θέση της υποδιαστολής καθορίζεται έτσι ώστε να υπάρχει το λιγότερο δυνατό σφάλμα από τις αρχικές τιμές. Ο δεύτερος είναι η ομαδοποίηση των βαρων βάσει κάποιου κριτηρίου. Κάθε ομάδα έχει μια αντιπροσωπευτική τιμή και έτσι τα βάρη καταλήγουν να ειναι λιγότερα.

2.2 Μεθοδολογία Κβαντοποίησης

Πραγματοποιούμε την κβαντοποίηση ώστε να ομαδοποιήσουμε τις τιμές των βαρών σε ομάδες, με κάθε ομάδα να έχει μια αντιπροσωπευτική τιμή. Αφού πραγματοποιηθεί η κβαντοποίηση οι πίνακες των βαρών περιέχουν πλέον indexes στο πίνακα με τις αντιπροσωπευτικές τιμές που ονομάζεται codebook (Σχ. 2.1). Το μέγεθός του εξαρτάται από το πόσα bit επιθυμούμε να καταλαμβάνει σε μέγεθος, το οποίο υπολογίζεται ως 2^{bits}.

Έτσι για να έχουμε πρόσβαση στις τιμές των βαρών πρέπει πρώτα να βρούμε την τιμή του index και βάσει αυτού να επιλέξουμε το κατάλληλο στοιχείο του codebook. Έτσι ο εσωτερικός υπολογισμός στον Αλγόριθμο 1 γίνεται:

$$Output[j][k][l] += Input[i][k_{in}][l_{in}] \times Codebook[Kernel[i][j][m][n]] \tag{2.1}$$



Σχήμα 2.1: Παράδειγμα clustering βαρών με 2-bit codebook [18]

Μετά τη συμπίεση κάθε βάρος είναι ένα integer index στον πίνακα codebook. Έτσι κάθε βάρος έχει μειωθέι σε $\log_2(k)$ (όπου k το μέγεθος του codebook) και ο λόγος συμπίεσης υπολογίζεται ως:

$$r = \frac{n * s}{n * \log_2 k + k * s} \tag{2.2}$$

Όπου n είναι ο αριθμός των βαρών και s = sizeof(float). Για μεγάλο αριθμό βαρών το μέγεθος του codebook μπορεί να θεωρηθεί αμελητέο. Δεδομένου επίσης ότι το μέγεθος του codebook είναι 2^bits η προηγούμενη σχέση γίνεται:

$$r = \frac{s}{bits} \tag{2.3}$$

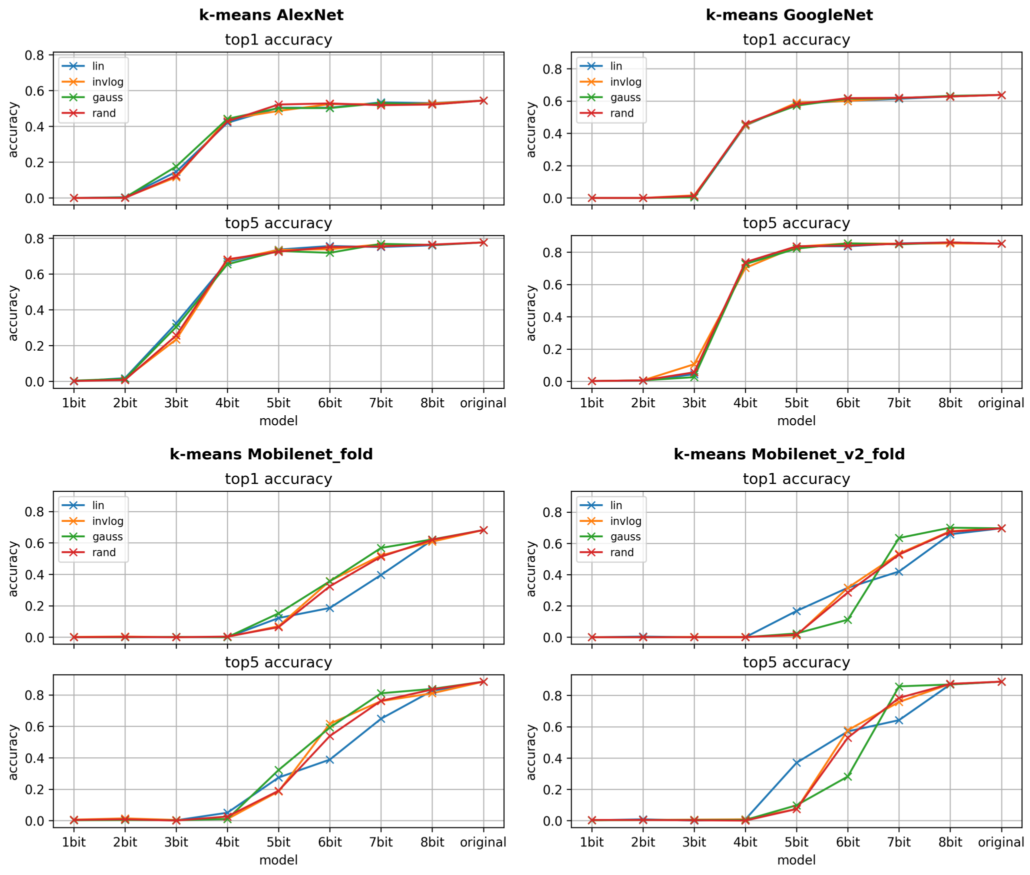
Επομένως για float αριθμούς 32bit ο μεγιστος λόγος συμπίεσης είναι 32 με 1bit codebook.

Για την αξιολόγηση της ακρίβειας επιλέγουμε το Caffe framework [26] για το πείραμά μας. Στο caffe υπάρχουν πολλά μοντέλα ήδη εκπαιδευμένα. Εμείς θα δοκιμάσουμε την κβαντοποίηση στο AlexNet [28], GooleNet [45], Mobilenet [23] και Mobilenet v2 που έχουν εκπαιδευθεί πάνω στο Imagenet [10].

Πραγματοποιούμε τον αλγόριθμο k-means με την βιβλιοθήκη scipy της python [41] και συγκρίνουμε διαφορετικές αρχικοποιήσεις (γραμμική, ομοιόμορφη, αντίστροφη λογαριθμική). Επειδή το caffe δεν υποστηρίζει πράξεις με κβαντοποιημένα βάρη, αντικαθιστούμε τα βάρη με τις τιμές που δείχνουν στο codebook για να πάρουμε μετρήσεις. Δοκιμάζουμε μεγέθη codebook από 1 έως 8 bit.

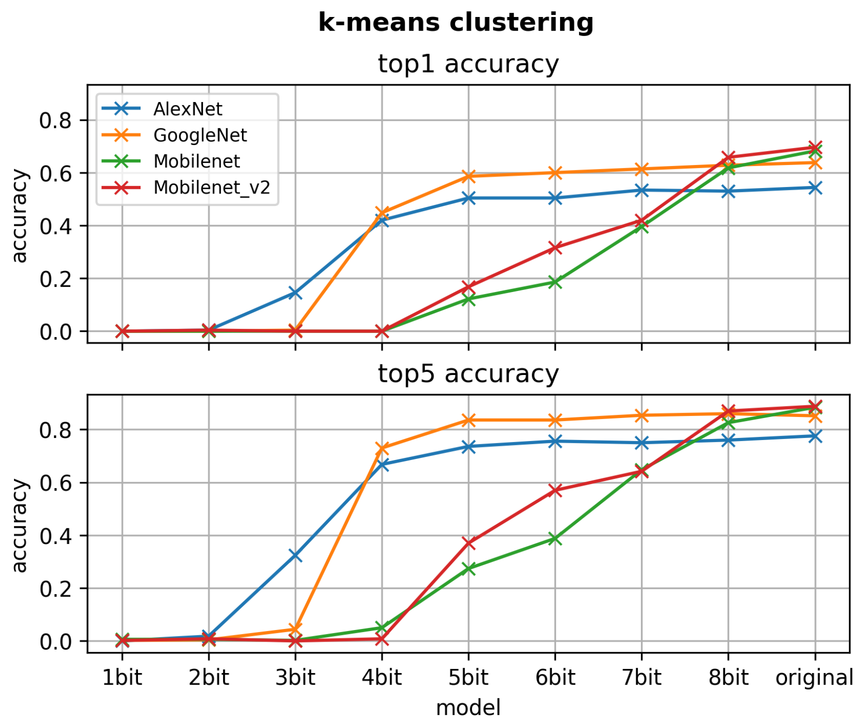
2.3 Αποτελέσματα

Από το Σχ. 2.2 βλέπουμε την επίδραση διαφορετικών αρχικοποιήσεων στην ακρίβεια των προβλέψεων. Παρατηρούμε ότι για τα μεγάλα μοντέλα GoogleNet και AlexNet η επίδραση είναι αμελητέα. Για τα μικρότερα Mobilenets παρατηρούμε διακυμάνσεις αλλά καμία μέθοδος δεν είναι σταθερά πάντα η καλύτερη. Έτσι επιλέγουμε να τα συγκρίνουμε όλα μαζί στην γραμμική αρχικοποίηση.



Σχήμα 2.2: Επίδραση του kmeans με δίαφορες αρχικοποιήσεις στην ακρίβεια των μοντέλων.

Έπειτα στο Σχ. 2.3 συγκρίνουμε την επίδραση του k-means στην ακρίβεια καθώς τα codebook γίνονται μικρότερα. Παρατηρούμε πως τα μεγάλα μοντέλα GoogleNet και AlexNet μπορούν να κβαντοποιηθούν έως 5bit χωρίς να έχουν μεγάλη απόκλιση από την αρχική ακρίβεια. Αυτό συμβαίνει γιατί έχουν πολλά περιττά βάρη. Αντίθετα τα Mobilenet που είναι ήδη αρκετά μιρκότερα καταφέρνουν να κβαντοποιηθούν μέχρι 8bit χωρίς να χάσουν σημαντική ακρίβεια.



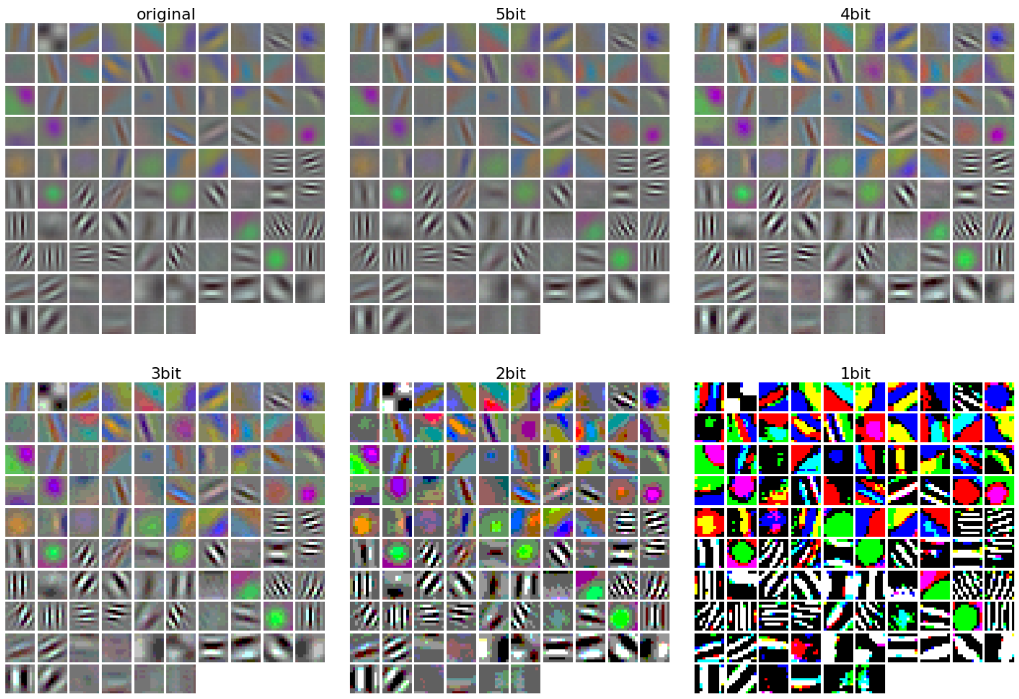
Σχήμα 2.3: Ακρίβεια των clustered μοντέλων καθώς μικραίνει το μέγεθος του codebook

Στα πειράματά μας χρησιμοποιούμε το ίδιο μήκος bit για τα codebooks όλων των layers. Έτσι η εξίσωση 2.3 περιγράφει τον λόγο συμπίεσης για όλο το μοντέλο. Βάσει αυτών υπολογίζουμε στον Πίνακα 2.1 την μέγιστη θεωρητική συμπίεση χωρίς πτώση της ακρίβειας των μοντέλων.

Πίνακας 2.1: Μέγιστη συμπίεση χωρίς πτώση ακρίβειας.

Μοντέλο	Αρχικό μέγεθος	Τελικό μέγεθος	Συμπίεση (Θεωρητική)	Μέγεθος Codebook
AlexNet	233MB	36.4MB	6.4	5bits
GoogleNet	51MB	7.9MB	6.4	5bits
Mobilenet	17MB	4.25MB	4	8bits
Mobilenet_v2	14MB	3.5MB	4	8bits

Πρακτικά η συμπίεση είναι μικρότερη. Αυτό γιατί το μικρότερο κομμάτι μνήμης που μπορεί να διευθυνσιοδοτηθεί είναι 8bits. Μια software λύση θα εισάγει παραπάνω καθυστέρηση στο σύστημα. Μια ουσιαστική λύση σε αυτό το πρόβλημα είναι υποστήριξη από ειδικό hardware (πχ. με fpga η asic) του sub byte indexing. Τέλος στο Σχ. 2.4 μπορούμε να δούμε πώς οι τιμές των βαρών αποκλίνουν σταδιακά από τις αρχικές τους τιμές.



Σχήμα 2.4: Τα βάρη του πρώτου επιπέδου του AlexNet καθώς χρησιμοποιούμε όλο και λιγότερα bit στην κβαντοποίηση.

Κεφάλαιο 3

Βελτιστοποίηση του inference για κβαντοποιημένα μοντέλα

Στο caffe δεν μπορούμε να αξιολογήσουμε τις επιπτώσεις στην απόδοση γιατί δεν υποστηρίζει πράξεις με κβαντοποιημένα βάρη. Έτσι δημιουργούμε κάποιες υλοποιήσεις του αλγορίθμου της συνέλιξης σε C++ για να τις μελετήσουμε. Το Clustering προκαλεί μη συνεχείς και μη κανονικές προσβάσεις στην μνήμη καθώς κάθε πρόσβαση είναι μία έμμεση αναφορά στην πραγματική τιμή του βάρους. Πρώτα βρίσκεται η τιμή που περιέχει το βάρος (που είναι index) και έπειτα βάση αυτής η κατάλληλη τιμή στο codebook. Αυτό καθιστά την πρόσβαση μνήμης τυχαία και είναι κακό τόσο για την επίδοση της cache όσο και για την δυνατότητα vectorization. Μελετάμε και προτείνουμε κάποιες νέες βελτιστοποιήσεις για να κρύψουμε ή να απαλείψουμε την καθυστέρηση που εισάγει το clustering.

3.1 Padding

Το padding μπορεί είτε να υλοποιηθεί όπως στον Αλγόριθμο 2, όπου με την συνθήκη ”if” ελέγχεται εάν κάθε index είναι μέσα στα όρια του πίνακα εισόδου, είτε με πραγματική αλλαγή στις διαστάσεις της εισόδου όπως στον Αλγόριθμο 3.

Algorithm 2 Συνέλιξη με έλεγχο συνοριακών τιμών

```
1: for $i = 0$ to $Cin - 1$ do
2:  for $j = 0$ to $Cout - 1$ do
3:   for $k = 0$ to $H_o - 1$ do
4:    for $l = 0$ to $W_o - 1$ do
5:     for $m = 0$ to $H_k - 1$ do
6:      for $n = 0$ to $W_k - 1$ do
7:       if $0 \leq k_{in} < H_i$ and $0 \leq l_{in} < W_i$ then
8:        $k_{in} = stride \times k + m - pad$
9:        $l_{in} = stride \times l + n - pad$
10:       $Output[j][k][l] += Input[i][k_{in}][l_{in}] \times Kernel[i][j][m][n]$
```

Algorithm 3 Συνέλιξη με αλλαγή διαστάσεων της εισόδου

```
1: for $i = 0$ to $Cin - 1$ do
2:  for $j = 0$ to $Cout - 1$ do
3:   for $k = 0$ to $H_o - 1$ do
4:    for $l = 0$ to $W_o - 1$ do
5:     for $m = 0$ to $H_k - 1$ do
6:      for $n = 0$ to $W_k - 1$ do
7:       $k_{in} = stride \times k + m$
8:       $l_{in} = stride \times l + n$
9:       $Output[j][k][l] += Input_{pad}[i][k_{in}][l_{in}] \times Kernel[i][j][m][n]$
```

3.2 Loop order

Ο αλγόριθμος της συνέλιξης δεν είναι πολύ αποδοτικός. Γι αυτό και τα δεδομένα της τροποποιούνται έτσι ώστε να μπορεί να υπολογιστεί με πολύ αποδοτικούς αλγορίθμους πολλαπλασιασμού πίνακα με πίνακα (GEMM). Αυτό για να επιτευχθεί πρέπει τα δεδομένα να αλλάξουν μορφή (μέθοδος im2col) κάτι που χρειάζεται παραπάνω μνήμη. Επίσης η απόδοση των GEMM με τις συνελίξεις δεν είναι η μέγιστη δυνατή λόγω των διαστάσεων των πινάκων. Σε αυτό το πρόβλημα μια λύση είναι η βελτιστοποίηση της αρχικής συνέλιξης τόσο ώστε η GEMM να μην είναι αναγκαία [56]. Αυτό το επιτυγχάνουν αλλάζοντας τη σειρά των loop στον αλγόριθμο, καθώς και τη σειρά των διαστάσεων των πινάκων. Αυτό φαίνεται στον Αλγόριθμο 4. Αυτή η σειρά εξασφαλίζει καλή επίδοση της cache καθώς έχουμε πολλές προσβάσεις σε συνεχείς θέσεις μνήμης. Επίσης δεν υπάρχουν εξαρτήσεις στο τελευταίο loop και επομένος μπορούν να χρησιμοποιηθούν Fused Multiply-Add(FMA) εντολές που κάνουν έναν πολλαπλασιασμό και μια πρόσθεση μαζί, και SIMD vectors που πραγματοποιούν πολλές εντολές ταυτόχρονα.

Algorithm 4 Συνέλιξη με αλλαγή της σειράς των loop

```
1: for $k = 0$ to $H_o - 1$ do
2:   for $m = 0$ to $H_k - 1$ do
3:     for $n = 0$ to $W_k - 1$ do
4:       for $i = 0$ to $Cin - 1$ do
5:         for $l = 0$ to $W_o - 1$ do
6:           for $j = 0$ to $Cout - 1$ do
7:             $k_{in} = stride \times k + m$
8:             $l_{in} = stride \times l + n$
9:             $Output[k][l[j] += Input[i][k_{in}][l_{in}] \times Kernel[m][n][i][j]$
```

3.3 Blocking

Όπως αναφέρθηκε πριν η συνέλιξη δεν είναι τόσο αποδοτική. Μια ακόμα πρόταση για την βελτίωσή της είναι η εφαρμογή blocking στα loops της ώστε να έχουμε αποδοτική επαναχρησιμοποίηση των δεδομένων και βέλτιστη χρήση των FMA SIMD εντολών [14] [56].

Βασιζόμενοι στα παραπάνω αναπτύξαμε τρεις εκδοχές blocking (αλγόριθμοι 12, 13 και 14). Η απόδοση μπορεί να βελτιωθεί αρχικά καθώς είναι εφικτό να καταφέρουμε να χωράει το working set στην cache. Για να κρύψουμε την καθυστέρηση L_{fma} των εντολών vector FMA με N_{vec} στοιχεία, πρέπει σε κάθε κύκλο να είναι διαθέσιμα και να εισάγονται τουλάχιστον $L_{fma}N_{vec}$ ανεξάρτητα στοιχεία σε κάθε FMA unit. Παράλληλα πρέπει να εξασφαλίσουμε ότι τα διαθέσιμα στοιχεία δεν θα είναι τόσα πολλά ώστε να γεμίσουν όλοι οι SIMD καταχωρητές, για να αποφύγουμε το register spilling στη μνήμη. Βασιζόμενοι στα παραπάνω δοκιμάσαμε διάφορα μεγέθη και τελικά το κατάλληλο blocksize που πληρεί τα παραπάνω και πειραματικά έχει την καλύτερη απόδοση φαίνεται στον Πίνακα 3.1

Algorithm 5 Συνέλιξη με blocking στις χωρικές διαστάσεις

```
1: $H_{ob} = H_o / Blocksize_H$
2: $W_{ob} = W_o / Blocksize_W$
3: for $i = 0$ to $Cin - 1$ do
4:   for $j = 0$ to $Cout - 1$ do
5:     for $k_b = 0$ to $H_o - 1$ step $H_{ob}$ do
6:       for $l_b = 0$ to $W_o - 1$ step $W_{ob}$ do
7:         for $m = 0$ to $H_k - 1$ do
8:           for $n = 0$ to $W_k - 1$ do
9:             for $k = k_b$ to $H_{ob}$ do
10:              for $l = l_b$ to $W_{ob}$ do
11:                $k_{in} = stride \times k + m$
12:                $l_{in} = stride \times l + n$
13:                $Output[j][k][l] += Input_{pad}[i][k_{in}][l_{in}] \times Kernel[i][j][m][n]$
```

Πίνακας 3.1: Μεγέθη block

	H_b	W_b	Cin_b	$Cout_b$
Αλγ. 5	16	16	-	-
Αλγ. 6	8	16	N_{vec}	N_{vec}
Αλγ. 7	-	16	32	8*N_{vec}

Algorithm 6 Συνέλιξη με blocking στις χωρικές διαστάσεις και στα channels

```
1: $Cin_b = Cin/N_{vec}$
2: $Cout_b = Cout/N_{vec}$
3: $H_{ob} = H_o/Blocksize_H$
4: $W_{ob} = W_o/Blocksize_W$
5: for $i_b = 0$ to $Cin - 1$ step $Cin_b$ do
6:  for $j_b = 0$ to $Cout - 1$ step $Cout_b$ do
7:   for $k_b = 0$ to $H_o - 1$ step $H_{ob}$ do
8:    for $l_b = 0$ to $W_o - 1$ step $W_{ob}$ do
9:     for $m = 0$ to $H_k - 1$ do
10:      for $= 0$ to $W_k - 1$ do
11:       for $i = i_b$ to $N_{vec}$ do
12:        for $j = j_b$ to $N_{vec}$ do
13:         for $k = k_b$ to $H_{ob}$ do
14:          for $l = l_b$ to $W_{ob}$ do
15:           $k_{in} = stride \times k + m$
16:           $l_{in} = stride \times l + n$
17:           $Output[j][k][l] += Input_{pad}[i][k_{in}][l_{in}] \times Kernel[i][j][m][n]$
```

Algorithm 7 Συνέλιξη με αλλαγή της σειράς των loop και blocking

```
1: $Cin_b = Cin/Blocksize_{Ci}$
2: $Cout_b = Cout/Blocksize_{Co}$
3: $W_{ob} = W_o/Blocksize_W$
4: for $j_b = 0$ to $Cout - 1$ step $Cout_b$ do
5:  for $i_b = 0$ to $Cin - 1$ step $Cin_b$ do
6:   for $k = 0$ to $H_o - 1$ do
7:    for $l = 0$ to $W_o - 1$ step $W_{ob}$ do
8:     for $m = 0$ to $H_k - 1$ do
9:      for $n = 0$ to $W_k - 1$ do
10:       for $i = i_b$ to $Cin_b$ do
11:        for $l = l_b$ to $W_{ob}$ do
12:         for $j = j_b$ to $Cout_b$ do
13:          $k_{in} = stride \times k + m$
14:          $l_{in} = stride \times l + n$
15:          $Output[k][l[j] += Input[i][k_{in}][l_{in}] \times Kernel[m][n][i][j]$
```

3.4 Just In Time Compilation

Υπάρχουν περιπτώσεις που η απόδοση του προγράμματος εξαρτάται από παραμέτρους που γίνονται γνωστές αφού το πρόγραμμα ξεκινήσει να τρέχει (στο runtime). Για να μπορέσουμε να εκμεταλλευτούμε ότι γνωρίζουμε αυτές τις παραμέτρους και να βελτιστοποιήσουμε το πρόγραμμά μας χρησιμοποιούμε Just In Time (JIT) compilation. Το JIT compilation κάνει compile κομμάτια του προγράμματος, παράγει και εκτελεί εξειδικευμένο κώδικα βέλτιστο για τις συγκεκριμένες παραμέτρους.

Η συνέλιξη μπορεί να ωφεληθεί από την παραπάνω τεχνική καθώς μόλις γνωρίζουμε τις διαστάσεις και τις τιμές του πίνακα των βαρών καθώς και τα codebook αυτά δεν αλλάζουν καθ' όλη τη διάρκεια της εκτέλεσης και επομένως ο κώδικας μπορεί να βελτιστοποιηθεί. Χρησιμοποιούμε δύο ήδη υπάρχοντα framework για JIT και έπειτα επιχειρούμε μια δικιά μας υλοποίηση.

Χρησιμοποιούμε αρχικά το asmjit framework [27], το οποίο εκτελεί JIT με κώδικα x86-64 assembly. Βασιζόμενοι σε παρόμοια έρευνα με στόχο τους αραιούς πίνακες, [53] υλοποιούμε την συνέλιξη βγάζοντας κοινό παράγοντα τα codebook. Κάθε στοιχείο της συνέλιξης μπορεί να υπολογιστεί από τον εξής τύπο.

$$out[j][k][l] = \sum_{i=1}^{C_{in}} \sum_{m=1}^{W_k} \sum_{n=1}^{H_k} codebook[kernel[i][j][m][n]] \times Input[i][k_{in}][l_{in}] \tag{3.1}$$

Εάν ανοίξουμε τα αθροίσματα έχουμε το εξής:

$$\begin{aligned} out[j][k][l] = {} & codebook[0] \times Input[1] + codebook[3] \times Input[2] + \\ & + codebook[0] \times Input[6] \\ & + codebook[2] \times Input[10] \\ & + ... \end{aligned} \tag{3.2}$$

Έπειτα βγάζοντας κάθε ένα codebook κοίνο παράγοντα:

$$\begin{aligned} out[j][k][l] = {} & codebook[0] \times (Input[1] + Input[6] + ...) \\ & + codebook[1](Input[9] + ..) \\ & + ... \\ & + codebook[2^{bits} - 1](Input[14] + ..) \end{aligned} \tag{3.3}$$

Τέλος για να μετατρέψουμε τον παραπάνω τύπο πάλι σε άθροισμα χρειάζεται να ξέρουμε ποια στοιχεία της εισόδου πολλαπλασιάζονται με ποιο codebook, ώστε να τα αθροίσουμε πρώτα όλα μαζί και να γλιτώσουμε πολλαπλασιασμούς. Αν αυτό υλοποιηθεί από έναν πίνακα m που περιέχει λίστες με τα στοιχεία που θα πολλαπλασιαστούν με κάθε codebook προκύπτει ο εξής τύπος.

$$out[j][k][l] = \sum_{i=0}^{2^{bits}-1} \left(codebook[i] \times \sum_{j=m[i].start}^{m[i].end} Input[j] \right) \tag{3.4}$$

Εμείς δημιουργούμε τέτοιο κώδικα στο runtime:

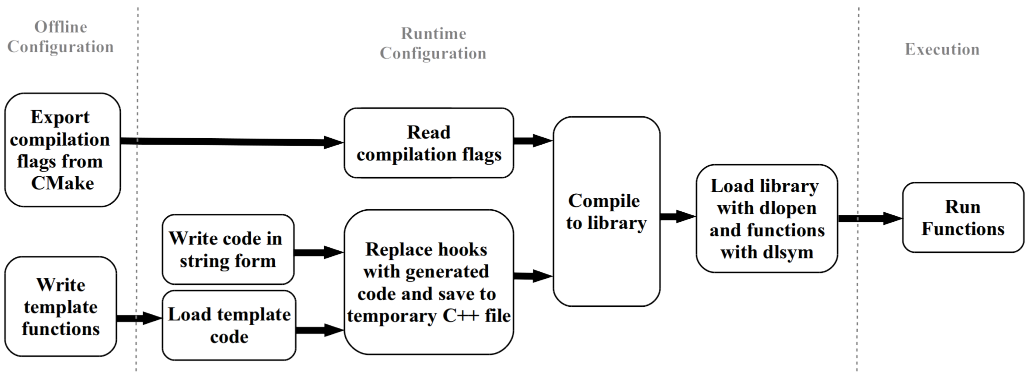
```
//add every input mapped to codebook[0]
xorps xmm1, xmm1 //sum=0
vaddss xmm1, xmm1, [rcx+4] //sum+=input[1]
vaddss xmm1, xmm1, [rcx+32] //sum+=input[8]
...
vfmadd231ss xmm2, xmm1, [rdx] //out[index]+=codebook[0]*sum
//add every input mapped to codebook[1]
xorps xmm1, xmm1 //sum=0
vaddss xmm1, xmm1, [rcx+8] //sum+=input[2]
vaddss xmm1, xmm1, [rcx+52] //sum+=input[13]
...
//perform scalar fused multiply add
vfmadd231ss xmm2, xmm1,[rdx+4] //out[index]+=codebook[1]*sum
...
//repeat for all codebook elements
```

Στη συνέχεια χρησιμοποιούμε το easyjit[3] framework το οποίο χρησιμοποιεί τον compiler clang ώστε να βελτιστοποιήσει μια συνάρτηση καθώς τρέχει το πρόγραμμα. Το μόνο που χρειάζεται είναι να του παρέχουμε τις παραμέτρους της συνάρτησης που θέλουμε να βελτιστοποιήσει. Αυτό μας επιστρέφει έναν function pointer στην βελτιστοποιημένη συνάρτηση από τον οποίο μπορούμε να την καλέσουμε με τις υπόλοιπες παραμέτρους.

```
    auto conv_opt = easy::jit(conv, c_in, rows, cols, \
                    c_out,kRows,  kCols, pad, stride, _1 , \
                    &codebook, &kernel, _2);
    conv_opt(&in,&out);
```

Αυτή η υλοποίηση είναι πολύ εύκολη και δεν χρειάζεται πολλές γνώσεις πάνω στο θέμα για να χρησιμοποιηθεί.

Τέλος επειδή οι δύο αυτές υλοποιήσεις δεν κάλυπταν τις ανάγκες μας σε επίπεδο ευκολίας χρήσης αλλα και ελευθερία επεξεργασίας του κώδικα υλοποιήσαμε μια βιβλιοθήκη που δημιουργεί, κάνει compile, και εκτελεί συναρτήσεις στο runtime. Αρχικά τα compiler options εξάγονται κατα το build. Έπειτα γράφουμε κώδικα C++ στο πρόγραμμά μας σε μορφή string ή/και χρησιμοποιούμε αποθηκευμένα templates για συντομία. Τα templates έχουν hooks τα οποία μπορούμε να αντικαταστήσουμε με stings στο runtime. Έπειτα το πρόγραμμα αποθηκεύεται σε ένα προσωρινό αρχείο για να γίνει compile. Τέλος το κάνουμε load με τη συνάρτηση dlopen και χρησιμοποιούμε τις συναρτήσεις του με την dlsym. Η διαδικασία αυτή φαίνεται σχηματικά στο Σχ. 3.1.


Offline Configuration
Runtime Configuration
Execution
Export compilation flags from CMake
Read compilation flags
Write template functions
Write code in string form
Load template code
Replace hooks with generated code and save to temporary C++ file
Compile to library
Load library with dlopen and functions with dlsym
Run Functions

Σχήμα 3.1: Δημιουργία- Compile- Εκτέλεση κώδικα

3.5 Loop unrolling

Το loop unrolling μπορεί να βελτιώσει την απόδοση της συνέλιξης καθώς ελέγχεται πιο σπάνια η συνθήκη εξόδου από το loop. Επίσης ο compiler μπορεί να βελτιστοποιήσει καλύτερα τον κώδικα. Ο τρόπος δημιουργίας κώδικα στο runtime που αναφέραμε παραπάνω μας επιτρέπει να κάνουμε unroll όλο το kernel της συνάρτησης. Επίσης χρησιμοποιούμε ενδιάμεσες μεταβλητές ως παράλληλους accumulators κάτι που επιτρέπει στον compiler να κάνει ακόμα περισσότερες βελτιστοποιήσεις.

3.6 Unique filters

Μετά τον αλγόριθμο k-means είναι πιθανό πολλά kernels να επαναλαμβάνονται. Αυτό μπορούμε να το εκμεταλλευτούμε και να δημιουργήσουμε μία συνάρτηση για κάθε kernel. Καθώς μπορούμε να δημιουργήσουμε κώδικα στο runtime μπορούμε να τυπώσουμε τις τιμές των codebooks και έτσι να μην υπάρχουν πλέον έμμεσες αναφορές στα βάρη. Ακόμη μπορούμε να εκτελέσουμε τις πράξεις βγάζοντας κοινό παράγοντα στη λογική της εξίσωσης 3.4. Συγκρίνουμε τα αποτελέσματα με υλοποιήσεις με κανονικούς πολλαπλασιασμούς.

3.7 Αποτελέσματα

Φτιάξαμε benchmarks πάνω στα οποία θα αξιολογήσουμε τις υλοποιήσεις. Θα λάβουμε τα αποτελέσματα στα μηχανήματα του Πίνακα 3.2. Τα benchmarks περιγράφονται στους Πίνακες 3.4 3.5 και 3.6. 'Ως μετρικές θα χρησιμοποιήσουμε τον χρόνο εκτέλεσης, το Speedup και τα FLOPS. Φροντίσαμε επίσης να ενεργοποιήσουμε το vectorization αλλά και τα unsafe math optimizations.

Πίνακας 3.2: Machine specifications.

Type	Computer	OS	CPU	RAM
Server		Ubuntu 18.04	2x Xeon Gold 5218	314GB
Desktop	Ideapad 510-15ikb	Ubuntu 16.04	i7-7500U	8GB
Edge	Raspberry Pi 3 B+	Ubuntu 18.04	Cortex A53	1GB

Πίνακας 3.3: CPU specifications.

CPU	arch	speed	cores/ threads	L1,L2,L3	misc
Xeon Gold 5218	x86-64	3.9GHz	16-32	32K, 1M, 22M	avx-512
i7-7500U	x86-64	3.5GHz	2-4	32K, 256K, 4M	avx2
Arm Cortex A53	aarch64	1.4GHz	4-4	16K, 512K	neon

Πίνακας 3.4: Benchmark με τις συνελίξεις του AlexNet.

	C_{in}	H_{in}	W_{in}	C_{out}	H_k	W_k	stride	pad
conv1	3	227	227	96	11	11	4	no
conv2	96	27	27	256	5	5	1	yes
conv3	256	13	13	384	3	3	1	yes
conv4	384	13	13	384	3	3	1	yes
conv5	384	13	13	256	3	3	1	yes

Πίνακας 3.5: Benchmark με μεγάλες συνελίξεις.

	C_{in}	H_{in}	W_{in}	C_{out}	H_k	W_k	stride	pad
bench1	256	128	128	384	3	3	1	yes
bench2	256	128	128	512	3	3	1	yes
bench3	384	128	128	384	3	3	1	yes
bench4	384	128	128	512	3	3	1	yes
bench5	384	112	112	512	3	3	1	yes
bench6	512	56	56	1024	3	3	1	yes
bench7	512	28	28	512	3	3	1	yes

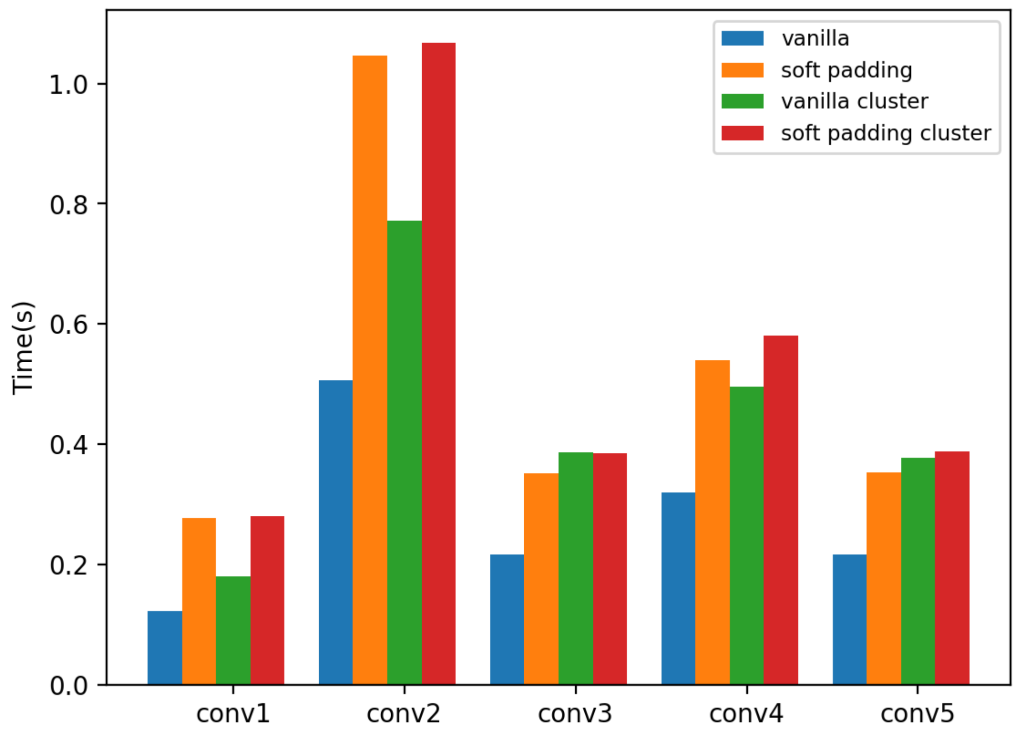
Πίνακας 3.6: Benchmark που στοχεύει το τελευταίο επίπεδο cache στα μηχανήματα του Πίνακα 3.2

	C_{in}	H_{in}	W_{in}	C_{out}	H_k	W_k	stride	pad	memory
Server	512	32	32	512	3	3	1	yes	13MB
(22MB)	512	32	32	1024	3	3	1	yes	24MB
Desktop	128	32	32	256	3	3	1	yes	2.5MB
(4MB)	128	32	32	768	3	3	1	yes	7MB
Edge	64	16	16	128	3	3	1	yes	481KB
(512KB)	64	16	16	256	3	3	1	yes	609KB

Πίνακας 3.7: Υλοποιήσεις και ονόματα στα διαγράμματα

Implementation	Name	Implementation	Name
Vanilla convolution	v	Block reordered Cluster	bl-re-cl
Vanilla convolution (cluster)	v-cl	Asmjit	asmjit
Soft padding	sft	Asmjit Vectorized	asmjit-v
Soft padding cluster	sft-cl	Unique filters	unique
Reordered	re	Unique filters hardcoded	hard
Reordered cluster	re-cl	Unique filters iterator	iter
Block Spatial	bl-sp	Unique filters loop	loop
Block spatial cluster	bl-sp-cl	Codegenv vanilla	cgen
Block all dimensions	bl-a	Codegen unroll	cgen-unr
Block all dimensions cluster	bl-a-cl	BLAS im2col	im2col
Block reordered	bl-re		

Το padding φαίνεται να επηρρεάζει την απόδοση. Από το Σχ. 3.2. Η μέθοδος του Αλγ. 2 φαίνεται να εισάγει περισσότερες καθυστερήσεις καθώς ελέγχει σε κάθε επανάληψη τις συνοριακές τιμές. Επομένως θα χρησιμοποιήσουμε τον Αλγ. 3 απο δω και στο εξής.

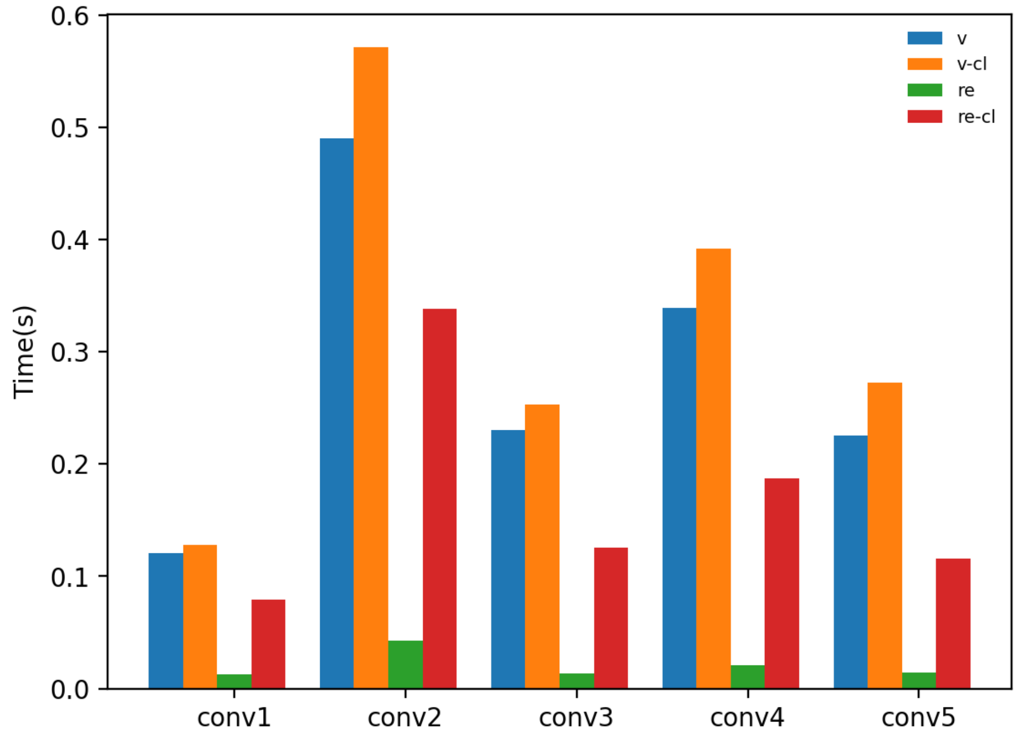


Σχήμα 3.2: Σύγκριση τεχνικών padding στην απλή συνέλιξη και στη συνέλιξη με clustering (Benchmark 8.3 στο μηχάνημα Desktop)

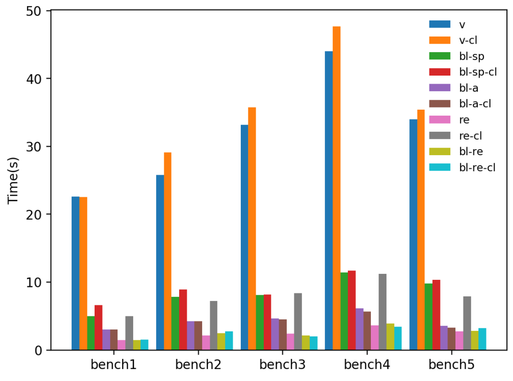
Παρατηρούμε στη συνέχεια πως ο Αλγ. 4 βελτιώνει την συνέλιξη. Βέβαια όπως μπορούμε να δούμε στο Σχ. 3.3 η cluster εκδοχή δεν βελτιώνεται τόσο, κάτι που μπορεί να αποδοθεί στις ακανόνιστες προσβάσεις μνήμης που προκαλεί το clustering.

Στη συνέχεια αξιολογούμε τις blocking υλοποιήσει όπου παρατηρούμε πως στα μηχανήματα Server και Desktop η απόδοση βελτιώνεται αισθητά στις blocking εκδοχές. Το blocking απλά στις χωρικές διαστάσεις φέρνει μια μέτρια βελτίωση αλλά οι Αλγόριθμοι 6 και 7 επιφέρουν πολύ μεγαλύτερη. Αυτό είναι αναμενόμενο καθώς το blocking στα channels είναι που επιτρέπει την καλύτερη αξιοποίηση των FMA units καθώς έχουμε περισσότερη ελευθερία στην επιλογή block size. Η βελτίωση είναι τόσο σημαντική που η cluster και η non cluster εκδοχές έχουν την ίδια υψηλή απόδοση. Αυτό συμβαίνει ακόμα και στον Αλγ. 7 πράγμα που δεν συνέβαινε στον Αλγ. 4 όπως είδαμε παραπάνω.

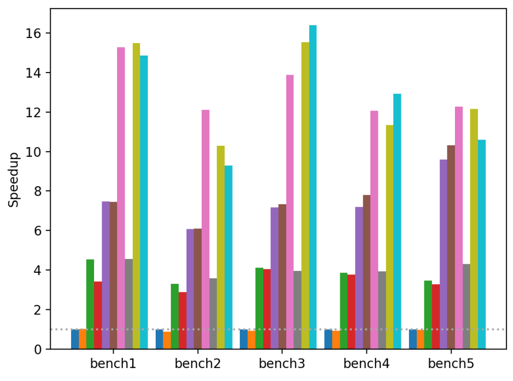
Στο μηχάνημα Edge ναι μεν το blocking βελτιώνει την απόδοση αλλά όχι στον ίδιο βαθμό με τα προηγούμενα. Παρατηρούμε ότι σε αντίθεση με παραπάνω οι cluster και non cluster υλοποιήσεις δεν είναι κοντά σε επίπεδο απόδοσης. Αυτό μπορεί να οφείλεται στις διαφορές στον compiler καθώς και στο μικρότερο vector length που παρουσιάζει το μηχάνημα. Παρατηρούμε επίσης από τα παραπάνω ότι καθώς αυξάνεται το vector length αυξάνεται και η απόδοση των blocking υλοποιήσεων.



Σχήμα 3.3: Reordered εκδοχή συνέλιξης με και χωρίς clustering στο Desktop και Benchmark 3.4.

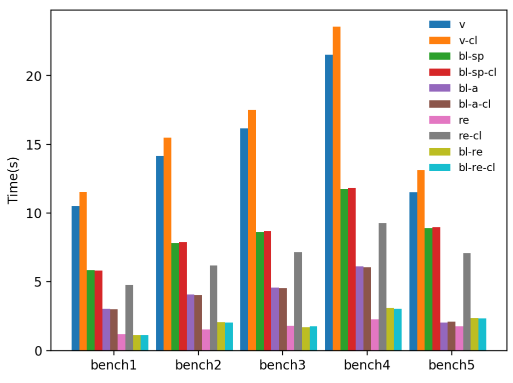


(a) Time

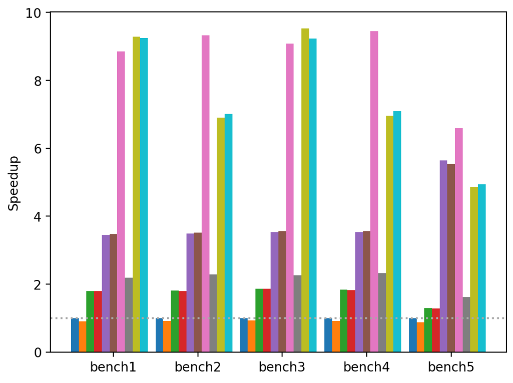


(b) Speedup

Σχήμα 3.4: Επίδοση των blocking υλοποιήσεων και των αντίστοιχων μη blocking (Πίνακας 3.5 στο μηχάνημα Desktop)

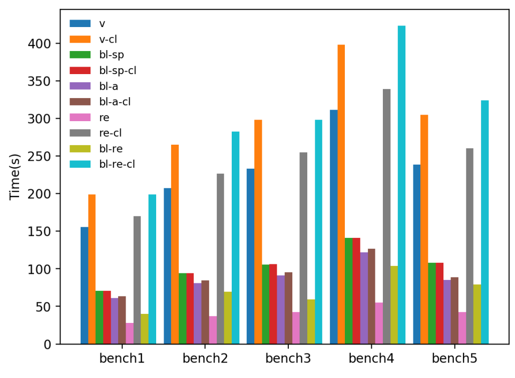


(a) Time

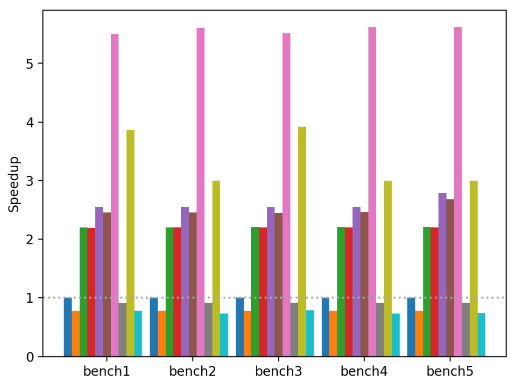


(b) Speedup

Σχήμα 3.5: Επίδοση των blocking υλοποιήσεων και των αντίστοιχων μη blocking (Πίνακας 3.5 στο μηχάνημα Server)



(a) Time

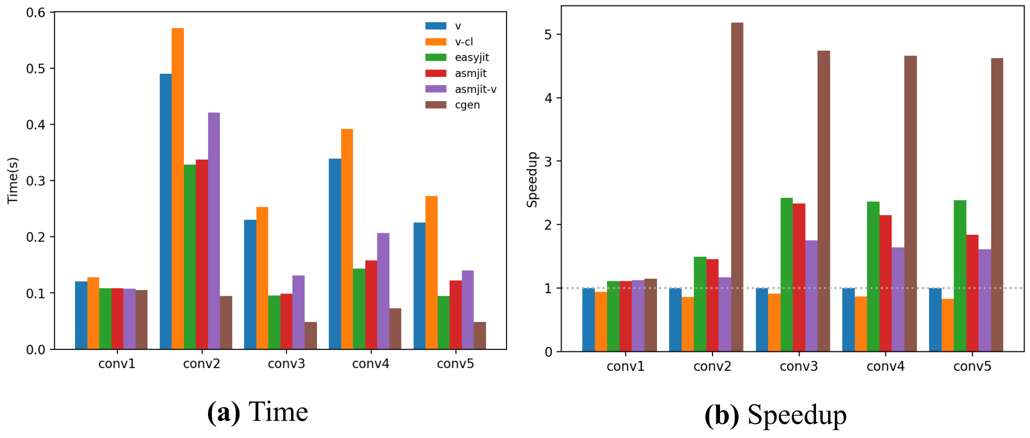


(b) Speedup

Σχήμα 3.6: Επίδοση των blocking υλοποιήσεων και των αντίστοιχων μη blocking (Πίνακας 3.5 στο μηχάνημα Edge)

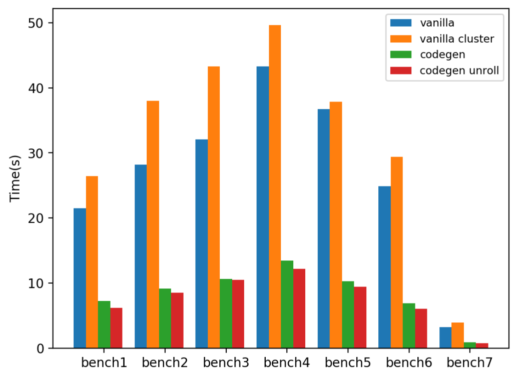
Στη συνέχεια αξιολογούμε και συγκρίνουμε τις JIT υλοποιήσεις με την αρχική συνέλιξη αλλά και μεταξύ τους. Από το Σχ. 3.7 παρατηρούμε πως η υλοποίηση του asmjit έχει περιορισμένη χρησιμότητα στην περίπτωσή μας. Επιτυγχάνει αποτελέσματα συγκρίσιμα με το easyjit αλλά με πολύ κόπο από τη πλευρά του προγραμματιστή. Παραπάνω βελτιστοποιήσεις και vectorization εξαρτώνται επίσης από τις ικανότητες του χρήστη. Τα παραπάνω σε συνδυασμό με την μεγάλη χρήση RAM για μεγάλα benchmarks δεν τη κάνει κατάλληλη για το πρόβλημά μας. Επίσης δεν φαίνεται να κερδίζουμε κάτι σε άποψη χρόνου από τους κοινούς παράγοντες της Εξ. 3.4. Αυτό μπορεί να ερμηνευθεί λόγω της διαφορετικής αρχιτεκτονικής του προβλήματός μας αλλά και από την χρήση από εμάς, vectorizes εντολών, όπου ο πολλαπλασιασμός και η πρόσθεση έχουν την ίδια καθυστέρηση. Το Easyjit αν και απλούστερο στη χρήση έχει τα μειονεκτήματά του. Μπορεί να βελτιστοποιήσει συναρτήσεις που έχουν αποκλειστικά C++ fundamental data types και δεν έχουμε έλεγχο στις βελτιστοποιήσεις.

Η απλή εκδοχή της δημιουργίας κώδικα (codegen) πρακτικά πραγματοποιεί ότι και το easyjit. Έχει όμως καλύτερη απόδοση γιατί μπορούμε να επέμβουμε πιο άμεσα στον κώδικα και να επιλέξουμε μόνοι μας τα κατάλληλα compilation flags. Επομένως καταλήγουμε στο ότι θα χρησιμοποιήσουμε την δική μας βιβλιοθήκη στις επόμενες συγκρίσεις. Την υλοποίηση αυτή συγκρίνουμε έπειτα με την πιο περίπλοκη unrolling εκδοχή της (Σχ 3.8). Οπως ήταν αναμενόμενο είναι καλύτερη, καθώς λόγω του unrolling ο compiler έχει περισσότερες ευκαιρίες για βελτιστοποιήσεις και παραλληλοποιήσεις.

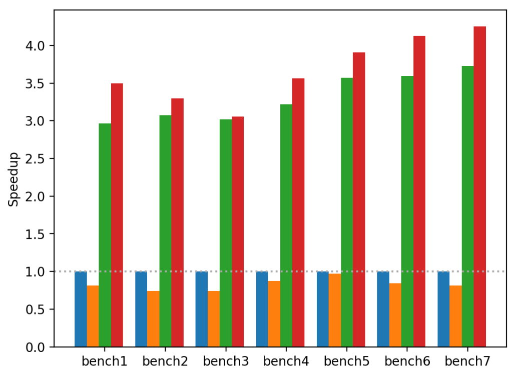


(a) Time **(b)** Speedup

Σχήμα 3.7: Απόδοση των vanilla, cluster asmjit and easyjit υλοποιήσεων στο Benchmark 8.3 στο Desktop



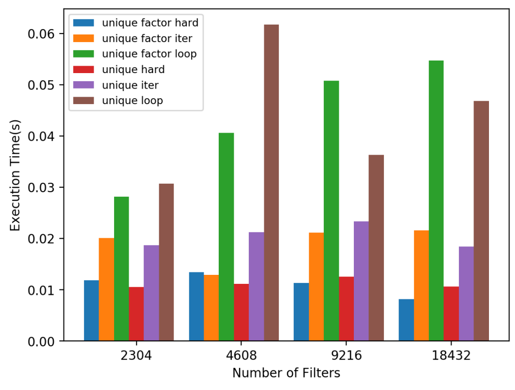
(a) Time



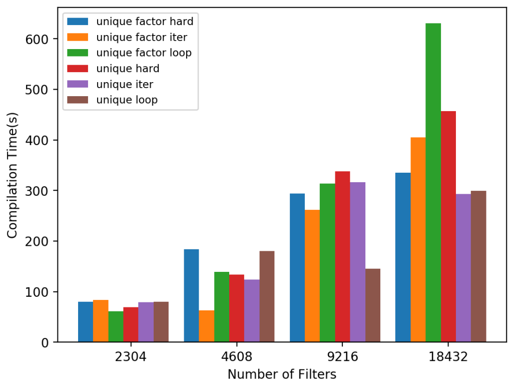
(b) Speedup

Σχήμα 3.8: Σύγκριση των jit υλοποιήσεων με την βιβλιοθήκη μας στο benchmark 8.4 στο μηχάνημα Desktop.

Κάθε unique filter έχει την δική του συνάρτηση. Καθώς ο αριθμός των φίλτρων αυξάνει περιμένουμε ο χρόνο για compile να αυξάνεται. Από το Σχ. 3.9 παρατηρούμε πως αν και όλες οι εκδοχές έχουν περίπου την ίδια απόδοση, η hard coded είναι ελαφρώς καλύτερη. Παρατηρούμε επίσης ξανά πως δεν κερδίζουμε κάτι από τον κοινό παράγοντα για τους λόγους που αναφέραμε παραπάνω.

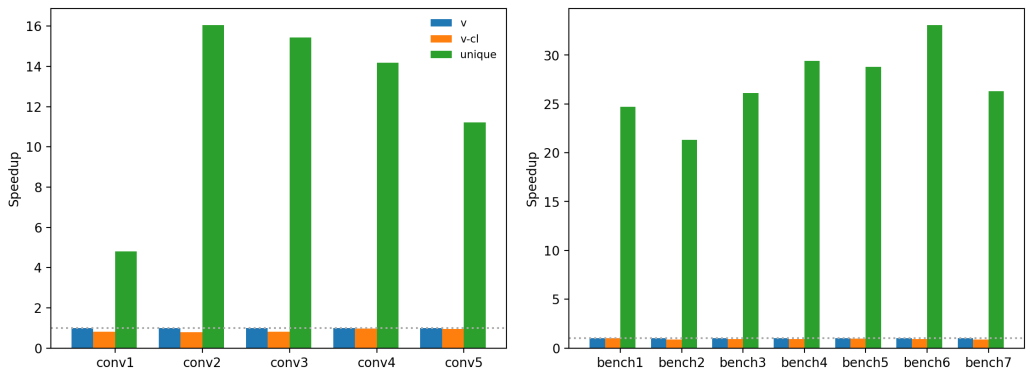


(a) Χρόνος εκτέλεσης στο Desktop



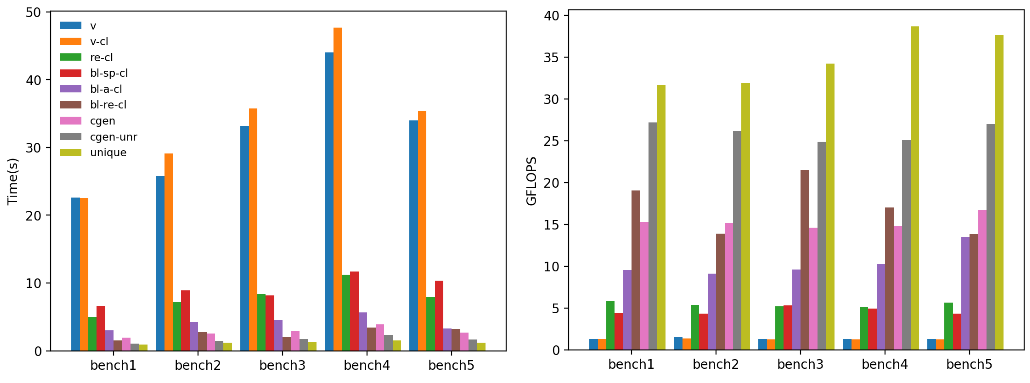
(b) Χρόνος Compilation στο Desktop

Σχήμα 3.9: Χρόνος συναρτήσει αριθμού kernel



Σχήμα 3.10: Σύγκριση unique filters στα benchmark 8.3 και 8.4 στο Desktop

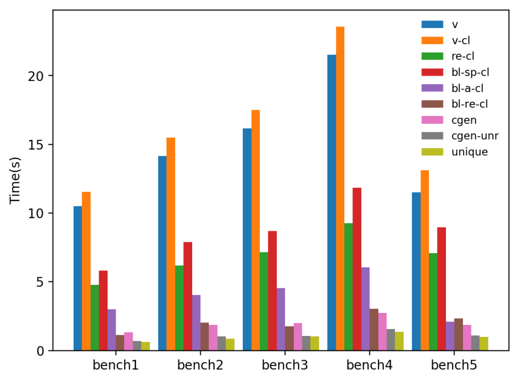
Αξίζει να σημειωθεί πως από τους compiler που δοκιμάσαμε ο icc παρήγαγε τον πιο αποδοτικό κώδικα. Στη συνέχεια ο clang και τέλος ο gcc. Αυτό οφείλεται στην ικανότητά τους να κάνουν vectorize η όχι κάθε υλοποίηση. Στη συνέχεια εξετάζουμε συγκεντρωτικά τα αποτελέσματα για τις παραπάνω υλοποιήσεις ώστε να εξάγουμε συμπεράσματα. Παρατηρούμε πως οι unique και unrolling υλοποιήσεις είναι οι πιο αποδοτικές στο Desktop και Server (Σχ. 3.11 και 3.12) Παρατηρούμε επίσης πως ο Server έχει ελαφρώς καλύτερη απόδοση πιθανών λόγω του avx-512 εναντι του avx2 του Desktop. Απο την άλλη στο μηχάνημα Edge παρατηρούμε από το σχήμα 3.13 πως η απλή codegen είναι καλύτερη. Αυτό είναι πολύ πιθανό να οφείλεται στο μικρότερο vector length του επεξεργαστή, καθώς και στη χρήση του clang compiler. Τέλος χωρίς vectorization στο ίδιο μηχάνημα (Σχήμα 3.14) οι unique και unroll αποδίδουν καλύτερα.



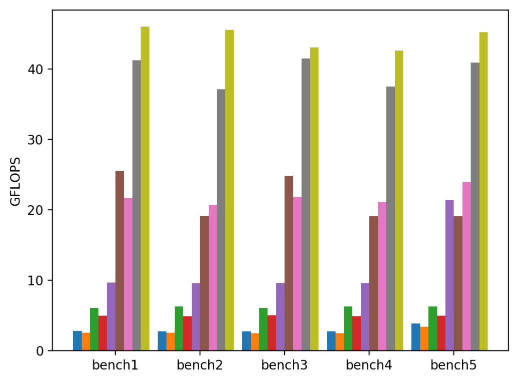
(a) Χρόνος

(b) FLOPS

Σχήμα 3.11: Συγκεντρωτικά η απόδοση στο Desktop για το benchmark 8.4

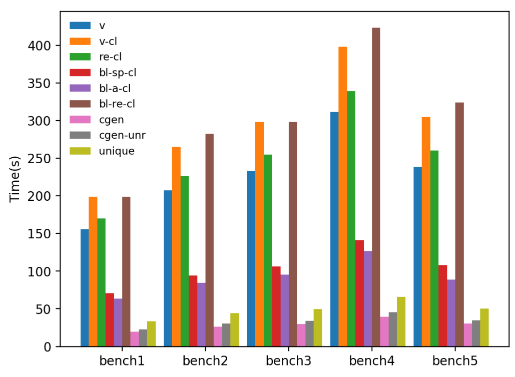


(a) Χρόνος

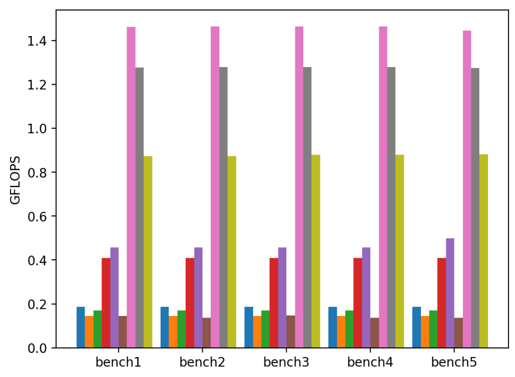


(b) FLOPS

Σχήμα 3.12: Συγκεντρωτικά η απόδοση στο Server για το benchmark 8.4

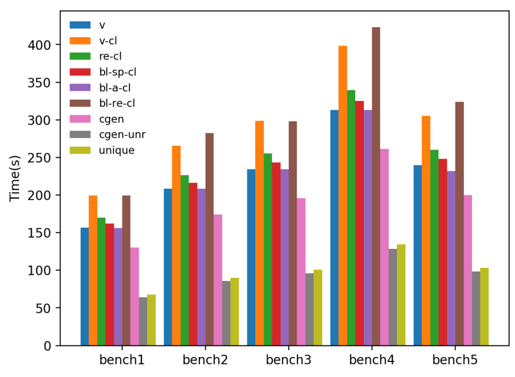


(a) Χρόνος

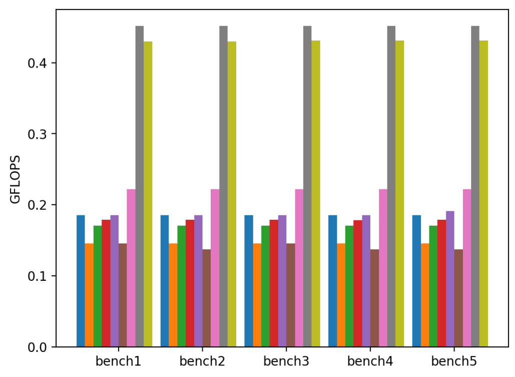


(b) FLOPS

Σχήμα 3.13: Συγκεντρωτικά η απόδοση στο Edge μηχάνημα για το benchmark 8.4



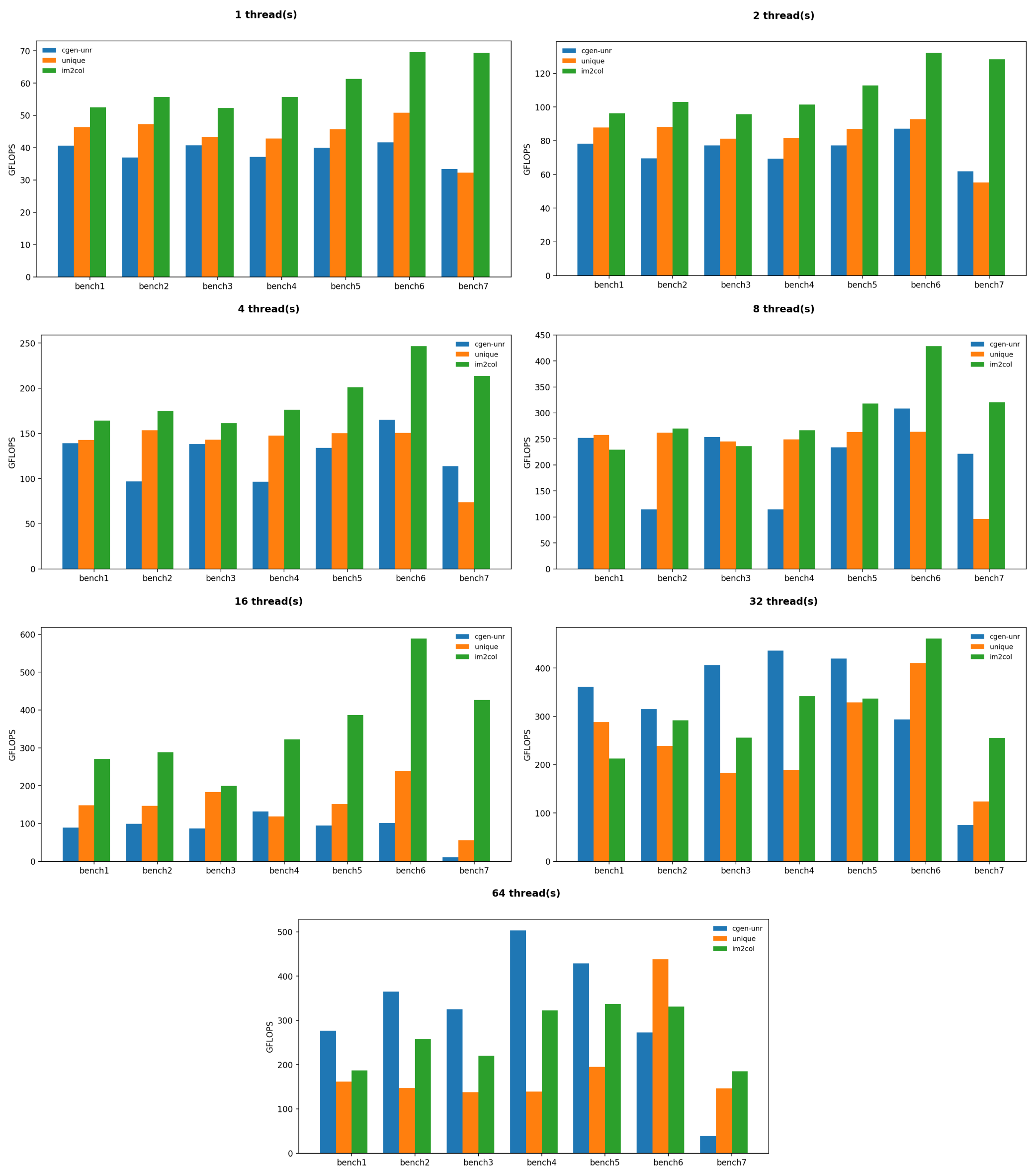
(a) Χρόνος



(b) FLOPS

Σχήμα 3.14: Απόδοση στο Edge μηχάνημα χωρίς vectorization για το benchmark 8.4

Τέλος για τις υλοποιήσεις με clustering που ξεχώρισαν με την απόδοσή τους, την codegen unrolling και την unique filters, εξετάζουμε τι απόδοση έχουν σε πολυπύρηνο περιβάλλον με μια απλή υλοποίηση OpenMP. Ταυτόχρονα τις συγκρίνουμε με την παραδοσιακή υλοποίηση για non clustering συνελίξεις, όπου πρώτα οι πίνακες μετασχηματίζονται σε 2-D ώστε να υπολογιστεί η έξοδος με πολλαπλασιασμό πινάκων από υψηλής απόδοσης GEMM functions. Παρατηρούμε αρχικά από το σχήμα 3.15 ότι η απόδοση των υλοποιήσεών μας είναι συγκρίσιμη με την GEMM υλοποίηση. καθώς τα threads αυξάνονται μεν υπάρχουν αυξομειώσεις αλλά η απόδοσή τους κλιμακώνει περίπου παρόμοια με την GEMM υλοποίηση. Επίσης παρατηρούμε πως η unrolling εκδοχή κλιμακώνει καλύτερα από την unique.



Σχήμα 3.15: Παράλληλη απόδοση του Server στο benchmark 8.4

Κεφάλαιο 4

Συμπεράσματα και μελλοντικές προεκτάσεις

Σε αυτή τη διπλωματική αρχικά μελετήσαμε τα αποτελέσματα το clustering ως μέθοδο συμπίεσης DNNs και διαπιστώσαμε πως επιφέρει καλά αποτελέσματα στην συμπίεση και ταυτόχρονα στη διατήρηση της ακρίβειας του μοντέλου. Παρ' όλ' αυτά εισάγει χρονικές καθυστερήσεις κάτι που οφείλεται στον ακανόνιστο τρόπο με τον οποίο προσπελάζεται η μνήμη. Οι έμμεσες αναφορές στη μνήμη είναι η κύρια πηγή για την κακή χρονική απόδοση καθώς μειώνει την τοπικότητα των δεδομένων και την δυνατότητα vectorization.

Στη συνέχεια επικεντρωθήκαμε στην ελαχιστοποίηση των καθυστερήσεων λόγω του clustering. Πειραματιστήκαμε με διάφορες τεχνικές βελτίωσης ώστε να κρύψουμε τις καθυστερήσεις. Χρησιμοποιήσαμε loop reordering και loop blocking. Τα αποτελέσματα δείχνουν πως με αυτές τις τεχνικές είναι εφικτό να πετύχουμε μια αρκετά μεγάλη αύξηση στην απόδοση. Οι τεχνικές αυτές στηρίζονται κυρίως στην ύπαρξη SIMD εντολών και είναι κατα συνέπεια χρήσιμες σε ισχυρούς υπολογιστές.

Στη συνέχεια πειραματιστήκαμε με Just In Time compilation. Προσαρμόσαμε τον αλγόριθμο της συνέλιξης και είδαμε αύξηση στην χρονική απόδοση. Χρησιμοποιήσαμε δύο framework για το παραπάνω, αλλά κάθε ένα είχε τους δικούς του περιορισμούς. Έτσι αναπτύξαμε μια δική μας βιβλιοθήκη παραγωγής και εκτέλεσης κώδικα στο runtime, που κάλυπτε τις ανάγκες μας και έτσι ήταν εφικτό να παράξουμε υλοποιήσεις που είχαν καλύτερη απόδοση από τα προηγούμενα framework. Οι υλοποιήσεις αυτές μπορούν να χρησιμοποιηθούν σε διάφορες πλατφόρμες αλλά συστήματα με λίγες υπολογιστικές ικανότητες θα ωφεληθούν περισσότερο.

Τέλος με την παραπάνω βιβλιοθήκη είχαμε την ελευθερία να απαλείψουμε όλες της έμμεσες αναφορές. Αυτό βελτίωσε σημαντικά την απόδοση σε όλα τα συστήματα. Το μειονέκτημα είναι πως χρειάζεται μικρός αριθμός kernels για να είναι κάτι τέτοιο εφικτό.

Συγκεντρωτικά σε αυτή τη διπλωματική καταφέραμε να συμπιέσουμε DNN μοντέλα έως και 6.4x χωρίς πτώση της ακρίβειάς τους. Ταυτόχρονα, μέσω του JIT compilation, καταφέραμε να διατηρήσουν και την ταχύτητά τους χωρίς το memory overhead των GEMM convolutions

Τα αποτελέσματα της διπλωματικής αυτής μπορούν να προεκταθούν σε πολλές κατευθύνσεις. Αρχικά κάθε βελτιστοποίηση της συνέλιξης μπορεί να προσαρμοστεί για βήμα του back propagation της εκπαίδευσης των DNNs. Τα loops και στις δύο περιπτώσεις είναι παρόμοια και οι βελτιστοποιήσεις θα έχουν κατά πάσα πιθανότητα παρόμοια αποτελέσματα.

Η βιβλιοθήκη που αναπτύξαμε χρησιμοποιήθηκε για να κάνει compile κάποιες παραμέτρους μέσα στη συνέλιξη αλλά και για να κάνουμε unroll τα loops του kernel. Με την ελευθερία που παρέχει η βιβλιοθήκη αυτή μπορεί η βελτιστοποίηση να πάει ένα βήμα παραπάνω με πιο fine grained λύσεις στο πρόβλημα είτε στοχεύοντας συγκεκριμένα μεγέθη kernel, είτε με πιο περίπλοκα loop transformations. Ακόμα έχει ενδιαφέρων η χρήση αυτών των βελτιστοποιήσεων σε παράλληλα περιβάλλοντα όπως οι GPUs.

Τέλος τεχνικές που μειώνουν τα μοναδικά kernels ως μέθοδο συμπίεσης υπάρχουν ήδη και μπορούν να ωφεληθούν από τη δουλεία μας. Συγκεκριμένα αυτές οι δουλειές επικεντρώνονται στο να λύσουν το πρόβλημα των περιορισμών μνήμης των ενσωματωμένων συσκευών. Οι συσκευές αυτές συνήθως έχουν και χαμηλή υπολογιστική ικανότητα κάτι στο οποίο θα μπορούσε να βοηθήσει η υλοποίηση με τα μοναδικά kernel.

Chapter 5

Introduction

Machine Learning applications today are becoming widely used and integrated in products and services (e.g. recommendation algorithms for products or movies). In particular, Deep Learning has become one of the most researched technologies, achieving near equal or even better performance than humans for specific tasks. The architecture behind Deep Learning is called Deep Neural Networks (DNNs). Although they were invented in the 1980s, their breakthrough was in the 2000s. At that time, the advances on graphics processing units (GPUs) resulted in a huge increase of computational capabilities. Using GPUs allowed for fast implementations of the computationally intensive DNN training. In the recent years DNNs have shown significant improvements in many AI applications. A particular class of DNNs called Convolutional Neural Networks (CNNs) dominate the domain of computer vision by achieving state of the art accuracy on tasks such as image classification[28] (Fig. 5.1), object recognition [39] and image segmentation [32]. CNNs have already surpassed traditional computer vision techniques as seen (Fig. 5.2). The general tendency for improving the accuracy performance of such networks has been to design networks with more layers (deeper) and more sophisticated architecture. However, as the networks grow larger and larger, this improvement comes at the cost of high memory consumption, computational requirements and energy consumption. For example AlexNet [28], has 62.3 million parameters, and needs 1.1 billion operations for the inference of a single image.

The high performance of DNNs and CNNs in particular has created the demand for such applications to be integrated on various embedded devices such as smart phones, IoT devices, and self driving cars [48]. The high memory and computational requirements can be easily satisfied by server grade and even some consumer grade computers but such demands are way over the range of the capabilities of most embedded devices due to their low memory, limited computational power and energy constraints. Therefore the deployment of such DNNs, as they are, is prohibitive on such devices. The need to utilize such models on resource limited devices has sparked a research effort to overcome the above limitations .

One approach is to design compact model architectures from scratch. Examples of this is using more efficient convolution blocks such as multi branched convolutions [45]

```
[(0.31244642, 'n02123045 tabby, tabby cat'),
 (0.23797025, 'n02123159 tiger cat'),
 (0.12387885, 'n02124075 Egyptian cat'),
 (0.10075199, 'n02119022 red fox, Vulpes vulpes'),
 (0.070957005, 'n02127052 lynx, catamount')]
```

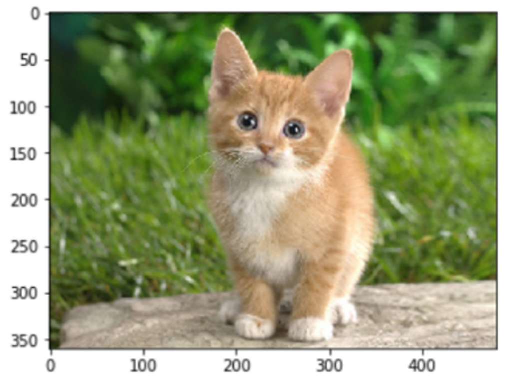


Figure 5.1: Example of image classification using CNN

[19], bottleneck convolutions[19] and depthwise separable convolutions [7] [23].

Another approach works by compressing existing models trained on powerful machines (e.g. GPU clusters) and then deploy them on resource limited devices with minimal accuracy loss (Fig. 5.3). This is based on the fact that contemporary big DNNs (Fig. 5.4) have significant redundancy in weights. This wastes both computational power and memory because not all of their numerous parameters are needed to correctly perform their tasks. Hence the the efforts on making the models smaller have been focused on having less weights and having less numerical accuracy. This is mainly achieved with connection pruning and quantization.

Quantization and pruning address different sources of redundancy on the models [55]. They are therefore complementary to each other and they even work better when combined, as first reported by Han et al. [18]. Their work is of the most thorough methods to compress DNNs for deployment on low resource devices. They propose a three stage compression pipeline consisting of pruning, quantization and Huffman encoding (Fig. 5.5). The three methods are complementary and can be used on top of each other. The combined techniques implemented, resulted in a 35x times reduction in size of the model.

Connection pruning removes less contributing neurons altogether to induce sparsity to a network. There are different approaches and criteria to determine which neurons are less important and therefore must be pruned. Some works prune individual weights [17] [2], others whole channels [49] [20] and others even whole filters [15] [49] [51].

Quantization is a process of converting a range of input values into a smaller set of output values that closely approximates the original data. Quantization uses fewer bits to represent each weight value of the neurons. This results in the quantized weights deviating from their original values. This deviation produces noise, which can be tolerated up to a point, since DNNs are trained to be robust against noise. Quantizaton is the main focus of

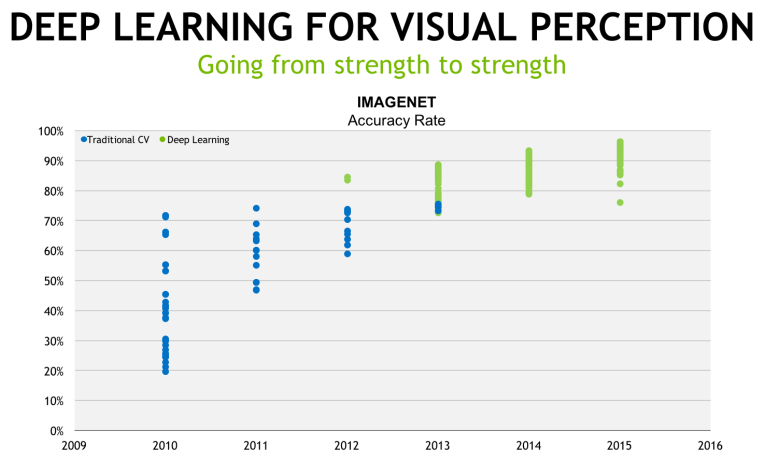


Figure 5.2: Performance of traditional computer vision vs Deep Learning

this thesis. A lot of research has focused on this subject which is mentioned in detail in the following section.

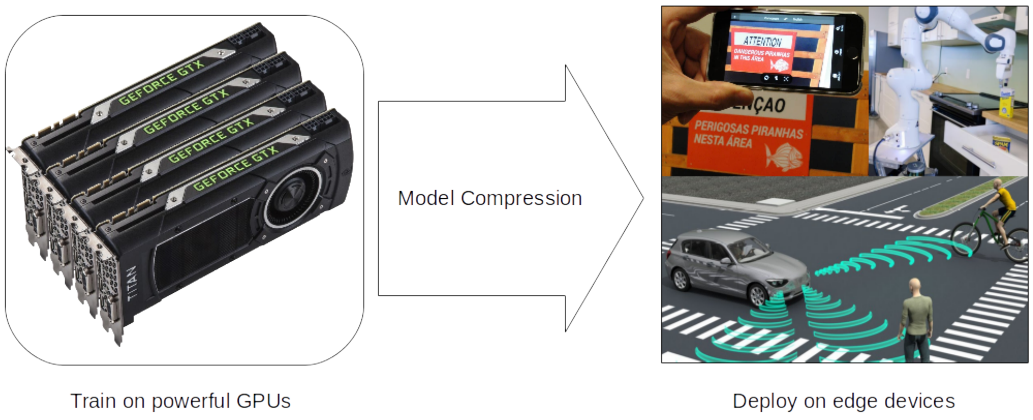


Figure 5.3: Train on powerful GPUs, compress and then deploy on embedded devices

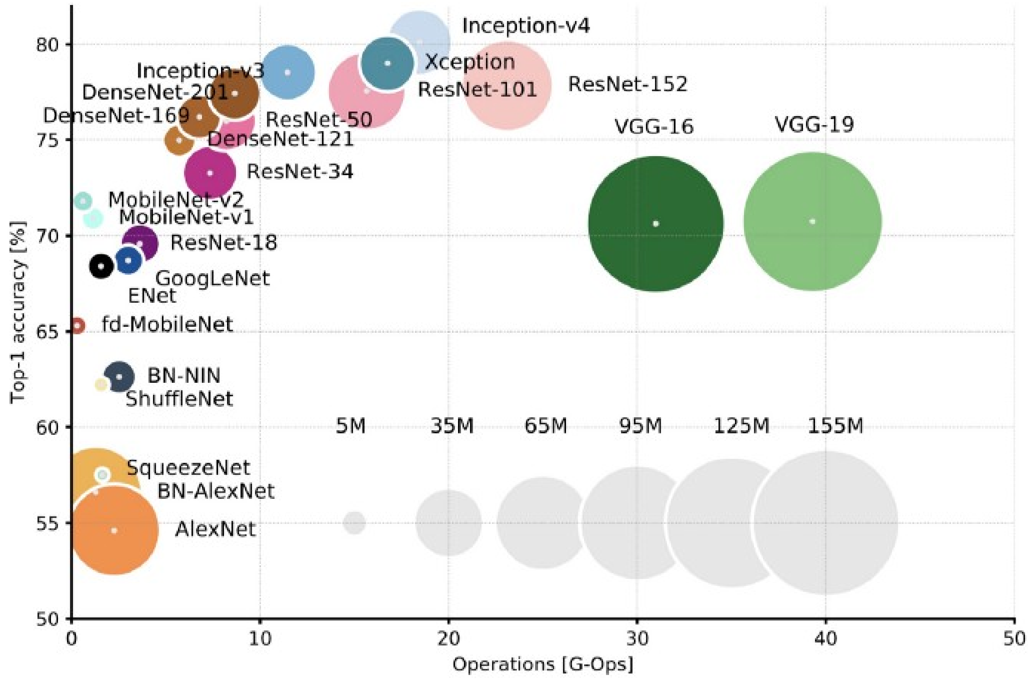


Figure 5.4: Top1 accuracy vs operations. The size of the blobs is proportional to the number of network parameters [4]

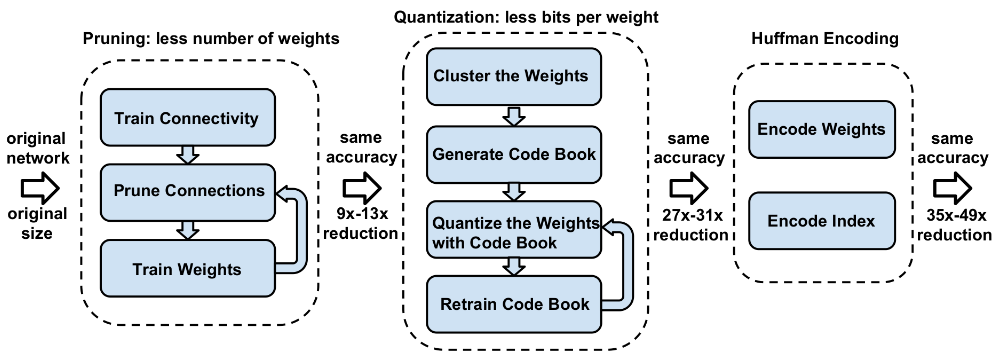


Figure 5.5: Deep compression pipeline [18]

5.1 DNN Quantization

Quantization focuses on using fewer bits to represent each the weight value of the neurons. The various quantization attempts mainly follow two paths to achieve reduced numerical precision and bit width of weights.

Some works focus on limiting the numerical precision of the parameters by converting floating point numbers to low precision fixed point number. These are called fixed point representations. To better represent the original numbers they take into account the distribution of the values being quantized by using dynamic fixed point. Fixed point quantization is a uniform quantization method. This means that the intervals between each quantized value are the same. Milde et al. [34] created a low-precision add-on for the Caffe framework [26] that rounds floating point values and represents them with integers for both the decimal and fractal part. They used this technique to quantize weights and activations. Shin et al. [42] optimizes fixed point representation by calculating iteratively the quantization step for better representing the dynamic range of weights. Miyashita et al. [36] used base-2 logarithmic representation of models. This way multiplications were replaced by cheaper shift operations. More aggressive attempts for quantization use even ternary [58] and binary weights [31] [8] [35] [21].

On the other hand some works achieve using fewer parameters by using indexed representations. The weights are grouped together by some criteria and each group represented by one value. These methods may not affect numeric precision, but they limit the available full precision numbers. Chen et al. [6] reduces the bitwidth needed to store the weights using a hash function to group weights into hash tables. The grouping is not optimal because it is random and thus doesn't take into account the weight values across the layer. This problem is addressed and overcome by Han et al. [18] by using k-means clustering to group weights together. In contrast with the previous work [6] this technique groups similar weight values together and therefore achieves better accuracy. For their experiments they achieve minimal accuracy loss while using as low as 5 bits for indexing. Other works experiment with k means clustering by using vector quantization and using vectors as cluster centers. In [50] they treat each weight row as a row vector and perform k means with these vectors as centroids. Other attempts [43] [54] even index whole filters with k means by utilizing k-means and 2D transformations.

De Prado et al. [9] observe that k-means quantization outperforms standard and dynamic fixed point quantization on both accuracy and compression rate. That is because k-means is a non uniform quantization method, meaning the intervals between the quantized values are different with each other. This way the fewer quantized values can more closely follow the original distribution of the weights.

Quantization introduces noise and that translates to reduced model accuracy. Most methods require training the models from scratch using the reduced precision weights. Others require or at least benefit from fine-tuning the models. This means retraining the network for a small number of iterations until the accuracy recovers from the drop that quantization introduced.

5.2 Contribution of this Thesis

In this thesis we focus specifically on quantization with indexed representations as a compression technique for CNNs. We investigate the effects of k-means clustering quantization on the weights of CNNs. We test their classification accuracy and performance. We then investigate the negative effects of irregular access patterns that clustering introduces on the performance of the convolution. In order to overcome the drawbacks of clustering we experiment with different approaches to first hide the latencies they introduce and we then create the necessary tools to eliminate them.

Chapter 6

Background

In this chapter we introduce some fundamental concepts that are mentioned throughout the thesis.

6.1 Machine Learning

Machine Learning (ML) is a subfield of Artificial Intelligence (AI). The domain of ML also intersects with Computer Science, Statistics and Information Theory. In general its aim is enabling computers to learn (or to be trained) from existing data to perform specific tasks automatically. The concept of learning in this context, means to gradually improve their performance on carrying out a task, without them being specifically programmed to do so.

The main Machine Learning algorithm subcategories are Supervised learning, Unsupervised learning, Semi-supervised learning and Reinforcement learning.

- Supervised Learning algorithms use existing data (train dataset) that consist of one or more inputs and an output or label to learn to predict the outputs of new data(test dataset) only from their inputs.

- Unsupervised learning algorithms are used when the data are unlabelled(are only inputs). The goal is to find patterns and commonalities in the data, recognize them in new data and organize similar data to clusters.

- Semi-supervised learning algorithms are somewhere in between the previous two categories. They use both labeled and unlabeled data but usually more unlabeled than labeled data. When unlabeled data are used with a small amount of labeled data the accuracy is vastly improved. This method is used when labeling the data requires skill and time, but unlabeled data are easy to obtain.

- Reinforcement learning algorithms interact with the environment with actions that have errors or rewards, with the goal being to maximize a reward function. This trial and error method enables computers to find the ideal behaviour within a specific context in order to maximize reward.

6.2 Neural Networks

One of the supervised machine learning methods are neural networks. Neural networks are computational models inspired from the biological neurons of humans and other animals. In that sense they obtain an input and they give an output depending on the importance of the input relative to other neurons. Neural networks are nowadays used in a variety of classification tasks. The neuron is the fundamental component of neural networks. A neuron receives a number of inputs from previous neurons and produces one output as seen in Fig. 6.1

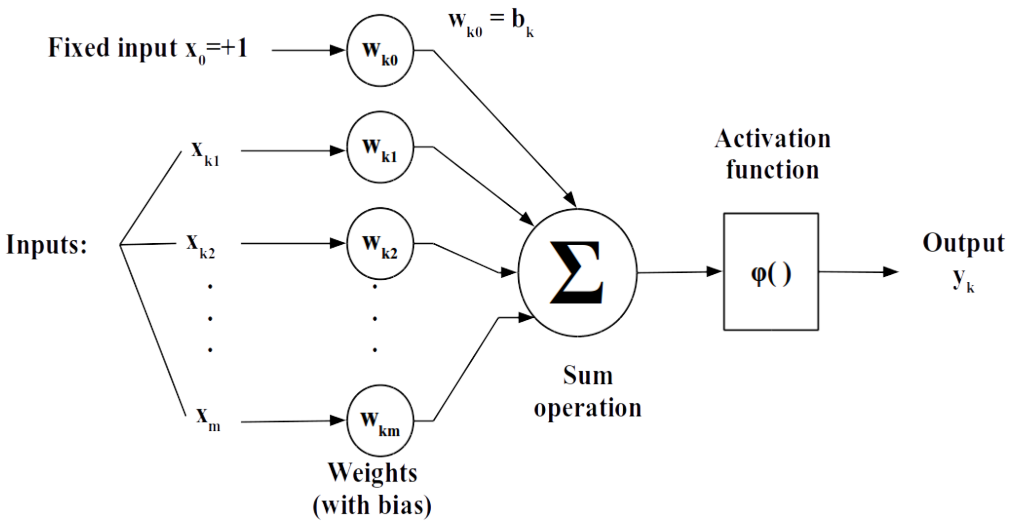


Figure 6.1: Model of a neuron

The output is the weighted sum of the inputs followed by the neuron's activation function. In mathematical form this is desribed as:

$$v_k = \sum_{j=0}^{m} w_{kj} x_j \tag{6.1}$$

$$y_k = \phi(v_k) \tag{6.2}$$

where x_j are the inputs and w_{kj} are the weights of the neuron. b_k is the bias. $\phi()$ is the activation function and its purpose is to introduce non-linearity into the output of a neuron. This enables the neuron to describe different linear and non-linear functions.

One of the most common arrangement of neurons is the multilayer perceptron (Fig 6.2). They consist of at least three layers of neuron nodes. Each neuron of one layer is connected to all neurons of the next layer and because of that they are called fully connected layers. The first layer is called input layer where the input is passed to the next layers. The last is called output layer, where we can measure the output. All the layers in between are called hidden layers and are responsible to predict the correct outputs from the inputs of the specific task. They learn to do that by adjusting the values of their weights during the learning process called training. The most common technique for training is back propagation. The weights are first randomly initialized. Then the model is given some inputs and predicts some outputs, in a process called inference. These outputs are compared to the correct outputs, called labels. Every time the label is different from the prediction, that error is transferred to all previous layers. Then the weights are adjusted according to how much they contributed to the error. This is done by first obtaining the gradients of the loss function with respect to each weight on each neuron with the back-propagation algorithm and using it to adjust the weight with gradient descent.

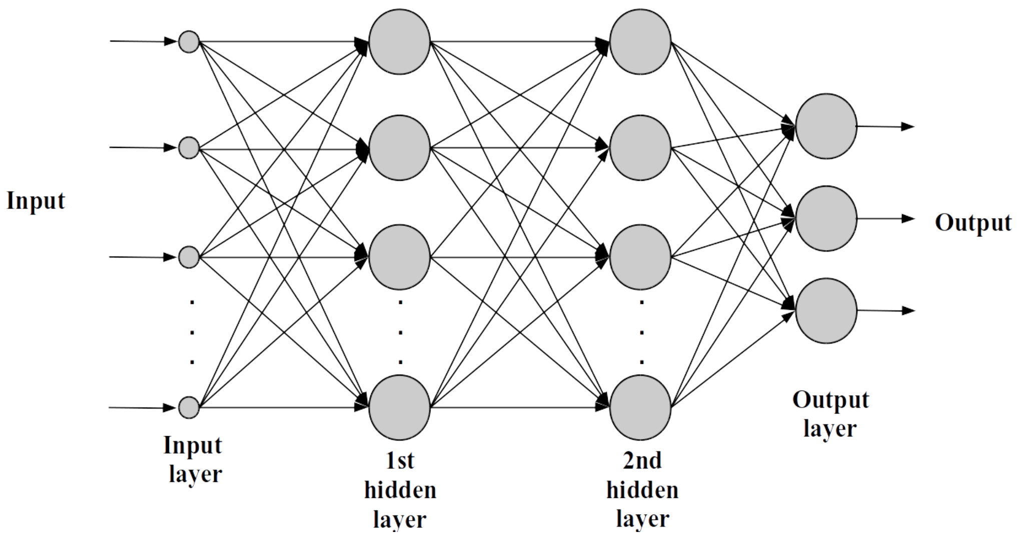


Figure 6.2: Multilayer perceptron with 2 hidden layers

Deep neural networks are neural networks with a large number of hidden layers between the input and the output. The number of hidden layers a Neural Network should have, to qualify as ”deep”, is not universally agreed upon but usually having two or more hidden layers counts as deep [22].

6.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a sub-category of DNNs that are specialized in image recognition. They have a lot in common with regular neural networks. Their layers also consist of neurons with weights and biases. The weights are also adjusted in the same manner during training. Neurons receive inputs, perform an operation with the weights and then apply an activation function. The difference is that CNNs scale better with images as inputs because the connections between two layers are restricted to local regions (Fig. 6.3) and are not fully connected with each other like regular neural networks (Fig. 6.2). This vastly reduces the amount of parameters in the network and also enables CNNs to recognize spatial traits of the input. Because of that they are able to detect important features of classes if given enough samples of each class.

CNN structure usually follows the pattern of a number of convolutional layers followed by max-pooling layers and at the end some fully connected layers(Fig. 6.4). In the following subsections the most commonly occurring layers on CNNs are briefly described.

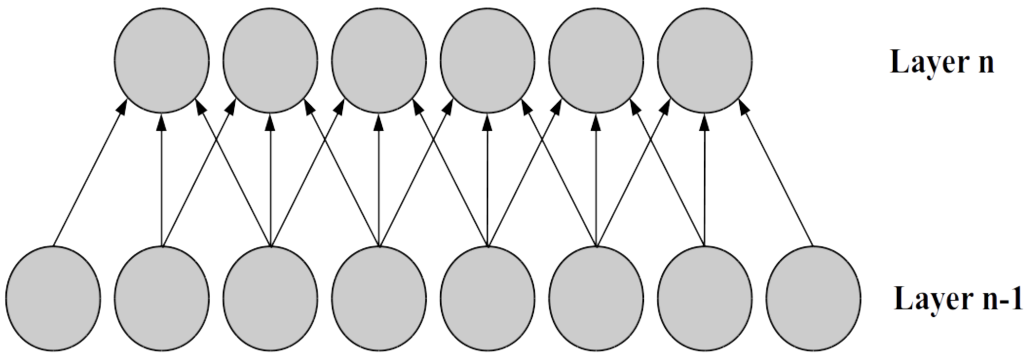


Figure 6.3: 2-D representation of local connections between convolution layers

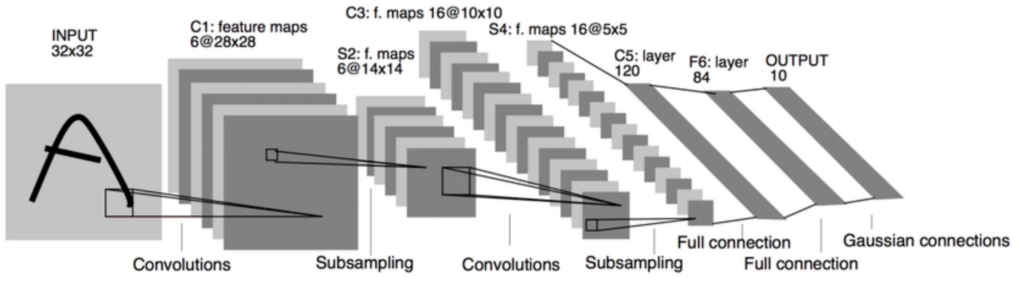


Figure 6.4: Lenet architecture [29]

6.3.1 Convolution layer

The fundemental operation of CNNs is the two dimensional convolution. Such convolutions are applied between inputs and weight matrices. Input dimensions are Input Channel, Height and Width. For example the first layer's input is usually an RGB image. The image is represented as a 3-D matrix with three color channels and its height and width dimensions as seen in Fig. 6.5.

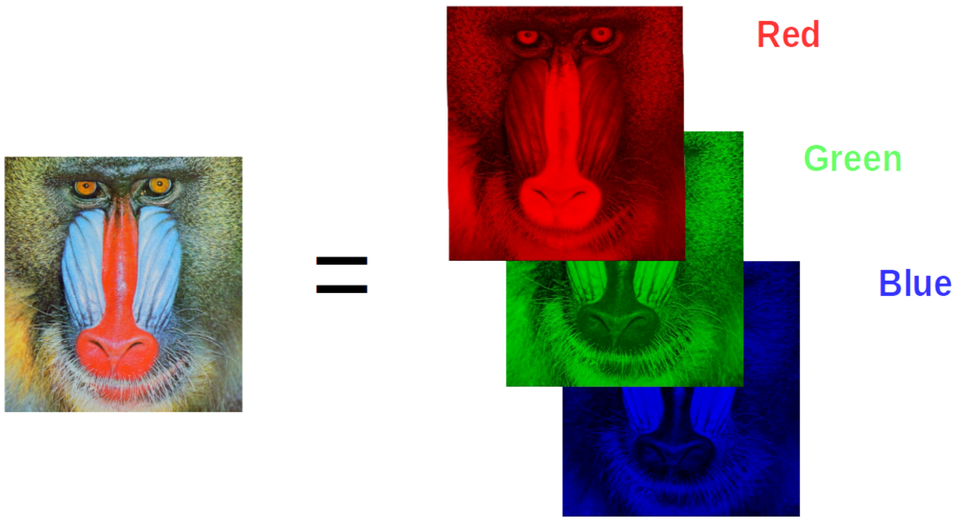


Figure 6.5: An RGB image consists of 3 channels red, green and blue

The weights of each layer are organized as 4-D structures (Output Channel, Input Channel, Height, Width). Esentialy they are an array that contains 3-D filters. A kernel is the 2-D sub-matrix of weights that slide over and are multiplied with the relevant 2-D part of the input to give an output (Fig. 6.6).

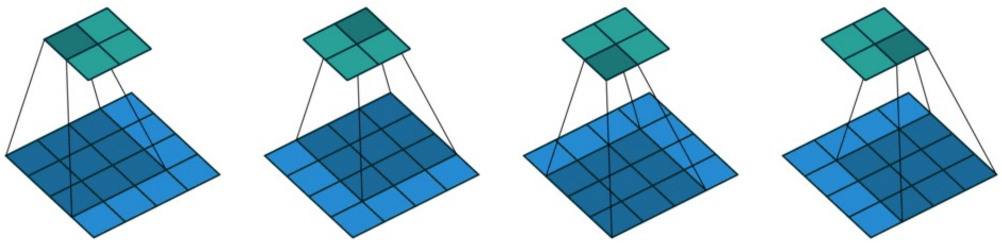


Figure 6.6: Example of a convolution of a single kernel with a single channel input [11]

A filter consists of multiple kernels, one for every input channel. The output of each filter is the addition of outputs of each such kernel. Each filter produces a single output channel. To produce the desired output channels, a number of filters equal to the output channels, are used (Fig. 6.7). Then the activation function is applied to each element and this final output is called a feature map.

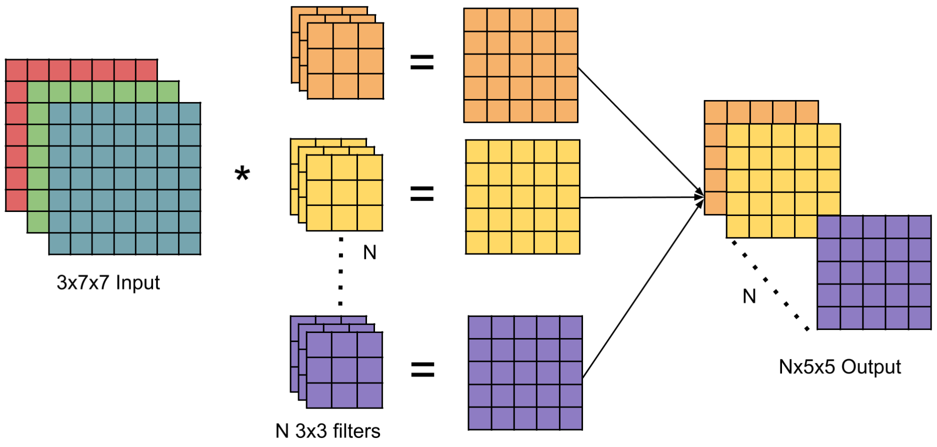


Figure 6.7: Example of a convolution of N filters with a 3 channel input(rgb image).

The number of elements the convolution filter moves vertically or horizontally at each step is called stride. For example in Fig. 6.6 the stride is 1. As the stride increases the feature maps are getting smaller. This is exploited when reduction of the output dimension size is needed, by using bigger strides to skip some elements.

Usually the size of the feature map ends up smaller than the input because the filter that slides over the latter is contained by it. We can observe that in Fig. 6.6. This makes it difficult to preserve the size of feature maps,when needed, because they would gradually shrink after each layer. To maintain the same size between the input and the output we can use a technique called zero padding. The input volume is padded with zeroes around its height and width dimensions Fig.6.8. To keep the input and output dimensions the same the padding is calculated as follows:

$$pad = \frac{F-1}{2} \tag{6.3}$$

where F is either one of the H_k or W_k kernel spatial dimensions.

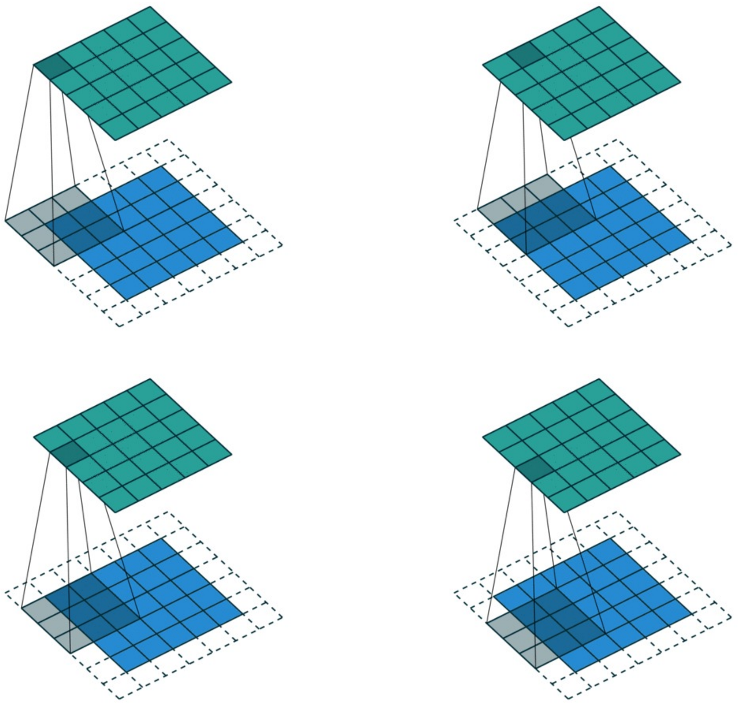


Figure 6.8: Example of convolution with zero padding [11]

Considering all the above, the output dimensions are calculated as follows:

$$H_o = \frac{(H_i - H_k + 2 * pad)}{stride} + 1 \tag{6.4}$$

$$W_o = \frac{(W_i - W_k + 2 * pad)}{stride} + 1 \tag{6.5}$$

Where:

Output channel : C_{out}

Input channel: C_{in}

Input Height, Width: H_i, W_i

Output Height, Width: H_o, W_o

Kernel Height, Width: H_k, W_k

Thus each convolution output element can be computed with the following equation:

$$Out(j,k,l) = \sum_{i=1}^{C_in} \sum_{m=1}^{H_k} \sum_{n=1}^{W_k} Kernel(i,j,m,n) * Input(i,k_{in},l_{in}) \tag{6.6}$$

By applying the previous equation to find the whole output matrix, we end up with the algorithm for the convolution operation (Alg. 8). To implement padding we check if the input elements are out of bounds.

Algorithm 8 Convolution algorithm

```
1: for $i = 0$ to $Cin - 1$ do
2:   for $j = 0$ to $Cout - 1$ do
3:     for $k = 0$ to $H_o - 1$ do
4:       for $l = 0$ to $W_o - 1$ do
5:         for $m = 0$ to $H_k - 1$ do
6:           for $n = 0$ to $W_k - 1$ do
7:             $k_{in} = stride \times k + m - pad$
8:             $l_{in} = stride \times l + n - pad$
9:             $Output[j][k][l] += Input[i][k_{in}][l_{in}] \times Kernel[i][j][m][n]$
```

6.3.2 Pooling layer

Usually a convolution layer is followed by a pooling layer. Pooling helps reduce the dimensions size and thus the overal computational intensity. Pooling works similarly to convolution. The filter passes over the image and extracts the most important information to a smaller sized output. Essentially pooling layers downsample the height and width dimensions of the feature map. Depending on the criteria of extracting important information from the input the two most notable types of pooling are average pooling and max pooling (Fig. 6.9) with the latter being the most commonly used because it outperforms other pooling techniques. Pooling tends to be replaced in favour of strided convolutions, achieving better accuracy with the same reduction in dimensionality (Springenberg et al. [44]).

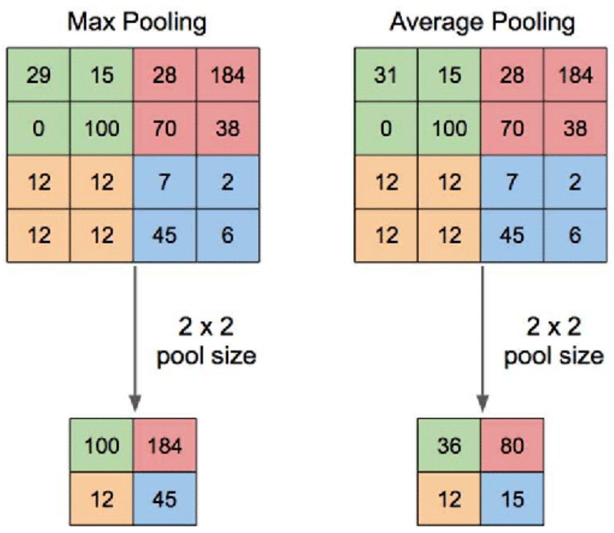


Figure 6.9: Example of average and max pooling [52]

6.3.3 Fully Connected Layer

After some iterations of convolution/pooling layers, usually the 3-D output of that process is flattened to one dimension and fed to a series of fully connected layers, just like in regular DNNs (Fig. 6.2). The last fully connected layer determines the prediction of the model. Thus its activation function is a softmax, to keep all predictions between 0 and 1 and their sum to 1. Some CNN models substitute fully connected layers with appropriate convolution layers because it achieves better performance and also enables models to be tweaked to perform semantic segmentation [32].

6.3.4 Batch Normalization Layer

DNNs update their weights after evaluating a number of training samples called a batch. Batch normalization or BatchNorm [25] is a technique that enables each layer of a network to learn more independently from other layers. It is a separate layer that during training normalizes each batch by subtracting the batch mean and then dividing in by the batch standard deviation. This brings the weight values between 0 and 1. To fix that, a linear transformation is performed with two learnable parameters γ, the scale, and β, the shift, of the transformation. Below are the relevant equations, where ϵ is a constant for numerical stability [25].

$$\mu_{batch} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{6.7}$$

$$\sigma^2_{batch} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{batch})^2 \tag{6.8}$$

$$\hat{x}_i = \frac{x_i - \mu_{batch}}{\sqrt{\sigma^2_{batch} + \epsilon}} \tag{6.9}$$

$$y_i = \gamma \hat{x}_i + \beta \tag{6.10}$$

During inference the batch normalization layer acts like a linear transformation. Since the convolution is also a linear transformation both layers can be, and often are, merged to save on computation cost and memory footprint. This process is called folding. The new weights are then calculated as follows:

$$W_{fold} = \gamma \frac{W}{\sqrt{\sigma^2 + \epsilon}} \tag{6.11}$$

6.4 K-means clustering

K-means clustering is an iterative unsupervised machine learning algorithm that groups a set of observations ($X = \{x_1, x_2, ..., x_n\}$) into k distinct non overlapping groups (clusters) $C = \{C_1, C_2, ..., C_k\}$. Each cluster is represented by a value called a centroid. These values together make a codebook. The algorithm minimizes the squared Euclidean distances of the elements of each cluster from their mean μ_i [33]. Mathematically this is expressed as:

$$\underset{C}{argmin} \sum_{i=1}^{k} \sum_{x_j \in C_i} \parallel x_j - \mu_i \parallel^2 \tag{6.12}$$

Where μ_i is the mean of cluster C_i.

The centroids first have some initial values. Then two alternating steps are performed (Fig. 6.10):

- Assignment step: Each observation is assigned to the cluster with the least squared Euclidean distance from its centroid.

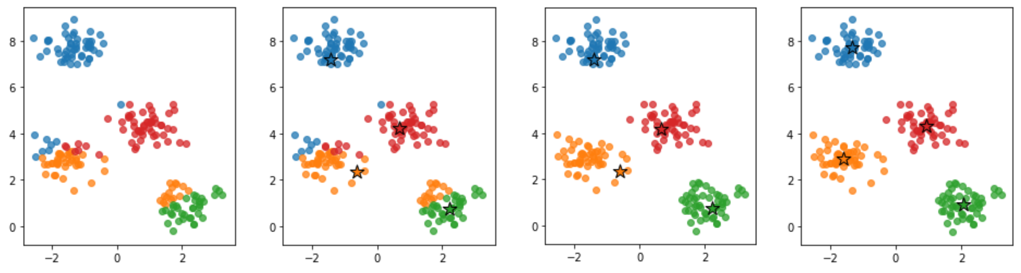
$$C_i = \{x_k \in X : \parallel x_k - m_j \parallel^2 \leq \parallel x_k - m_i \parallel^2 j, 1 \leq j \leq k\} \tag{6.13}$$

- Update step: The centroid value of each cluster (m_i) is recalculated as the mean of all observations assigned to that cluster.

$$m_i = \frac{1}{n_i} \sum_{x_j \in C_i} x_j \tag{6.14}$$

  Where n_i is the number of elements assigned to cluster C_i

The algorithm continues until the assignments no longer change. That is when the algorithm has converged. The algorithm does not guarantee to find the optimum solution because it can converge to local minima. That is why to obtain the best results, it is useful to run it several times with random initializations. K-means is one of the most popular



(a) **(b)** **(c)** **(d)**

Figure 6.10: Alternating steps of k-means [24]. (a) Each value is assigned to a cluster. (b) The cluster centers are calculated. (c) Each value is assigned to a new cluster. (d) The cluster centers are calculated again.

vector quantization algorithms. Vector quantization is a data quantization technique that originates from signal proccessing [47] and is a lossy data compression. It works by dividing a big set of observations into groups, with each group represented by a centroid value. K-means achieves the above by minimizing squared Euclidean distances.

Chapter 7

DNN Quantization

In this chapter we introduce a quantization methodology for DNNs and evaluate its effects on model compression and accuracy.

7.1 Quantization Methodology

To perform quantization we encode the weight values to fewer representative values with clustering. Similar to Han et al. [18] we use the k-means algorithm to group the weights of each layer (kernel matrix) into a number of clusters and find for each one the representative values (centroids) closest to the original ones. Depending on how many bits we are using for the codebook size, the number of clusters and therefore distinct entries available on the codebook matrix is 2^{bits}. After performing k-means, the values of the weight matrix are just indexes that point to the codebook matrix (Fig 7.1). Instead of accessing the elements of the weight matrix directly, there is a level of indirection. Thus the computation of the innermost loop (line 9) of algorithm 8 becomes:

$$Output[j][k][l] += Input[i][k_{in}][l_{in}] \times Codebook[Kernel[i][j][m][n]] \tag{7.1}$$

Son et al. [43] performed kernel quantization by using k-means, with the kernel as the main quantization unit, to cluster whole kernels instead of single weights. They achieve 10x compression with clustering and 30x compression with prunning and clustering kernels. Yu et al. [54] perform the same clustering but focus more on 3x3 kernels. This way they increase the maximum theoretical compression rate from 32 to 288. Their results achieve a 5.78 compression rate without accuracy drop.

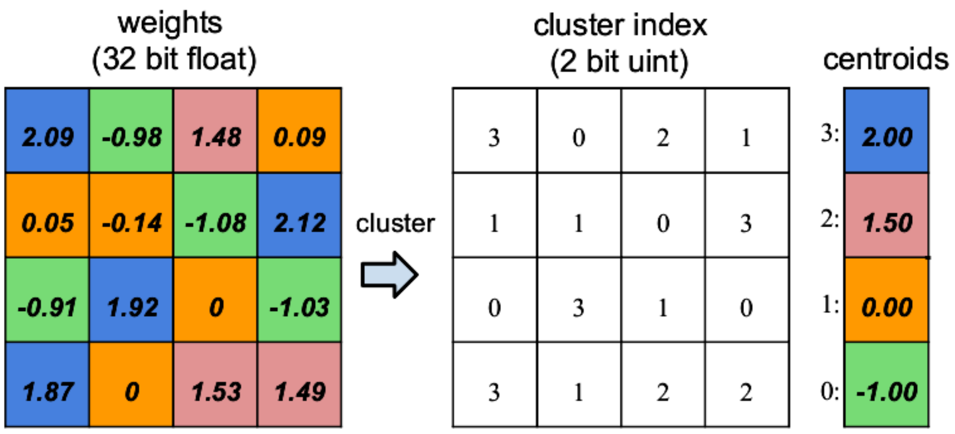


Figure 7.1: Example of clustering kernel weights using 2-bits [18]

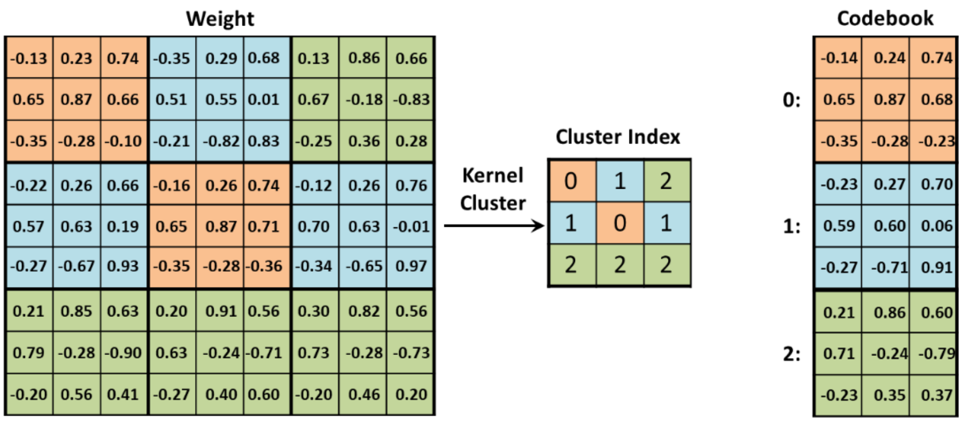


Figure 7.2: Example of clustering whole kernels [54]

7.2 Compression rate

After k-means quantization each weight is an index to the codebook matrix. Thus the size of each weight is reduced to $\log_2(k)$ where k is the codebook size. The compression rate r can be calculated as:

$$r = \frac{n * s}{n * \log_2 k + k * s} \tag{7.2}$$

Where n is the number of weights and s = sizeof(float),

Assuming that the number of codebook entries is negligible compared to the number of weights and given that the codebook size is 2^{bits} the compression rate for each layer is:

$$r = \frac{s}{bits} \tag{7.3}$$

For 32bit floating point numbers the maximum theoretical compression rate is 32 if we use 1 bit length codebooks.

7.3 Quantization noise

Quantization produces quantization noise which results in accuracy loss. Lin et al. [30] proves that if the weights follow a Gaussian distribution, then all the weights and activations contribute equally to the total signal to quantization noise ratio (SQNR). Zhou et al. [57] proved that each layer's quantization contributes independently to the total accuracy degradation because of the quantization noise of a model. Both cases where studying uniform quantization. K-means performs non-uniform quantization so the relationship between SQNR and classification accuracy is not well described, but there is a general trend that as SQNR increases we expect the accuracy to drop.

De Prado et al. [9] using the above trend between SQNR and accuracy loss, optimized the accuracy versus compression rate problem by finding each layers optimal number of bits. This required analysing the effect the quantization of each layer has to the total accuracy. Thus layers contributing more to the accuracy of the model would be given more precision and to compensate for that, layers that do not contribute as much, would then be compressed more.

7.4 Experimental Setup

7.4.1 Deep Learning Frameworks

We used Caffe [26] for a deep learning framework. Caffe is one of the first deep learning frameworks. In Caffe the architecture of the models is described in a high level representation (e.g. the model parameters and hyper-parameters), a ”prototxt” file. Then the model is deployed using input data and the weight values that are stored in caffemodel files. The weights of the models mentioned in the next sections are available in Caffe as already trained caffemodels.

To test the accuracy of the quantized models, we created new variants of Convolutional and Fully-Connected Layers that support clustering for the caffe framework. The data were proccessed, then we performed k-means clustering for various bit lengths and weight initializations for AlexNet, GoogleNet, Mobilenet and Mobilenet v2 and measured accuracy.

7.4.2 Datasets

There are various datasets on which to use the CNN models. Since the pre-trained CNN models availabel for Caffe were trained on Imagenet, we use that to test the accuracy of the models. Imagenet is a large scale dataset, containing human annotated images, designed to be used in visual object recognition [10]. There are around 14 million images and 21 thousand categories or classes. The dataset is used in the ILSVRC challenge. Models are trained on a train set of images and then are tested on a separate set of images they have never encountered before called the test set. The classification accuracy metrics are Top-1 and Top-5 accuracy. Top-1 accuracy measures the percentage of correctly labeled images. The Top-5 accuracy measures the percentage of images, where one of the 5 labels with the highest probability was the correct one (Fig. 5.1).

7.4.3 CNN Models

AlexNet [28] made a breakthrough winning the ImageNet competition in 2012 by a large margin and demonstrating the potential of large CNNs trained on massive datasets on the now widely available gaming GPUs. Based on LeNet [29] it expanded on its concept by utilizing a number of convolution layers with ReLu activations and max-pooling. The use of dropout and data augmentation (image cropping, rotation, flipping and PCA color augmentation) prevented overfitting. This gave AlexNet state of the art accuracy of top-1 63.3% and top-5 84.6%

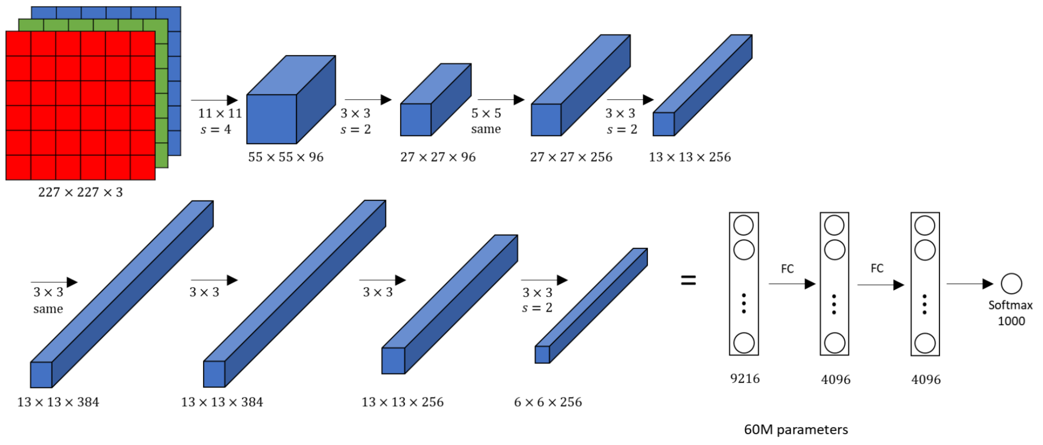


Figure 7.3: AlexNet architecture

GoogleNet was one implementation that used the inception module [45] which introduced multiple sized filters operating on the same level. Their outputs are concatenated and passed to the next layer. All parallel paths use the appropriate padding to give the input and output the same height and width. This was done to overcome the problem of scale variation in important visual traits of the image that made choosing the correct filter size for recognizing said important traits difficult. It also reduced overfitting and gradient loss by using two more predictions across the model together with the final to evaluate the accuracy loss. Also using smaller stacked convolutions reduced the computational intensity. The pretrained caffe model achieved top-1 accuracy is 63.8% and top-5 85.2%

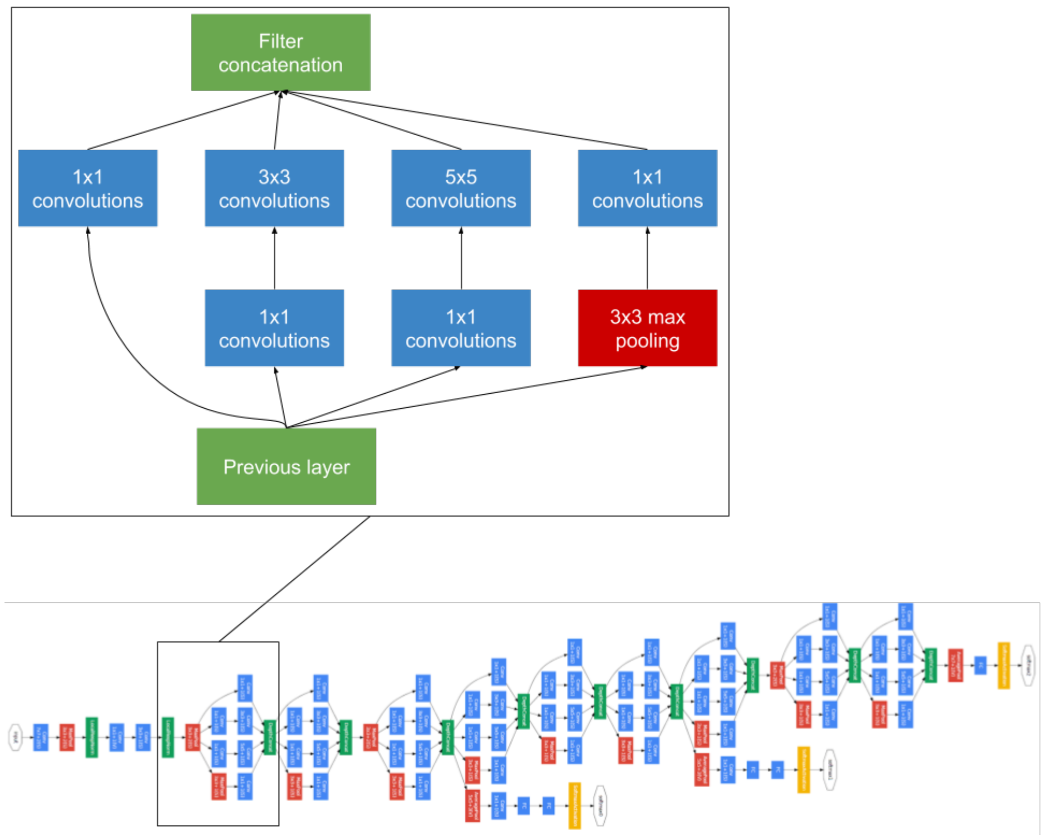


Figure 7.4: GoogleNet architecture and inception module detail

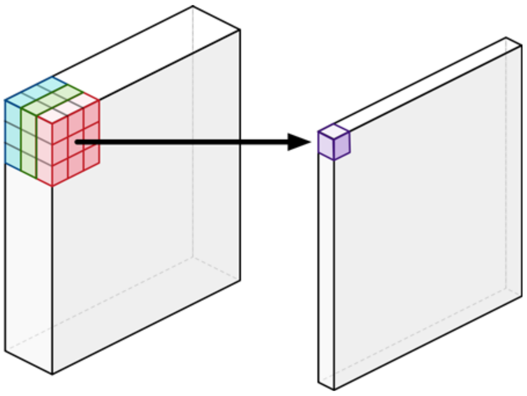
Mobilenets [23] [40] were designed with limited resources in mind. They utilise Depthwise Separable Convolutions in order to use less parameters to save memory and computational power. This process is performed in two parts (Fig. 7.5 (b)):

- Depthwise convolution where a single kernel per each input channel is applied but unlike regular convolutions the outputs are not combined to a single output channel. This way an intermediate output with size $Cin \times H_o \times W_o$ is produced.

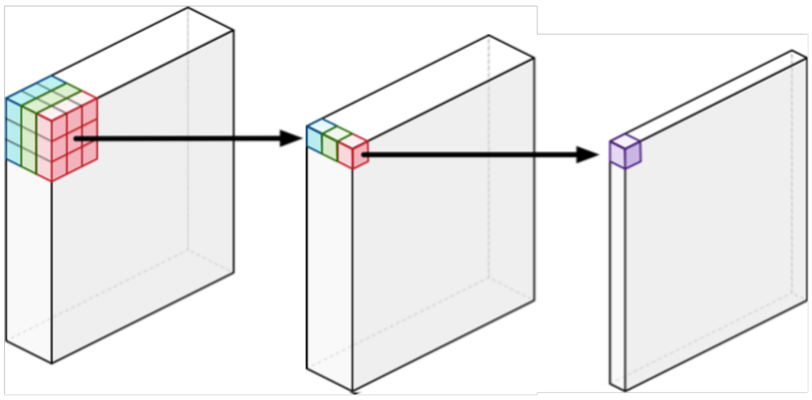
- Pointwise convolution, $Cout \times Cin$ of simple 1×1 convolutions, are then used for a linear combination of the elements of each channel of the previous output, to create a linear combination of of the output of the depthwise convolution.

We used two pretrained Mobilenet models [1], Mobilenet and Mobilenet v2. On Mobilenet we measured 68.2% top-1 accuracy and 88.4% top-5 accuracy and Mobilenet v2 69.6% and 88.8% respectively. The saving in both storage and multiplication operations is significant. Regarding storage they achieve the same accuracy with AlexNet with just a fraction of its parameters. AlexNet has 61M and is 233MB in size while the two Mobilenet variants have 17MB and 14MB respectively. In a normal convolution there are:

$Cout \times Cin \times Hk \times W_k \times Ho \times Wo$ multiplications. In Depthwise Separable Convolutions there are $(1 \times Cin \times Hk \times Wk \times Ho \times Wo) + (Cout \times Cin \times Hk \times Wk \times 1 \times 1)$ multiplications. For example to get a 256x8x8 output from a 3x12x12 input image with a normal convolution we need 256x3x5x5x8x8=1 228 800 multiplications. With Depthwise Separable Convolution we need 1x3x5x5x8x8 = 4800 plus 256x1x1x3x8x8=49152 for a total of only 53 952 multiplications.



(a)



(b)

Figure 7.5: (a) Normal convolution (b) Depthwise separable convolution

7.4.4 Data preprocessing

The Mobilenet models utilize Batchnorm layers after their convolution layers. To be able to correctly quantize the weights we merge them with the previous convolution layers. This process is called folding. First we change the model description. Each Batchnorm layer are removed. Then the next and previous layers each Batchnorm layer was connected to, are connected with each other. Lastly we substitute the weights values of the previous convolution layer with the result of the linear transformation performed to the weights by the Batchnorm layer.

7.4.5 Performing k-means

We used the scipy python library [41] to perform k-means on each layer's flattened vector of weights. K-means was performed on the weights of both convolution and fully connected layers when present. Codebook sizes ranges from 8 bits to 1 bit. The weights initialy contain floating point numbers but after clustering contain integer indexes and the codebook contains the floating point centroids.

Generally the initialization of the clusters in k-means in conjunction with the distribution of original values affects the end cluster values. In this case the weight distribution of each layer of the models is for the most part gaussian and can be seen in Fig. 7.6. Since the end cluster values affect the accuracy of the quantized model we performed and compared k-means with linear, gaussian and inverse logarithmic centroid initialization.

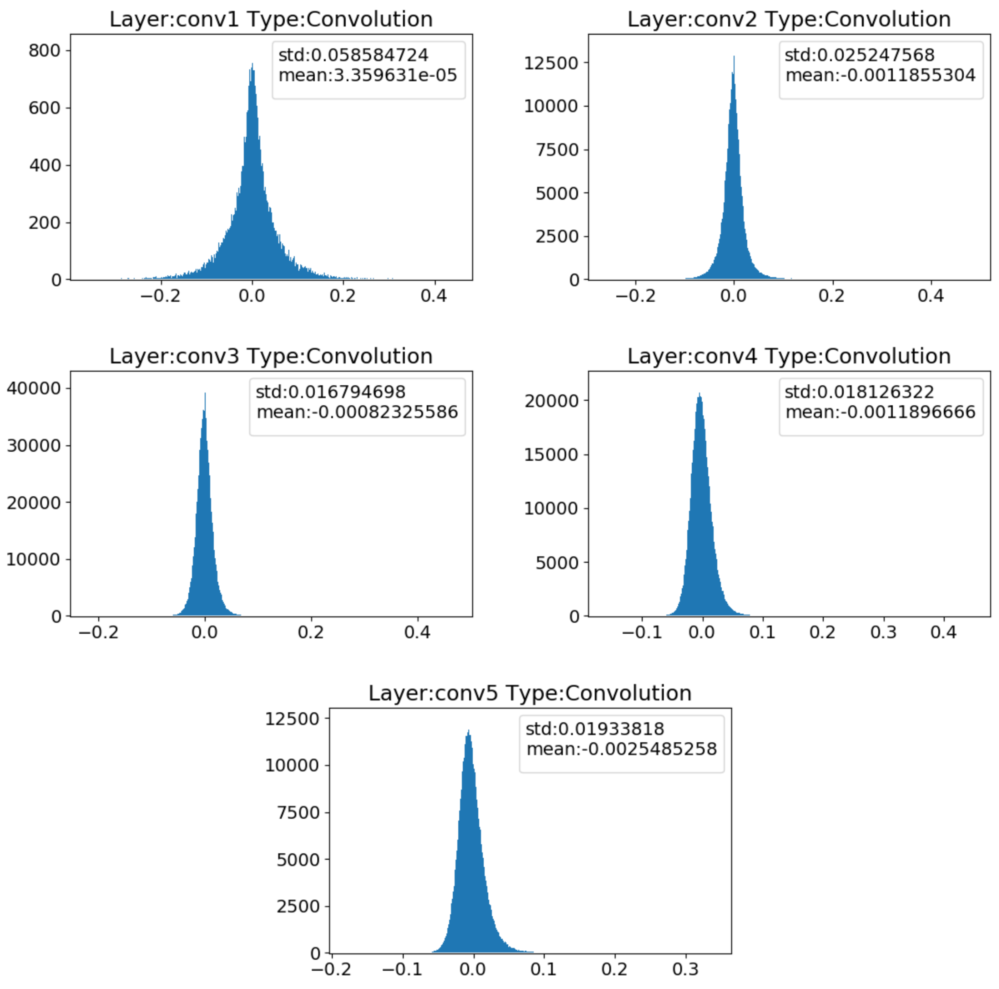
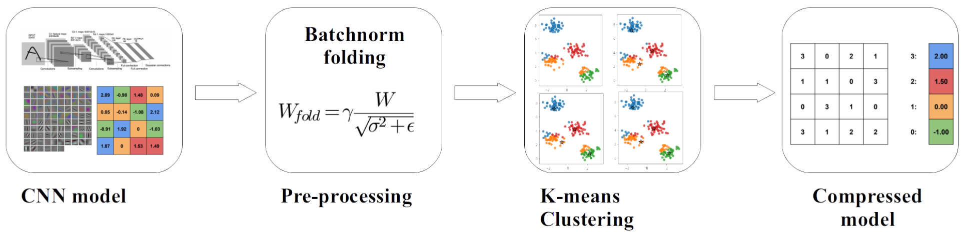


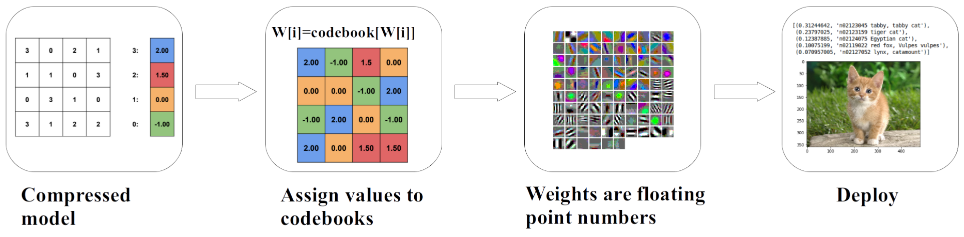
Figure 7.6: Weight distributions of AlexNet convolution layer

7.4.6 Evaluating accuracy

Caffe utilizes a General Matrix Multiply (GEMM) function to perform the convolution operation. In caffe there is no GEMM function that supports clustering weights. Thus convolution with cluster weights is not possible as is. Before inference the weights that now contain indexes to the codebook have to be replaced with the codebook values they point to. The whole pipeline for clustering the weights and then performing accuracy measurments can be seen in Fig. 7.7 (b). To have an accurate measurement we measure each accuracy 100 times.



(a) Compression steps


Compressed model
W[i]=codebook[W[i]]
Assign values to codebooks
Weights are floating point numbers
Deploy

(b) Inference steps

Figure 7.7: Quantization pipeline (a) For compression. (b) For inference

7.4.7 Computer specifications

The specifications of the machines used to run the experiments on can be seen in Tables 7.1 and 7.2.

Table 7.1: Machine specifications.

Computer	OS	CPU	RAM	GPU
Ideapad 510-ikb	Ubuntu 16.04	i7-7500U	8GB	GTX-940MX

Table 7.2: CPU specifications.

CPU	architecture	speed	cores-threads	L1	L2	L3
i7-7500U	x86-64	3.5GHz	2-4	32KB	256KB	4MB

7.5 Evaluation

From Fig.7.8 we observe that for AlexNet and GoogleNet the effect of the various initializations to the model accuracy is negligible. For both Mobilenets we observe that no method is consistently better than the others across the plots but for every x-axis value an initialization method is usually better. Both above observations make choosing one type of k-means initializations for the best possible accuracy, inconclusive. This is probably due to the low dimensionality of the k-means clustering, that in this case is performed in 1 dimension. The accuracy of the models starts to drop significantly on the same codebook width regardless of initialization, so we choose the linear initialization model to use from this point onward.

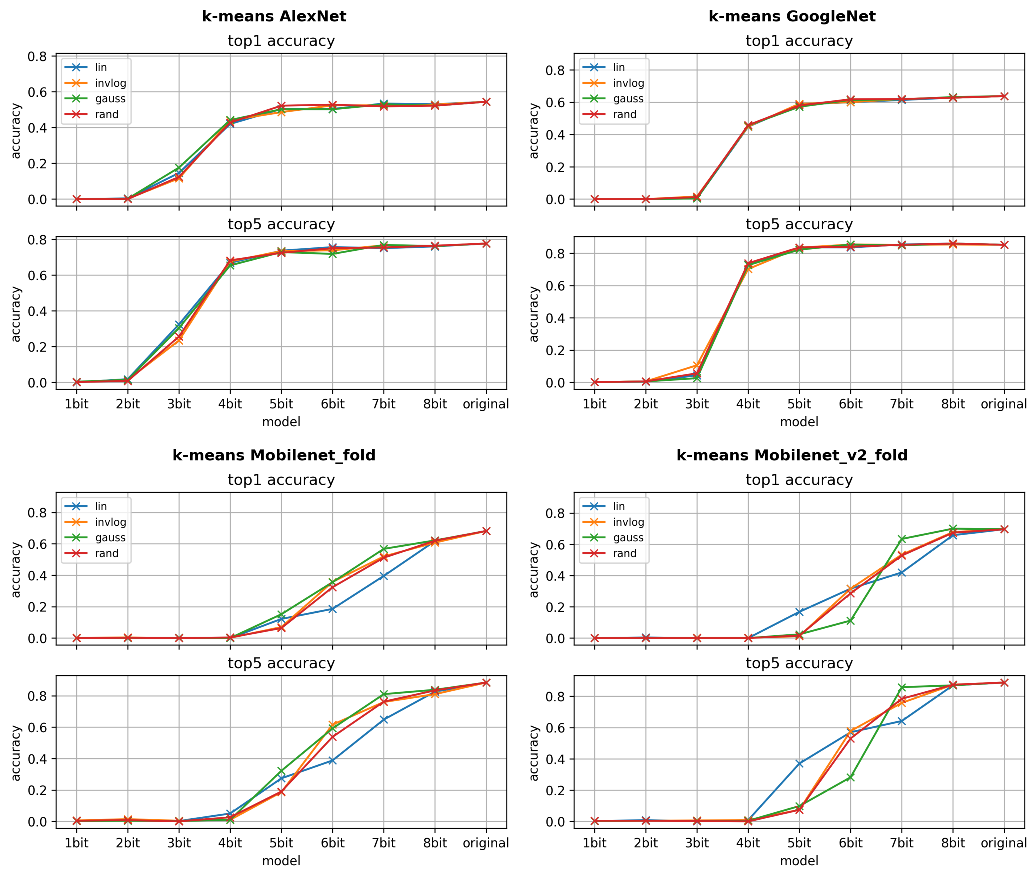


Figure 7.8: kmeans effect on accuracy with linear, random, gaussian and inverse logarithmic centroid initializations

Then observe the accuracy of the all the compressed models versus the original model in Fig. 7.9. We observe that larger models (Googlenet and AlexNet), that have a lot of redundant weights, can be compressed more before their accuracy deteriorates significantly. They can go as low as 5 bit quantization before any noticeable effect in accuracy is observed. In comparison the accuracy of Mobilenets suffers immediately after using anything lower than 8 bit quantization. If fine tuning is also introduced the accuracy of the quantized models can recover up to some point as mentioned by Han et al. [18].

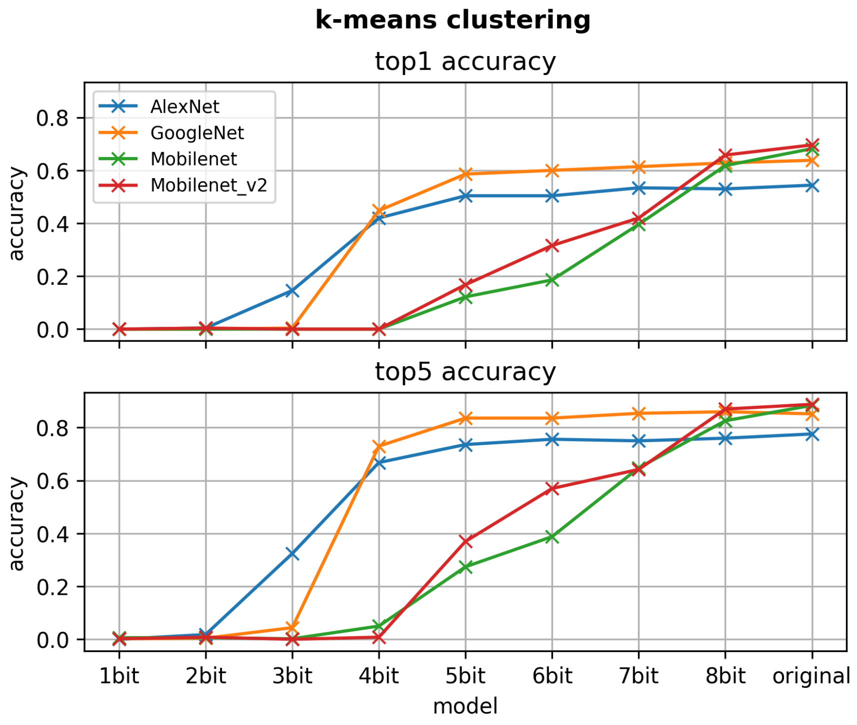


Figure 7.9: Accuracy vs bits used for clustering

In our experiment we are using the same number of bits across all layers. As a result Eq. 7.3 also calculates the compression rate for the whole model. At Table 7.3 using Eq. 7.3 and the results of Fig. 7.9, we calculate the maximum compression for each model using the smallest codebook size that does not affect its accuracy, for 32bit floating point numbers.

Table 7.3: Compression of models while retaining their accuracy.

Model name	Original size	Final size	Compression (theoretical)	Codebook size
AlexNet	233MB	36.4MB	6.4	5bits
GoogleNet	51MB	7.9MB	6.4	5bits
Mobilenet	17MB	4.25MB	4	8bits
Mobilenet_v2	14MB	3.5MB	4	8bits

However due to the smallest addressable unit of memory being 1 byte the effective compression is lower. Essentialy the smallest a weight can be is 8bits if it is a uint8 type. One workaround for this limitation could be s ub-byte indexing by storing and extracting many sub-byte values inside a uint8 type utilizing bit manipulation. Due to the extra complexity, accessing each value that way requires a number of shifts and other logic operations and as a result is going to be much slower and thus is not a viable solution. In our implementation it was at least 2 times slower. A viable solution would be having support for data types less than 1 byte which would be handled by dedicated hardware (e.g. asic or fpga) for accelerating the accesses.

Accuracy drops as the codebook size get smaller and the weight values diverge more and more from the originals. This can be visualized in Fig. 7.10 where the weights of the first convolution layer of AlexNet where plotted. We chose the particular filters due to them being easily visualized as an image since they have 3 channels.

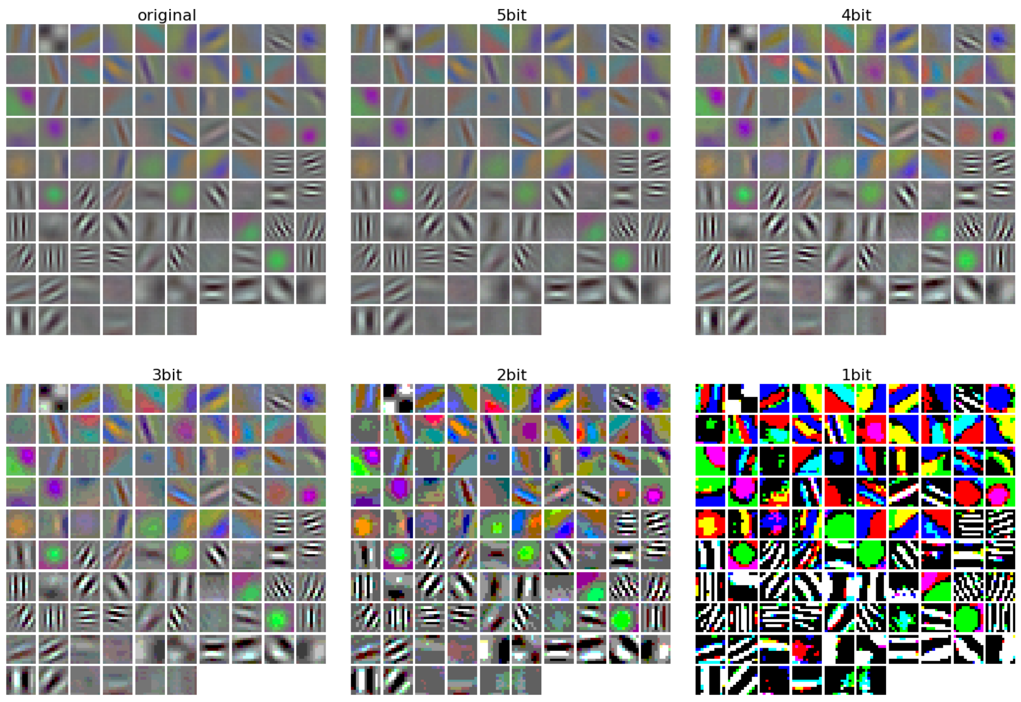


Figure 7.10: Visualization of the effect of weight quantization on the first convolution layer of AlexNet

Chapter 8

Optimization of Inference for Quantized Models

In this chapter, we develop convolutional layer implementations to accompany the quantization approach proposed in the previous chapter. Clustering introduces indirect references for accessing the codebooks. This is a common problem for compressed indexed representations in general, e.g CSR format for Sparse matrices [16] [12]. In order to access the real value of each weight, which is stored in the codebook, we first have to access the index stored in the weight matrix, and then find the codebook entry that the index points to. Accessing a matrix in the above manner requires a lot of pointer chasing and makes the memory accesses irregular. The indirect references makes the memory accesses from unit strides to random. Since the weight values do not occupy consecutive memory addresses the compiler optimizations and vectorization are also limited and that has a penalty on performance.

In this chapter we first investigate how performance optimizations targeting convolutions and indexed representations affect the performance of clustering convolutions. The goal is to find such optimization techniques to offset the performance penalties mentioned above. We start by hiding the latency of indirect references by optimizing other aspects of the code and then we eliminate the indirect references altogether.

8.1 Padding

In literature there are mainly two approaches for implementing padding on a convolution. The first is to check if the indexes convolution elements are out of bounds and in that case skip them(this is performed with the if statement of Alg. 9), and the second is to physically resize the input before the convolution. With the input padded the if statement is omitted and the input coordinates are recalculated in Alg. 10

Algorithm 9 Convolution with soft padding

```
1:  for $i = 0$ to $Cin - 1$ do
2:    for $j = 0$ to $Cout - 1$ do
3:      for $k = 0$ to $H_o - 1$ do
4:        for $l = 0$ to $W_o - 1$ do
5:          for $m = 0$ to $H_k - 1$ do
6:            for $n = 0$ to $W_k - 1$ do
7:              $k_{in} = stride \times k + m - pad$
8:              $l_{in} = stride \times l + n - pad$
9:              if $0 \leq k_{in} < H_i$ and $0 \leq l_{in} < W_i$ then
10:               $Output[j][k][l] += Input[i][k_{in}][l_{in}] \times Kernel[i][j][m][n]$
```

Algorithm 10 Convolution with physical (hard) padding

```
1:  for $i = 0$ to $Cin - 1$ do
2:    for $j = 0$ to $Cout - 1$ do
3:      for $k = 0$ to $H_o - 1$ do
4:        for $l = 0$ to $W_o - 1$ do
5:          for $m = 0$ to $H_k - 1$ do
6:            for $n = 0$ to $W_k - 1$ do
7:              $k_{in} = stride \times k + m$
8:              $l_{in} = stride \times l + n$
9:              $Output[j][k][l] += Input_{pad}[i][k_{in}][l_{in}] \times Kernel[i][j][m][n]$
```

8.2 Loop order

The naive convolution loop (Alg. 8) is not very efficient. That is why modern deep learning frameworks manipulate the data to transform the convolution operation in a matrix multiplication which can utilize the variety of efficient GEMM algorithms that exist. To achieve that the data has to be flattened to 2 dimensional matrices beforehand using the im2col operation. This approach introduces additional memory requirements since the data necessary of each output element have to be copied and the flattened matrices end up with duplicates of each input element. Moreover the performance of the GEMMs themselves is below their best achievable performance. That is because, to utilize the full performance potential of the GEMM, the common dimension between the two input matrices has to be smaller than the dimensions of the output matrix. That is generally not the case in convolutions where the input matrix is reshaped to $(H_f \times W_f \times C_i) \times (H_o \times W_o)$ and the kernel matrix to $(C_o) \times (H_f \times W_f \times C_i)$ and their common dimension $(H_f \times W_f \times C_i)$ is usually a lot larger.

Zhang et al. [56] address the above problem by proposing an optimization to the vanilla convolution loop to make it a viable alternative to GEMM functions. The optimizations focus on increasing cache locality the SIMD vectorization potential and the Fused Multiply-Add (FMA) unit utilization of the convolution. The later is important because the convolution operation is dominated by multiply adds as seen in Alg. 8 and therefore can benefit a lot from FMA instructions. Also in new processors many of these instructions can be performed at the same time using SIMD vectorization. The optimization is performed by reordering both the convolution loop and the order of the dimensions of the input, output, and kernel matrix. The innermost dimension and loop are now the output channel, followed by the input channel and then the matrix dimensions(Alg. 11). This ensures the maximum number of elements loaded for one convolution, occupying consecutive memory locations. Since the innermost dimension is the output channel, there are also no dependencies between consecutive elements. That way the Fused Multiply-Add (FMA) units and SIMD vector registers are used more efficiently. The downside to this optimization is that the order of the dimensions of the input, output and kernel matrices needs to be changed for the whole model.

The FMA units perform a multiplication and an addition at the same time, and the vector registers allow for multiple data to be computed in a single instruction. Intel's version of the above can be seen in Fig. 8.1. The FMA instruction is faster than performing the multiplications and additions separately. The accuracy of the computation is also greater. When multiplying two n-bit numbers the result can be up to 2n bits. To store the result to a n-bit register a rounding is performed so some accuracy is lost. This loss of accuracy increases the more operations are performed. When using FMA there is less accuracy loss because only one rounding is performed to the end result. On the contrary doing a separate multiplication and addition introduces a rounding after every computation (e.g. when computing the expression a*b+c the result will be round(round(a * b) + c) without FMA and round(a * b + c) with FMA).

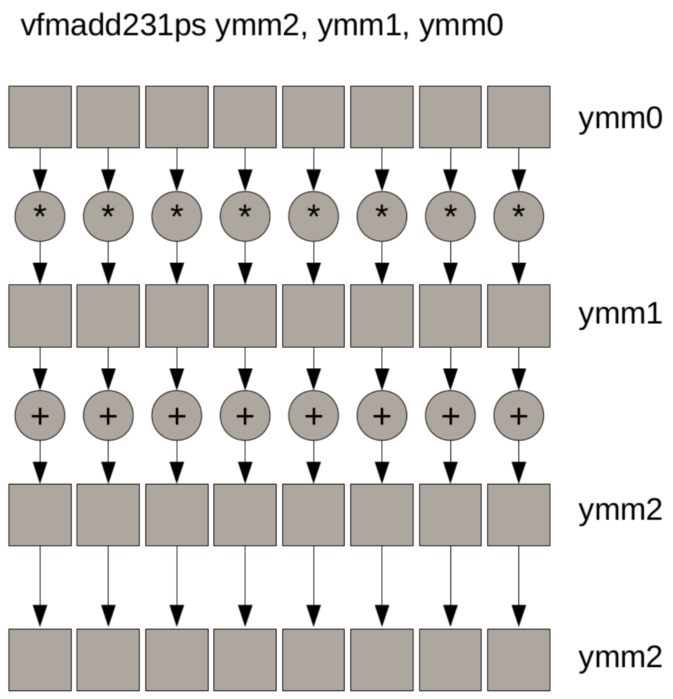


Figure 8.1: 8 fused multiply adds using Intel vfmadd instruction with 256bit vector registers

Algorithm 11 Reordered Convolution

```
1: for $k = 0$ to $H_o - 1$ do
2:  for $m = 0$ to $H_k - 1$ do
3:   for $n = 0$ to $W_k - 1$ do
4:    for $i = 0$ to $Cin - 1$ do
5:     for $l = 0$ to $W_o - 1$ do
6:      for $j = 0$ to $Cout - 1$ do
7:       $k_{in} = stride \times k + m$
8:       $l_{in} = stride \times l + n$
9:       $Output[k][l[j] += Input[i][k_{in}][l_{in}] \times Kernel[m][n][i][j]$
```

8.3 Blocking

As mentioned in the previous section the convolution algorithm is not very efficient. Georganas et al. [14] proposes register blocking in the output dimensions in order to achieve better data reuse. Zhang et al. [56] also propose a blocking variant of the reordered convolution in order to achieve better data locality as well. In both cases the target is the same as in the previous section, efficient 'cache friendly' data reuse and optimized deployment of SIMD FMA instructions.

Inspired from the above works we implement various blocking convolution algorithms. On algorithm 12 we have applied register blocking in the spatial output dimensions H_o and W_o. Then in algorithm 13 we also use blocking on the input and output channels. In this case the block size is equal to the target machine's vector length. Lastly in algorithm 14 we combine blocking with loop reordering and we perform blocking on the the input and output channels as well as the output dimension W_o.

Blocking is performed to make sure that the working set is small enough and therefore cache friendly. This can decrease cache traffic, improve data reuse and performance. It also makes sure that the resources for the SIMD FMA instructions are optimally used. The appropriate size of each block is determined by a lot of factors such as the cache size, the SIMD vector length as well as the latency of the FMA instructions. Block size affects how many independent elements can be computed by the FMA units. Zhang et al. [56] state that in order to hide the latency L_{fma} of the FMA instructions there have to be at least $N_{vec}L_{fma}$ output elements available to be issued in each cycle. This way one FMA result can be produced on each cycle for each of the N_{fma} FMA units. However only $N_{vec}N_{regs}$ element can be stored at vector registers. Having any more elements available will cause register spilling into memory and this will hinder performance. Considering the above, the upper and lower bounds of the number $\mathcal{E}$ of independent output elements that have to be computed in each cycle in order to reach the maximum attainable performance were expressed mathematically by Zhang et al. [56].

$$N_{vec}N_{fma}L_{fma} \leq \mathcal{E} \leq N_{vec}N_{regs} \tag{8.1}$$

We have to keep in mind that the above describes the limits for the non clustering convolution. Due to the irregular access patterns, more load, store and scatter/gather SIMD instructions have to be used and they need extra registers for the mask and vector of addresses. While the exact amount of registers available for storing the results and perform consecutive FMA operations as efficiently as possible will depend on the compiler and implementation, we expect the upper bound of Eq. 8.1 to be lower than what Zhang et al. expect.

Algorithm 12 Convolution with blocking on spatial dimensions

```
1: $H_{ob} = H_o/Blocksize_H$
2: $W_{ob} = W_o/Blocksize_W$
3: for $i = 0$ to $Cin - 1$ do
4:  for $j = 0$ to $Cout - 1$ do
5:   for $k_b = 0$ to $H_o - 1$ step $H_{ob}$ do
6:    for $l_b = 0$ to $W_o - 1$ step $W_{ob}$ do
7:     for $m = 0$ to $H_k - 1$ do
8:      for $n = 0$ to $W_k - 1$ do
9:       for $k = k_b$ to $H_{ob}$ do
10:        for $l = l_b$ to $W_{ob}$ do
11:         $k_{in} = stride \times k + m$
12:         $l_{in} = stride \times l + n$
13:         $Output[j][k][l] += Input_{pad}[i][k_{in}][l_{in}] \times Kernel[i][j][m][n]$
```

Algorithm 13 Convolution with blocking on spatial dimensions and channels

```
1: $Cin_b = Cin/N_{vec}$
2: $Cout_b = Cout/N_{vec}$
3: $H_{ob} = H_o/Blocksize_H$
4: $W_{ob} = W_o/Blocksize_W$
5: for $i_b = 0$ to $Cin - 1$ step $Cin_b$ do
6:  for $j_b = 0$ to $Cout - 1$ step $Cout_b$ do
7:   for $k_b = 0$ to $H_o - 1$ step $H_{ob}$ do
8:    for $l_b = 0$ to $W_o - 1$ step $W_{ob}$ do
9:     for $m = 0$ to $H_k - 1$ do
10:     for $= 0$ to $W_k - 1$ do
11:      for $i = i_b$ to $N_{vec}$ do
12:       for $j = j_b$ to $N_{vec}$ do
13:        for $k = k_b$ to $H_{ob}$ do
14:         for $l = l_b$ to $W_{ob}$ do
15:          $k_{in} = stride \times k + m$
16:          $l_{in} = stride \times l + n$
17:          $Output[j][k][l] += Input_{pad}[i][k_{in}][l_{in}] \times Kernel[i][j][m][n]$
```

Algorithm 14 Reordered convolution with blocking

```
1: $Cin_b = Cin/Blocksize_{Ci}$
2: $Cout_b = Cout/Blocksize_{Co}$
3: $W_{ob} = W_o/Blocksize_W$
4: for $j_b = 0$ to $Cout - 1$ step $Cout_b$ do
5:  for $i_b = 0$ to $Cin - 1$ step $Cin_b$ do
6:   for $k = 0$ to $H_o - 1$ do
7:    for $l = 0$ to $W_o - 1$ step $W_{ob}$ do
8:     for $m = 0$ to $H_k - 1$ do
9:      for $n = 0$ to $W_k - 1$ do
10:       for $i = i_b$ to $Cin_b$ do
11:        for $l = l_b$ to $W_{ob}$ do
12:         for $j = j_b$ to $Cout_b$ do
13:          $k_{in} = stride \times k + m$
14:          $l_{in} = stride \times l + n$
15:          $Output[k][l[j] += Input[i][k_{in}][l_{in}] \times Kernel[m][n][i][j]$
```

8.4 Just In Time Compilation

There are cases where the performance of some code implementations is tied to parameters which are only known at runtime. If these values are known beforehand the code can be optimized specifically for these parameters and this usually leads to better performance. In such scenarios some form Just In Time (JIT) compilation can be utilized to optimize the code after it was begun executing. JIT compilation compiles part of the code after the program starts and produces specialized code that then runs on the fly. The code is optimized for the specific parameters of that run.

In our case the code to be optimized is the convolution function and the parameters are the dimensions and values of matrices which are calculated from the arguments when the program begins. Before the convolution function is called there is a configuration step. At that time the dimensions of the input, output, kernel and codebook matrices calculated, memory is allocated for them and then they are initialized. When the configuration step is over for a particular instance of the program, all the parameters stay the same besides input and output values. At this point we can use the information of those parameters to generate and execute optimized code at runtime.

In order to use JIT compilation to optimize part of the convolution function at runtime we expermented with two JIT frameworks that we deemed suitable for our needs.Then we adressed thier drawbacks by creating our own runtime code generation and execution scheme.

8.4.1 Asmjit framework

Yilmaz et al. [53] use an assembly JIT framework, the asmjit project [27] to generate specific multiplication instructions at runtime. They propose and evaluate various optimizations of the Sparse Matrix Vector Multiplication. The sparse matrices are stored using the CSR (Compressed Sparse Row) format. This is similar to how in our case the data are compressed using codebooks. The similarity lies to the fact they both require indirect references to access the values of the matrix for their respective operations because one matrix's elements are indexes for another matrix. To counter the effects of indirect indexing they substitute indirect references and denote common factors between multiplications and end up with less multiplication operations.

Doing a similar analysis in our particular problem we observe that each output convolution element can be written as in Eq. 8.2 . In that formula the number of the codebook matrix elements is a few powers of two but the for kernel elements much more.

$$out[j][k][l] = \sum_{i=1}^{C_{in}} \sum_{m=1}^{W_k} \sum_{n=1}^{H_k} codebook[kernel[i][j][m][n]] \times Input[i][k_{in}][l_{in}] \tag{8.2}$$

In Eq. 8.3 we unfold the sums and replace the kernels with their values. For simplicity we will use the 1D equivalent index for matrices. Input indexes are arbitary. Then Eq. 8.4 in we can have each codebook as a common factor between the input values it is multiplied with.

$$\begin{aligned} out[j][k][l] = {} & codebook[0] \times Input[1] + codebook[3] \times Input[2]+ \\ & + codebook[0] \times Input[6] \\ & + codebook[2] \times Input[10] \\ & + ... \end{aligned} \tag{8.3}$$

$$\begin{aligned} out[j][k][l] = {} & codebook[0] \times (Input[1] + Input[6] + ...) \\ & + codebook[1](Input[9] + ..) \\ & + ... \\ & + codebook[2^{bits}](Input[14] + ..) \end{aligned} \tag{8.4}$$

To regroup Eq. 8.4 back to a sum we assume a mapping of which specific input elements are being multiplied with each codebook element so that we can add them together and perform only one multiplication. Such mapping can be implemented as an index table that contains lists of input indexes m of size 2^{bits}. Eq. 8.2 then is transformed to Eq. 8.5.

$$out[j][k][l] = \sum_{i=0}^{2^{bits}-1} \left(codebook[i] \times \sum_{j=m[i].start}^{m[i].end} Input[j] \right) \tag{8.5}$$

This limits the total number multiplications that have to be performed. Also the codebook elements are now referenced directly. To achieve better data locality the channel dimension can be kept and only the 2-D kernels be unfolded.

$$out[j][k][l] = \sum_{k=1}^{C_{in}} \sum_{i=0}^{2^{bits}-1} \left(codebook[i] \times \sum_{j=m[k][i].start}^{m[k][i].end} Input[j] \right) \tag{8.6}$$

To implement this approach we utilize JIT compilation, more specifically the asmjit project [27], which is a framework that provides an API to generate and run x86-64 assembly code during runtime. For mapping codebooks with input indexes we used Boost library's bimaps [5]. For the multiplications we use Intel's single fused multiply add instructions. The code generated for computing the value of one output element would look like this.

```
//add every input mapped to codebook[0]
xorps xmm1, xmm1 //sum=0
vaddss xmm1, xmm1, [rcx+4] //sum+=input[1]
vaddss xmm1, xmm1, [rcx+32] //sum+=input[8]
...
vfmadd231ss xmm2, xmm1, [rdx] //out[index]+=codebook[0]*sum
//add every input mapped to codebook[1]
xorps xmm1, xmm1 //sum=0
vaddss xmm1, xmm1, [rcx+8] //sum+=input[2]
vaddss xmm1, xmm1, [rcx+52] //sum+=input[13]
...
//perform scalar fused multiply add
vfmadd231ss xmm2, xmm1,[rdx+4] //out[index]+=codebook[1]*sum
...
//repeat for all codebook elements
```

To further boost the performance of the generated code, the next step was to vectorize the computations using SIMD instructions. The CPU this program was tested on supported the AVX2 x86-64 instruction set. The widest vector registers AVX2 architecture supports are 256bit. The fma instruction, now packed, it is performed to all elements of the vector registers as seen in Fig. 8.1. This way 8 consecutive 32bit floating point output elements are computed simultaneously.

An example of such generated assembly instructions can be seen below.

```
//add every input mapped to codebook[0]
xorps xmm0, xmm0 //make sum 0
vaddss xmm0, xmm0, [rcx+128]
vaddss xmm0, xmm0, [rcx+60]
...
xorps xmm1, xmm1 //make sum 0
vaddss xmm1, xmm1, [rcx+100]
vaddss xmm1, xmm1, [rcx+44]
...
xorps xmm2, xmm2 //make sum 0
vaddss xmm2, xmm2, [rcx+28]
vaddss xmm2, xmm2, [rcx+40]
...
xorps xmm7, xmm7 //make sum 0
```

```
vaddss xmm7, xmm7, [rcx+88]
...
//fill 256bit register with all sums
vinsertps xmm9, xmm0, xmm1, 16
vinsertps xmm10, xmm4, xmm5, 16
vinsertps xmm11, xmm9, xmm2, 32
vinsertps xmm12, xmm10, xmm6, 32
vinsertps xmm13, xmm11, xmm3, 48
vinsertps xmm14, xmm12, xmm7, 48
vinsertf128 ymm0, ymm13, xmm14, 1
//fill codebook to 256bit register
vmovss xmm3, [rdx+48]
vinsertps xmm1, xmm3, xmm3, 16
vinsertps xmm2, xmm1, xmm3, 32
vinsertps xmm4, xmm2, xmm3, 48
vinsertf128 ymm1, ymm4, xmm4, 1
//load 256bit register with
vmovdqu ymm2, [rax]
//perform packed fused multiply add
vfmadd231ps ymm2, ymm1, ymm0
//save 8 output elements
vmovdqu [rax], ymm2
...
//repeat for all codebook elements
```

8.4.2 Easy Jit framework

We also used the easy-jit[3] framework to optimize the code. Easy jit performs just in time compilation by calling LLVMs clang compiler during runtime as a plugin. It takes as input a function and its known arguments and then provides a function pointer of the optimized function.

Normaly the convolution function would be called like this:

```
conv(c_in, rows, cols, c_out, kRows, kCols, pad, stride, \
    &in, &codebook, &kernel, &out);
```

All arguments except the inputs and outputs are already known before calling the function. The function is optimized by compiling the already known arguments and a function pointer to the optimized function is returned. Then the optimized function can be called using the rest of the arguments.

```
auto conv_opt = easy::jit(conv, c_in, rows, cols, \
                c_out,kRows,  kCols, pad, stride, _1 , \
                &codebook, &kernel, _2);
conv_opt(&in,&out);
```

This framework provides a ”hassle free” way to compile known parameters into functions. It is a ”plug and play” framework with complicated compilation concepts hidden. Although good at what it does, it does not allow changes in the source code of the function. Also it faces problems optimizing functions with struct type arguements or return values.

There are options for levels of performance and code size optimizations but tweaking individual compiler options is not implemented. SIMD vectorization is supposed to be performed since the SLP and Loop vectorizer are used [37] when applicable. The SLP vectorizer combines multiple scalars into vectors and the Loop Vectorizer vectorizes instructions in loops to perform multiple same instructions of consecutive iterations [38].

To confirm that vectorization occurs we used the debug options of the framework to dump the optimized function's LLVM's intermediate representation and found FMA vector operations (Lst. 8.1).

```
cat easyjitdump.txt|grep fmul

...
  %208 = fmul reassoc nsz arcp contract <8 x float> %193, %204
  %209 = fmul reassoc nsz arcp contract <8 x float> %193, %207
  %230 = fmul reassoc nsz arcp contract <8 x float> %193, %226
  %231 = fmul reassoc nsz arcp contract <8 x float> %193, %229
  %259 = fmul reassoc nsz arcp contract <8 x float> %252, %258
  %260 = fmul reassoc nsz arcp contract <8 x float> %255, %258
...
```

Listing 8.1: Easyjit's intermediate representation dump containing vector FMA commands.

8.4.3 Generating C++ code at runtime

The previous JIT frameworks were used to improve of the cluster convolution algorithm. Unfortunately they both have their limitations in terms of easiness to use, freedom on emitting code and memory overheads which are thoroughly described in the evaluation section. Some key traits that they lack but we require when generating code at runtime are:

- generating a C++ function at runtime
- memory lightweight just in time compilation
- control over the compilation parameters
- CPU architecture independent
- Compiler independent

We developed a new scheme where C++ functions are generated as a string, saved to a temporary file, compiled and then run, all during runtime in what is functionaly JIT compilation. The process is performed in the following steps:

Step 1 Export the build's compilation flags to a file when configuring the build with CMake

```
file(WRITE build/cxxflags.txt ${CMAKE_CXX_FLAGS})
```

Step 2 Write C++ code and then save it to a temporary .cpp file

```
stringstream program;
program << "#include <cmath>\n";
        << "extern \"C\" int lib_func(,"
        << "float** input, float** output)\n";
...
ofstream out( "tmpfiles/tmp.cpp" );
out << program.str();
out.close();
```

Step 3 Invoke the compiler compiler with the appropriate flags and compile the file to a library

```
stringstream cmd;
cmd << "${CXX} -fPIC -shared" << cxxflags.c_str()
    << " -o tmpfiles/libtmp.so tmpfiles/tmp.cpp";
system(cmd.str().c_str());
```

Step 4 Load the library with dlopen and use dlsym to access the desired function(s).

```
void* handle = dlopen("libtmp.so",RTLD_LAZY);
void (*func_print_name)(float*,float*);
*(void**)(&func_ptr) = dlsym(handle, "lib_func");
```

Step 5 Run the desired function(s)

```
func_ptr(&input,&kernel,&output);
```

The above functionality was implemented in two C++ classes, one responsible for creating and compiling the C++ file and the other responsible for loading the library and the functions. By using a temporary file to store the code before compilation this scheme overcomes the main problems we have with the previous two JIT frameworks.

8.4.4 Generating convolutions from template files

To save time and reuse code, in step 2 of the previous section, we extend our code generation scheme to use user defined templates. We provide template text files as skeletons for functions. These template files also contain hooks, special words, which we substitute with desired code on runtime and then produce and compile the C++ files. The entire process for generating and executing code at runtime can be seen in Fig. 8.2. An example of the generating code process can be seen on the following code snippets.

```
StringCompile sc;

sc.append("../templates/conv_loop.tmp");
sc.replace("$c_in",std::to_string(c_in));
sc.replace("$c_out",std::to_string(c_out));
sc.replace("$outrows",std::to_string(outrows));
sc.replace("$outcols",std::to_string(outcols));
sc.replace("$krows",std::to_string(krows));
sc.replace("$kcols",std::to_string(kcols));
sc.replace("$stride",std::to_string(stride));
sc.replace("$pad",std::to_string(pad));
sc.replace("$rows",std::to_string(rows));
sc.replace("$cols",std::to_string(cols));
sc.save();
```

Listing 8.2: Replacing hooks with parameters

```
extern "C" int conv()
{
int ki,ko,i,j,m,n,ii,jj;
int out_addr,k_addr,in_addr;
for (ki=0; ki<$c_in; ki++){
 for (ko=0; ko<$c_out; ko++){
  for(i=0; i < $outrows; i++){
   for(j=0; j < $outcols; j++){
    for(m=0; m < $krows; m++){
     for(n=0; n < $kcols; n++){

     jj = $stride*j + n- $pad;
     ii = $stride*i + m - $pad;
     out_addr=(ko*$outrows+i)*$outcols+j
    ;
     in_addr=(ki*$rows+ii)*$cols+jj;
     k_addr=((ko*$c_in+ki)*$krows+m)*
    $kcols+n;
     (*out)[out_addr]+=(*in)[in_addr]*
     (*code)[(*kern)[k_addr]];

     }
    }
   }
  }
 }
}
return 0;
}
```

Listing 8.3: Original template

```
extern "C" int conv()
{
int ki,ko,i,j,m,n,ii,jj;
int out_addr,k_addr,in_addr;
for (ki=0; ki<96; ki++){
 for (ko=0; ko<256; ko++){
  for(i=0; i < 23; i++){
   for(j=0; j < 23; j++){
    for(m=0; m < 5; m++){
     for(n=0; n < 5; n++){

     jj = 1*j + n- 0;
     ii = 1*i + m - 0;
     out_addr=(ko*23+i)*23+j;

     in_addr=(ki*27+ii)*27+jj;
     k_addr=((ko*96+ki)*5+m)*5+n;

     (*out)[out_addr]+=(*in)[in_addr]*
     (*code)[(*kern)[k_addr]];

     }
    }
   }
  }
 }
}
return 0;
}
```

Listing 8.4: Generated file with hooks replaced

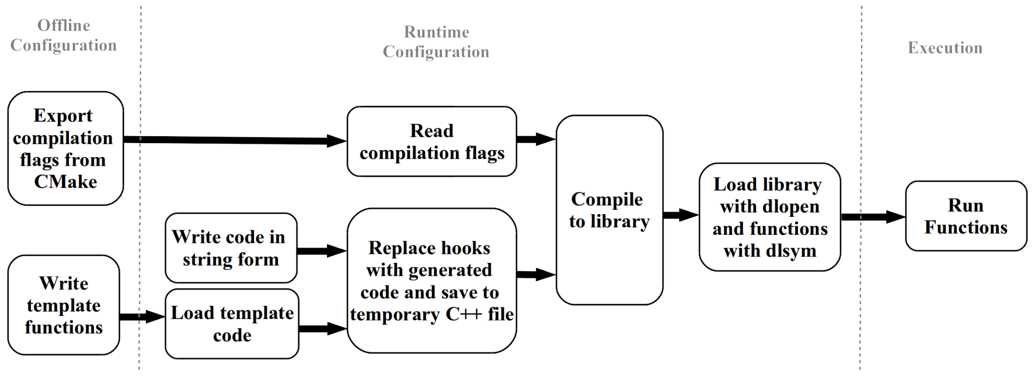

Offline Configuration
Runtime Configuration
Execution
Export compilation flags from CMake
Read compilation flags
Compile to library
Load library with dlopen and functions with dlsym
Run Functions
Write code in string form
Replace hooks with generated code and save to temporary C++ file
Write template functions
Load template code

Figure 8.2: Code generation pipeline

8.5 Loop unrolling

Vanhoucke et al. [46] propose loop unrolling to optimize the convolution loop. This technique improves performance by reducing the overhead of checking for loop termination and making the code more vectorizable. In addition they also propose using parallel accumulators for each unrolled computation in order to give the compiler more opportunities to perform optimizations. Loop unrolling is an optimization that is usually performed to some extent by the compiler itself, but code generation gives us the freedom to completely unroll the two innermost loops of the convolution operation. These two techniques can be seen bellow:

```
    (*out)[out_addr]+=(*code)[(*kern)[k_addr+0]]*(*in)[in_addr+0];
    (*out)[out_addr]+=(*code)[(*kern)[k_addr+1]]*(*in)[in_addr+1];
    (*out)[out_addr]+=(*code)[(*kern)[k_addr+2]]*(*in)[in_addr+2];
    (*out)[out_addr]+=(*code)[(*kern)[k_addr+3]]*(*in)[in_addr+29];
    (*out)[out_addr]+=(*code)[(*kern)[k_addr+4]]*(*in)[in_addr+30];
    (*out)[out_addr]+=(*code)[(*kern)[k_addr+5]]*(*in)[in_addr+31];
    (*out)[out_addr]+=(*code)[(*kern)[k_addr+6]]*(*in)[in_addr+58];
    (*out)[out_addr]+=(*code)[(*kern)[k_addr+7]]*(*in)[in_addr+59];
    (*out)[out_addr]+=(*code)[(*kern)[k_addr+8]]*(*in)[in_addr+60];
```

Listing 8.5: Loop unrolling

```
    int ps0=(*code)[(*kern)[k_addr+0]]*(*in)[in_addr+0];
    int ps1=(*code)[(*kern)[k_addr+1]]*(*in)[in_addr+1];
    int ps2=(*code)[(*kern)[k_addr+2]]*(*in)[in_addr+2];
    int ps3=(*code)[(*kern)[k_addr+3]]*(*in)[in_addr+29];
    int ps4=(*code)[(*kern)[k_addr+4]]*(*in)[in_addr+30];
    int ps5=(*code)[(*kern)[k_addr+5]]*(*in)[in_addr+31];
    int ps6=(*code)[(*kern)[k_addr+6]]*(*in)[in_addr+58];
    int ps7=(*code)[(*kern)[k_addr+7]]*(*in)[in_addr+59];
    int ps8=(*code)[(*kern)[k_addr+8]]*(*in)[in_addr+60];
    (*out)[out_addr]+=ps0+ps1+ps2+ps3+ps4+ps5+ps6+ps7+ps8;
```

Listing 8.6: Loop unrolling with parallel accumulators

8.6 Unique filters

After quantization there is the possibility of repeating 2d-kernels existing across the channels. If the number of unique ones is low enough, they can be shared between channels to further reduce the size of the model. The above kernel repetition and appearance of unique filters shared among channels is similar to clustering using kernels as the clustering unit, performed by Son et al. [43] and Yu et al. [54]. In their case each codebook entry is a unique filter. Given the promising results of the above works, even if he have a lot of unique kernels at first, a smaller number could be enforced by utilizing their clustering methods to end up with less unique kernels.

If we generate one function per filter during runtime with our code generating scheme, it can even make the computational burden of the above clustering smaller because it can increase the number of repeating kernels even more. Due to our code generation scheme we have the versatility to bypass the indirect references by unfolding them at configuration time and directly printing the values of each codebook. We also have the freedom to perform factorization like Eq. 8.5 to avoid multiplications. This removes the problems introduced by clustering but will increase the compilation time and end executable size proportionally to the number of unique filters.

```
void Unique1(...){
    for(int i=0; i < 27; i++){
      for(int j=0; j < 27; j++){
        int out_idx=i*27+j;
        int in_idx=i*29+j*1;
        (*out)[out_addr+out_idx]+=(0x1.9f65f8p+7f)*((*in)[(ki*29+2)*29+0+in_idx])+
            (0x1.43e0f2p+7f)*((*in)[(ki*29+1)*29+1+in_idx])+
            (0x1.be70bep-31f)*((*in)[(ki*29+2)*29+2+in_idx])+
            (0x1.ca9ba8p+6f)*((*in)[(ki*29+0)*29+2+in_idx])+
            (0x1.d92f4p-124f)*((*in)[(ki*29+1)*29+2+in_idx])+
            (0x1.aa90b4p+7f)*((*in)[(ki*29+0)*29+0+in_idx])+
            (0x1.791524p+7f)*((*in)[(ki*29+1)*29+0+in_idx])+
            (0x1.9a244ap+7f)*((*in)[(ki*29+0)*29+1+in_idx]+(*in)[(ki*29+2)*29+1+
    in_idx]);
        }
    }
}
```

Listing 8.7: Filter function with factorized multiplications. Codebooks are printed as std::hexfloats to avoid rounding errors

Each filter function need to perform the convolution operation for the channels that share that filter. This channel information needs to be provided to the function. We handle this in two ways:

- The shared channels are hardcoded on the function as a 2d matrix.

```
    void Unique1(...){
      int channels[3][2]={{2,0},{2,2},{4,1}};
        for(int c=0;c<3;c++){
          ko=channels[c][0];
          ki=channels[c][1];
            ...
          }
    }
```

  Then we need to call each unique function, stored in a vector, only once.

```
    for(const auto& fnit: map.fvec){
        fnit(&in,&out,&kernel,&codebook,0,0);
    }
```

- Channel information stored in a hash table and accessed via a loop or an iterator. The function is called for each channel that shares the filter.
  - Loop

```
for (int ko=0; ko<c_out; ko++){
    for (int ki=0; ki<c_in; ki++){
        int addr=(c_in*ko+ki)*kRows*kCols;
        map.fmap[addr](&in,&out,&kernel,&codebook,ko,ki);
        }
}
```

  - Iterator

```
for(const auto& fnit: map.fmap){
    int quot=(fnit.first/(kRows*kCols))/c_in;
    int rem=(fnit.first/(kRows*kCols))%c_in;
    fnit.second(&in,&out,&kernel,&codebook,quot,rem);
}
```

We benchmark the above accesses and we perform multiplications normaly and using Eq. 8.5. This produces a total of six different implementations for us to test.

8.7 Experimental Setup

8.7.1 Computer specifications

The specifications of the machines used to run the experiments on can be seen in Tables 8.1 and 8.2.

Table 8.1: Machine specifications.

Type	Computer	OS	CPU	RAM
Server		Ubuntu 18.04	2x Xeon Gold 5218	314GB
Desktop	Ideapad 510-15ikb	Ubuntu 16.04	i7-7500U	8GB
Edge	Raspberry Pi 3 B+	Ubuntu 18.04	Cortex A53	1GB

Table 8.2: CPU specifications.

CPU	arch	speed	cores/ threads	L1,L2,L3	SIMD
Xeon Gold 5218	x86-64	3.9GHz	14-28	32K, 1M, 22M	avx-512
i7-7500U	x86-64	3.5GHz	2-4	32K, 256K, 4M	avx2
Arm Cortex A53	aarch64	1.4GHz	4-4	16K, 512K	neon

8.7.2 Test benchmarks

As it is common in such benchmarks we first test on the convolution sizes of AlexNet layers to measure a real convolutional network (Table 8.3). Then we test on a synthetic benchmark Table 8.4 to observe the effects of clustering as the layers become narrower and deeper. For the blocking impementations we will skip the last two test cases due to the dimensions being non divisible by the blocksize. Lastly Table 8.5 has two layer sizes for each machine configuration. On the first the working set fits in the last level cache and on the other it does not. That way we can measure the effect cache have on the convolution. The working set consists of the input, output, kernel, and codebook matrices.

Table 8.3: Benchmark with AlexNet convolutions.

	C_{in}	H_{in}	W_{in}	C_{out}	H_k	W_k	stride	pad
conv1	3	227	227	96	11	11	4	no
conv2	96	27	27	256	5	5	1	yes
conv3	256	13	13	384	3	3	1	yes
conv4	384	13	13	384	3	3	1	yes
conv5	384	13	13	256	3	3	1	yes

Table 8.4: Benchmark with large convolutions.

	C_{in}	H_{in}	W_{in}	C_{out}	H_k	W_k	stride	pad
bench1	256	128	128	384	3	3	1	yes
bench2	256	128	128	512	3	3	1	yes
bench3	384	128	128	384	3	3	1	yes
bench4	384	128	128	512	3	3	1	yes
bench5	384	112	112	512	3	3	1	yes
bench6	512	56	56	1024	3	3	1	yes
bench7	512	28	28	512	3	3	1	yes

Table 8.5: Benchmark targeting last level cache for each machine in Table 8.1

	C_{in}	H_{in}	W_{in}	C_{out}	H_k	W_k	stride	pad	memory
Server	512	32	32	512	3	3	1	yes	13MB
(22MB)	512	32	32	1024	3	3	1	yes	24MB
Desktop	128	32	32	256	3	3	1	yes	2.5MB
(4MB)	128	32	32	768	3	3	1	yes	7MB
Edge	64	16	16	128	3	3	1	yes	481KB
(512KB)	64	16	16	256	3	3	1	yes	609KB

8.7.3 Block size

Usually it is optimal for the fastest moving dimension to be a multiple of the vector length. That is more achievable on the reordered implementation because the innermost dimension, the output channel, is usually divisible by the vector length (they are both powers of two). Moreover that dimension it is usually big enough to be able to be divided with bigger multiples of the vector length.

In order to choose the correct block size for our blocking implementations need to take some parameters into account. From Eq 8.1 we can find the upper and lower bounds of the number of elements that need to be computed in each cycle in order for the FMA to be efficient. In our case the FMA latency is 4 cycles [13]. This gives us a rough estimate of what block sizes to try when experimenting (Table 8.6). Based on Table 8.6 we experiment with block size. The upper bound, especially on the Server machine, are quite big, sometimes bigger than the loop sizes, so we will have to use smaller block sizes to efficiently block our loops. This is more of a problem for spatial dimensions that are usually small and restrict our design. After some tweaks and experimenting the block size of each algorithm can be seen in Table 8.7.

Table 8.6: Element bounds for efficient FMA

	N_{vec}	N_{regs}	N_{fma}	bounds
Server	16	32	1	[64,512]
Desktop	8	16	2	[64,128]

Table 8.7: Block sizes for blocking

	H_b	W_b	Cin_b	$Cout_b$
Alg. 12	16	16	-	-
Alg. 13	8	16	N$_{vec}$	N_{vec}
Alg. 14	-	16	32	8*N_{vec}

8.7.4 Compilers

The choice of compiler plays also a role in optimizing the code and performing vectorization. For our benchmarks we will use a variety of compilers (Table 8.8) when needed. We will compare both the performance and the code produced by the compilers to observe differences in optimizations.

We use LLVM's Clang compiler which is needed for some of our implementations and then compare the results to with the other compilers. Also for some benchmarks it is usefull to compare results with the gcc compiler. Lastly, for x86-64 cpus we will also use Intel's proprietary compiler, icc, because sometimes it handles vectorization better and thus can potentially produce better optimized code.

Table 8.8: Compilers.

Type	Clang	GCC	ICC
Server	-	7.5.0	19.0.4.243
Desktop	6.0.0	5.5.0	19.0.4.243
Embedded	6.0.0	7.5.0	-

8.7.5 Vectorization

Since many of the optimizations rely on vectorization we make sure to include all the necessary compiler options for each compiler to generate vectorized code. For icc unsafe math is enabled by default, but for gcc and clang we have to specifically enable them. Icc also uses unique flags for AVX2 and AVX512 insructions. For AVX2 we use the -xCORE-AVX2 flag, but for the AVX-512 we go for the -xCOMMON-AVX512 flag which uses 512 register instructions at higher frequency than -xCORE-AVX512.

We also have to keep in mind that even though many newer embedded CPUs (like arm cortex a53) have vectorization capabilities there are still a lot that do not. For that reason we will also evaluate the benchmarks for the Edge machine with vectorization turned off. To sum up, for vectorization we use the flags as described in Table 8.9.

Table 8.9: Compiler options for vectorization.

Type	General Options, (x86-64 specific)
icc	-,(-xCORE-AVX2/-xCOMMON512)
gcc	-funsafe-math-optimization -ftree-vectorize, (-mavx2/512 -mfma)
clang	-funsafe-math-optimization -ftree-vectorize -ffp-contract=fast

8.7.6 Metrics

For performance metrics we will use Execution Time, Speedup and FLOPS.

- Execution Time is the time the convolution takes to run, after any configuration step.

- Speedup in this case measures is the relative performance of two implementations processing the same convolution. As a baseline time we use the time of the normal convolution (Alg. 10.)

$$Speedup = \frac{Time_{base}}{Time_{new}} \tag{8.7}$$

- FLOPS are the number floating point operations a processor performs in the unit of time. There are two floating point operations in the innermost loop of the convolution, one multiplication and one addition. Given that all the loops in the convolution increment by one, the total number of Floating point operations is calculated as follows:

$$FLOPs = 2H_{out}W_{out}C_{out}C_{in}H_kW_k \tag{8.8}$$

  And the floating point operations per second(FLOPS) are then calculated using FLOPs.

$$FLOPS = \frac{FLOPs}{execution\ time} \tag{8.9}$$

In our benchmarks, when a piece of code performs better that others it can be hypothesized that among other things, better vectorization and FMA utilization has taken place. But in order to confirm that the above optimizations happened we need find FMA instructions, vectorized or not, using the objdump GNU utility to disassemble the executable (Listing 8.8 and 8.9). The appearance of FMA instuctions confirms that the compiler used, at least performed that optimization. The appearance of vectorized FMA instructions confirms that vectorization has taken place. Other compiler optimizations may also take place such as efficient vector memory loads and stores, scatter/gather instructions, that also can lead to optimal or sub optimal utilization of FMA and SIMD instructions. When comparing versions of the same implementation built with different compilers the presence of above instructions are some aspects of the code we can look at in order to explain any performance variations.

```
objdump -d conv|grep 'vfmadd'

...
40155b: c4 c2 59 99 02        vfmadd132ss (%r10),%xmm4,%xmm0
401614: c4 a2 75 a8 04 10     vfmadd213ps (%rax,%r10,1),%ymm1,%ymm0
401672: c4 a2 49 99 04 83     vfmadd132ss (%rbx,%r8,4),%xmm6,%xmm0
4016a4: c4 a2 41 99 04 83     vfmadd132ss (%rbx,%r8,4),%xmm7,%xmm0
...
```

Listing 8.8: Part of x86-64 assembly code that contains fma instructions. The ones ending in ss are single and the ones ending in ps are vectorized

```
objdump -d conv|grep 'fmla\|fmadd'
...
  40189c: 1f002120  fmadd s0, s9, s0, s8
  4021d4: 4e21cc03  fmla  v3.4s, v0.4s, v1.4s
  4021d8: 4e22cc04  fmla  v4.4s, v0.4s, v2.4s
  402218: 1f010800  fmadd s0, s0, s1, s2
...
```

Listing 8.9: Part of arm (aarch64) assembly code that contains fma instructions. The fmadd are single and fmla are vector instructions

8.7.7 Implementation names

On the plots we annotate each implementation with a small string. To better follow the plots we present each implementation and the respective name in the plot legend.

Table 8.10: Implementations and names in legends

Implementation	Name	Implementation	Name
Vanilla convolution	v	Block reordered Cluster	bl-re-cl
Vanilla convolution (cluster)	v-cl	Asmjit	asmjit
Soft padding	sft	Asmjit Vectorized	asmjit-v
Soft padding cluster	sft-cl	Unique filters	unique
Reordered	re	Unique filters hardcoded	hard
Reordered cluster	re-cl	Unique filters iterator	iter
Block Spatial	bl-sp	Unique filters loop	loop
Block spatial cluster	bl-sp-cl	Codegenv vanilla	cgen
Block all dimensions	bl-a	Codegen unroll	cgen-unr
Block all dimensions cluster	bl-a-cl	im2col & GEMM	im2col
Block reordered	bl-re		

8.8 Evaluation

For the first part of the evaluations we will use the clang compiler in order to fairly compare with the implementations that require clang. Sometimes it will be needed to use other compilers. In any case all the compiler builds will be compared with each other on a separate section.

8.8.1 Codebook length

We investigate the effects codebook size has on the performance of the convolution by comparing the speed of the cluster convolution for 8bits to 1bit codebooks.

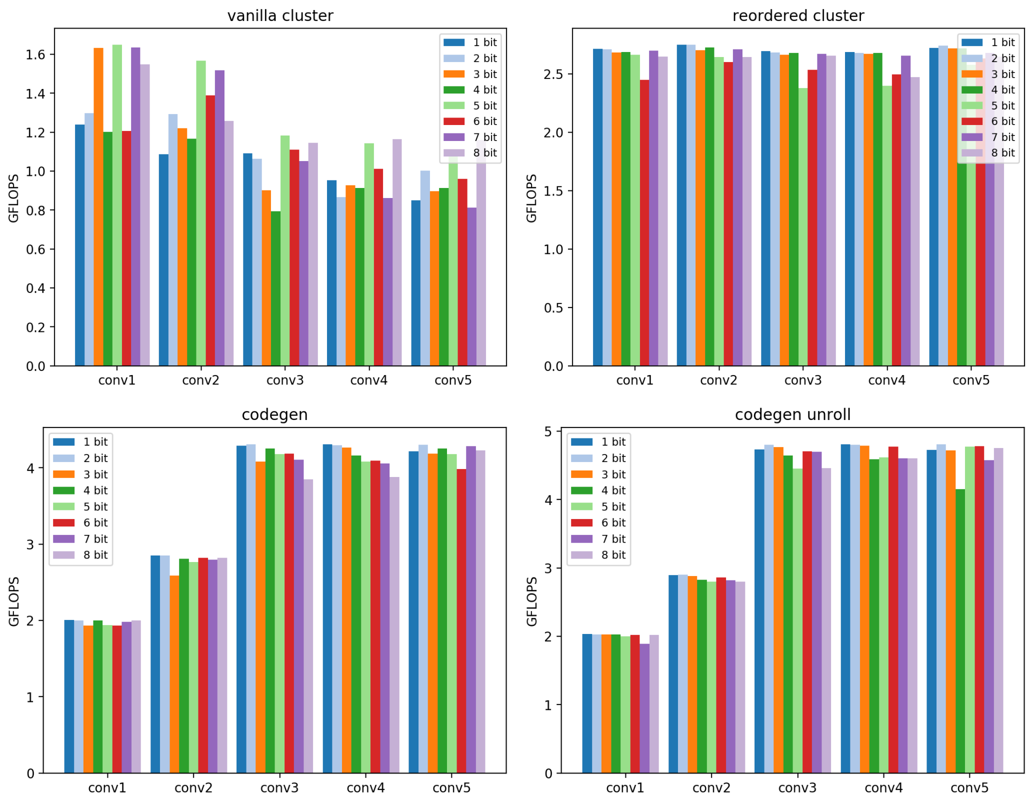


Figure 8.3: Comparing different codebook sizes for various clustering convolution implementations on Benchmark Table 8.3 on Desktop

In Fig. 8.3 we observe that codebook size does not affect the performance of the convolution on a consistent manner. This means that no codebook size is globaly optimal for performance. This can be attributed to the small size of the codebook compared to the working set. It is unlikely it will make a difference on whether the working set fits in the last level cache or not. For the rest of the experiments we will use 5 bit codebooks since this size was the optimal for accuracy for bigger models as seen in the previous chapter (Fig. 7.9).

8.8.2 Padding

We measure the performance of padding implemented with ”if” boundary check (soft padding) as in Alg. 8 and with real input padding (hard padding) as in Alg.10). We compare these two approaches for vanilla and cluster convolutions in Fig. 8.4

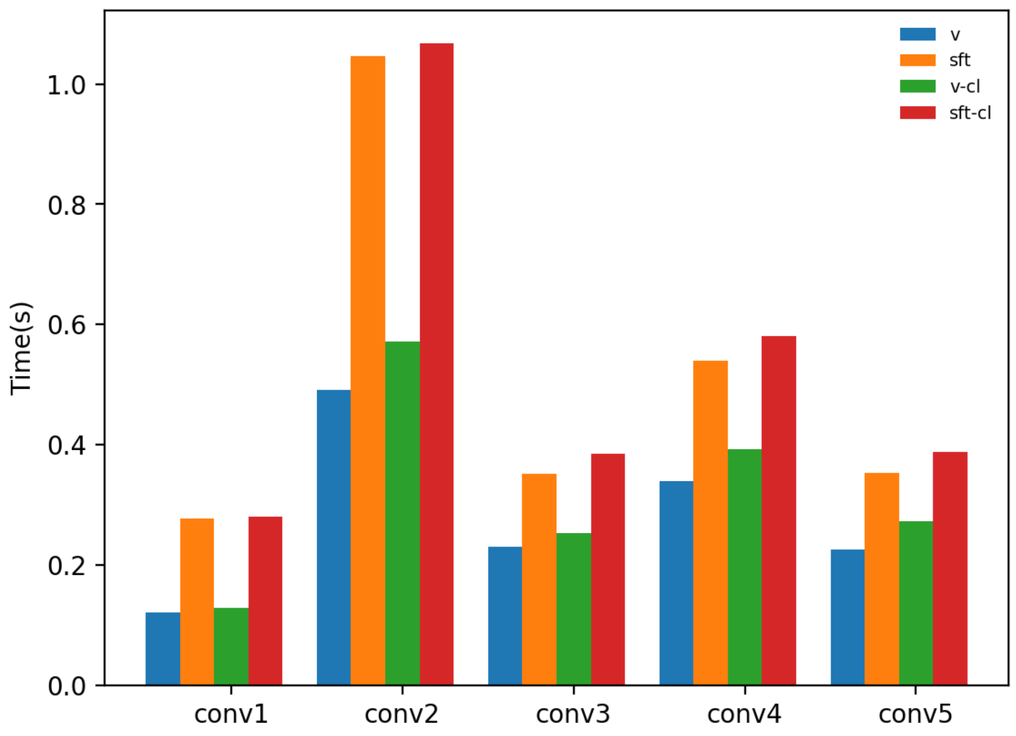
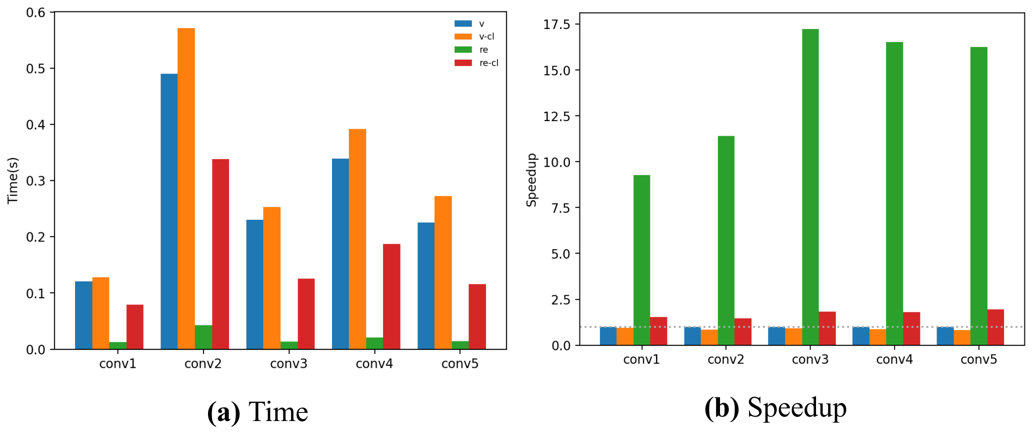


Figure 8.4: Comparing padding approaches on vanilla and clustering convolution implementations on Table 8.3 on Desktop

The hard padding method appears superior. The reason is that the soft padding method utilizes boundary checks on each iteration and this introduces time delays. Thus we will use the hard padding method for the rest of the experiments.

8.8.3 Reordered convolutions

We perform clustering on reordered convolutions as in Alg. 11. The reordering of convolutions maximizes their cache locality, FMA utilization and vectorization potential. From Fig.8.5 we can conclude that using reordered convolutions for clustering definitely has a positive impact on performance. But the clustering convolution does not improve as much as the vanilla convolution from reordering. The disparity between the two improvements is indicative of the impact clustering has on convolution performance. Clustering comes at a cost. Due to the indirect referencing weight values are not in consecutive memory addresses any more and as a result the vectorization potential and cache benefits are decreased.



(a) Time **(b)** Speedup

Figure 8.5: Effects of clustering in vanilla and reordered convolution on Table 8.3 on Desktop.

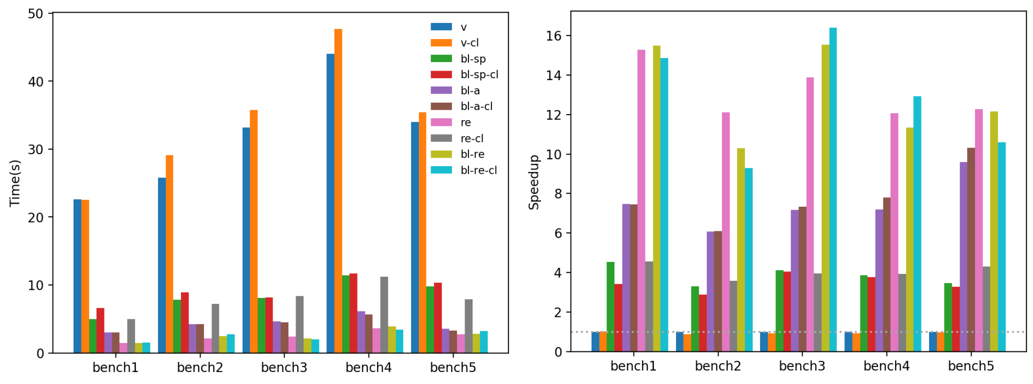
8.8.4 Blocking

For this section we will used the icc compiler for the x86-64 machines since it yielded the best results. As we expected blocking improves the convolution significantly on most cases. For the Server and Desktop machines an important observation is that the overhead that clustering introduces is not so apparent in the blocking implementations (Fig. 8.6 and 8.7). This is more evident in the reordered implementations and their blocking counterparts. The improvement from blocking is so big that the blocking reordered cluster convolution performs on the same level as the non cluster one.

As someone might expect blocking on the spatial dimensions only (Alg. 12) might have some benefits, but as previously mentioned the small size of such dimensions does not allow for bigger block size and that hinders bigger improvements. When we also perform blocking on the channels (Alg. 13) the performance increases. Lastly when the output channel is the innermost loop as in reordered implementation (Alg. 14) there is room for even more performance gain since we have bigger block size on the inner dimension. It is worth noting that as the number of channels increases Alg. 13 seem to catch up on Alg. 14 in performance.

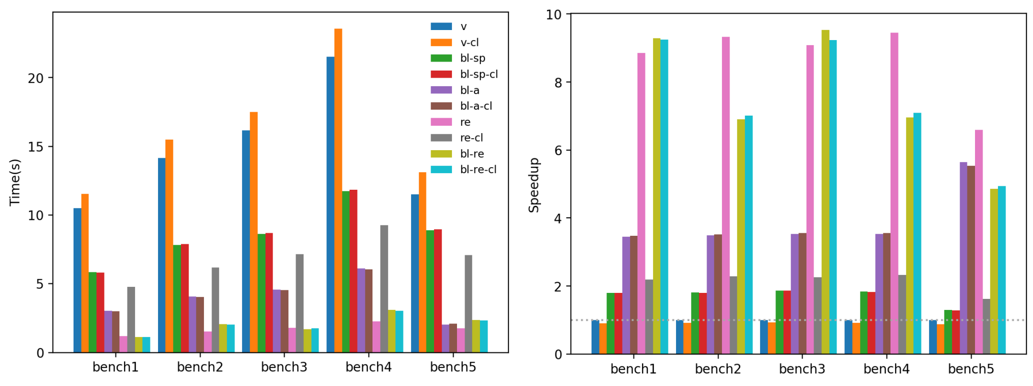
On the Edge device (Fig. 8.8) we see that on only Alg. 12 and 13 are beneficial to the performance of the cluster convolution but not to the same extent as in the Server and Desktop benchmarks. Contrary to Desktop and Server observations, for Alg. 14 the blocking and non-blocking cluster reordered convolution perform equally bad. This can be attributed to various factors such as the difference in compilers or the small vector width that hiders the vectorization potential.

From the above observations we can conclude that blocking in cluster convolutions improves performance due to data locality in some extent, but the main performance gains are tied to the better utilization of SIMD FMA instructions. On the Edge machine that has 128bit SIMD registers there is a moderate performance gain, on Desktop with 256bit there is a considerable gain and lastly on the Server with 512bit registers we observe the maximum performance gain.



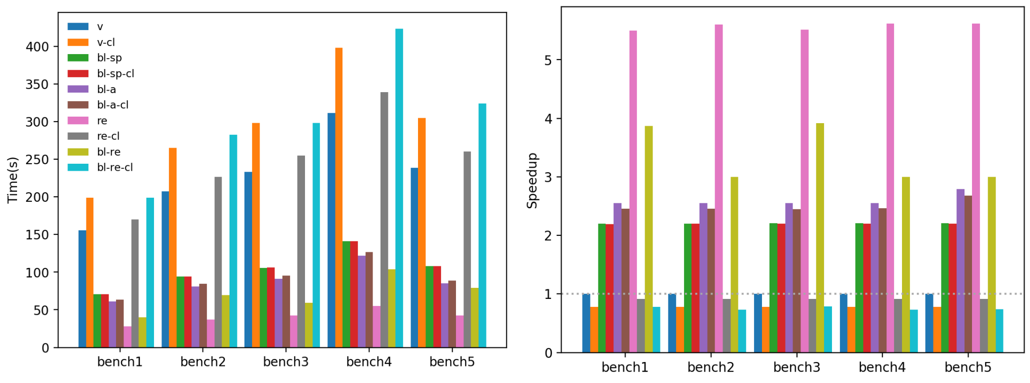
(a) Time **(b)** Speedup

Figure 8.6: Performance of blocking implementations and their non blocking counterparts on Table 8.4 on Desktop



(a) Time **(b)** Speedup

Figure 8.7: Performance of blocking implementations and their non blocking counterparts Table 8.4 on Server



(a) Time **(b)** Speedup

Figure 8.8: Performance of blocking implementations and their non blocking counterparts Table 8.4 on Edge

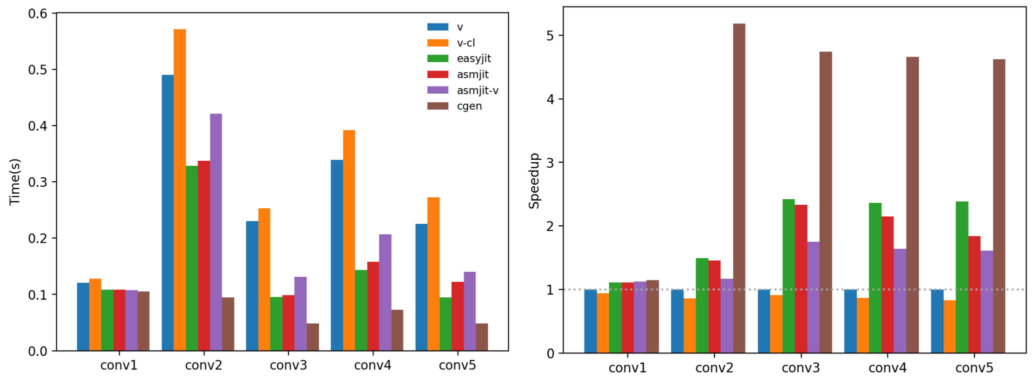
8.8.5 Comparision of Jit implementations

We first compare the asmjit and easyjit frameworks as well as our code generation against the clustered vanilla convolution. We use the convolution layers of AlexNet to test (Table 8.3). Since easyjit framework relies on clang, we will use it to be able to have a fair comparison with the other jit implementations. From Fig. 8.9 we can conlclude that the Asmjit framework has limited usefulness for our usage scenario. Firstly the code needed for the convolution is emitted directly to assembly instead of C or C++. This makes the process of actually generating the code for the convolution operation more perplexing. Additionally any code optimizations (vectorization unrolling etc.) must be performed by the users themselves, which will most likely be inferior to optimizations performed by a C++ compiler. We can see this in Fig. 8.9 where the vectorized version of the asmjit implementation actually performs worse than the non vectorized. Secondly the generated functions are stored to buffers with limited capacity. Using more buffers to overcome the capacity problem is also prohibitive because it ends up using too much RAM. For this reason asmjit framework will not be viable for larger benchmarks.

We do not gain any speed benefits by using the codebooks as common factors. This does not necessarily contradict the results of Yilmaz et al. [53]. In their case the common factor method was one of many methods, where the appropriate one was applied based on profiling. They also state that it was not producing the best results every time, only that when it did, it produced the best results by far. The improvement in performance could also be due to the sparsity of the matrix in general. Moreover unlike us they did not use vector assembly instructions. Regular multiplications have bigger latency than regular additions and thus by using the common factor trick they use less multiplications and achieve speed benefits. In our case however the vectorized fused multiply add instruction has the same latency with the vectorized add and multiply instructions [13]. This means that the benefit we gain in speed is limited and the sophisticated and time consuming process of finding common factors in order to perform Eq. 8.5 might not be worth it.

Easyjit, while being simpler and easier to use, also has its drawbacks. Firstly it can only optimize functions with arguments of C++ fundamental data types. Secondly only functions preexisting in the code could be passed for optimization. To unfold the weights and get rid of the indirect references, as well as to perform the convolution as in Eq. 8.5, we would need to be able to also generate the function code itself at runtime.

The naive implementation of code generation already adresses all of the above problems and that is evident by its better performance. As mentioned in subsection 8.4.4 functionally performs the same optimizations as easyjit. The reason it performs better in some cases could be that in code generation we can tweak specific compilation flags. With code generation the performance can be further increased, thus this is the JIT implementation we will be using from now on.



(a) Time **(b)** Speedup

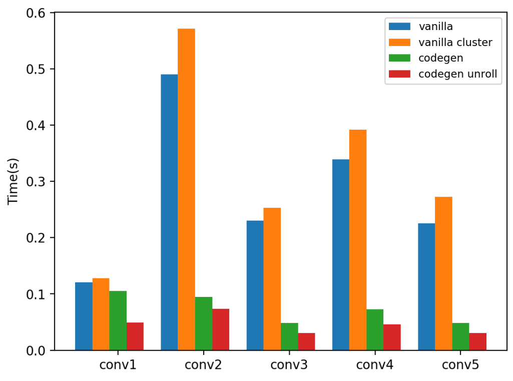
Figure 8.9: Performance of vanilla, cluster asmjit and easyjit implementations on Table 8.3 on Desktop

8.8.6 Loop unrolling

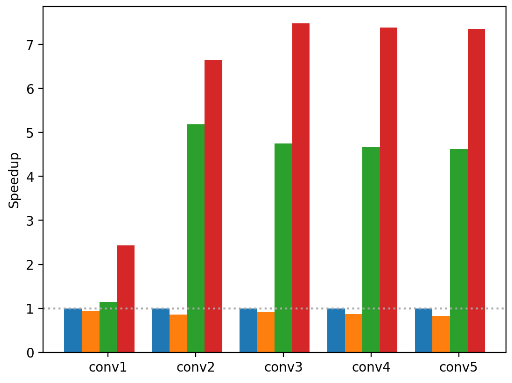
We compare the novel code generation implementation with its loop unrolling variant as described in section 8.5. Unrolling performs better on all the the benchmarks, for all the Desktop machine configurations (Fig. 8.10, 8.11 and 8.12)

From Fig. 8.10 we observe that the unrolling implementation is always better. Its performance is consistent besides the first convolution where the kernels are big (11x11) as well as the input (227x227) that produces too many independent output elements and leads to register spilling.

The naive code generation performs consistently besides the first convolution. On Fig. 8.10 the first convolution has the biggest input by far so there is no speedup, and on Fig. 8.12 as the input becomes smaller its performance increases. From the above we see a connection between its performance and the size of the input.

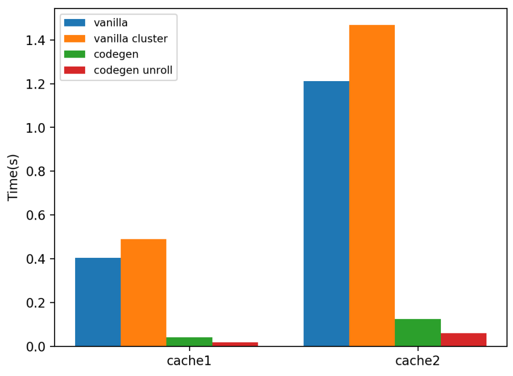


(a) Time

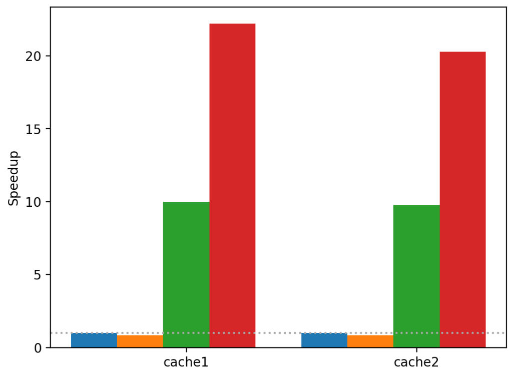


(b) Speedup

Figure 8.10: Comparing previous jit implementations with our code generation scheme on Table 8.3 on Desktop.

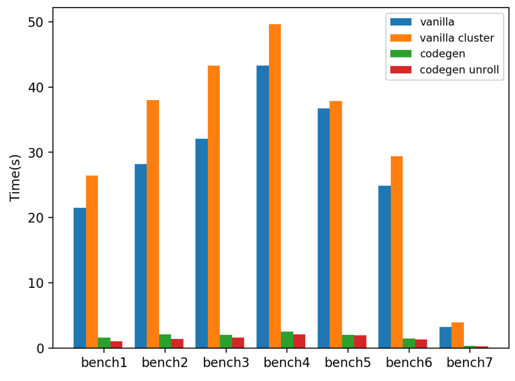


(a) Time

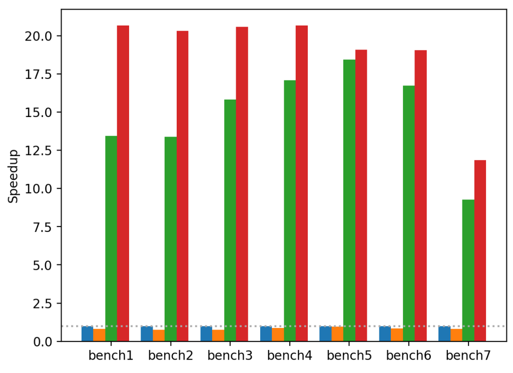


(b) Speedup

Figure 8.11: Comparing jit implementations with our code generation scheme on Table 8.5 on Desktop.



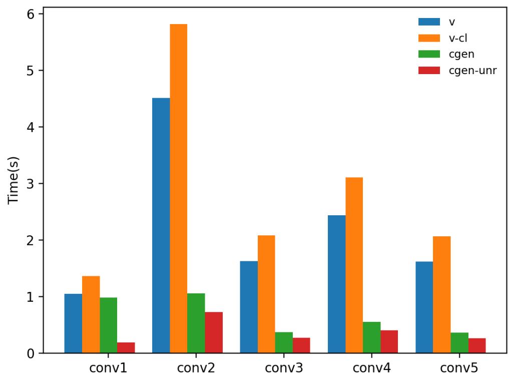
(a) Time



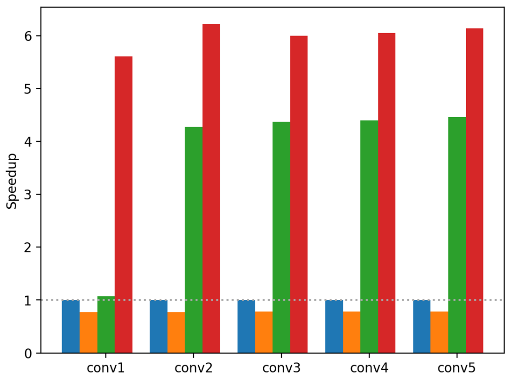
(b) Speedup

Figure 8.12: Comparing jit implementations with our code generation scheme on Table 8.4 on Desktop.

From Fig. 8.13, 8.14 and 8.15 we observe that for the Edge device the naive codegen improves the convolution a lot, and at some cases the unrolling code generation improves the performace by a larger margin. At Fig. 8.13 we can see that the unrolling is superior, at Fig. 8.14 they are equally good and at Fig. 8.15 unrolling performs worse that naive codegen. This can be attributed to the smaller size of the SIMD registers that the Edge machine has.

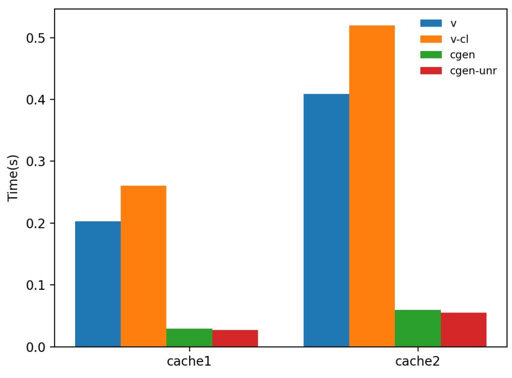


(a) Time

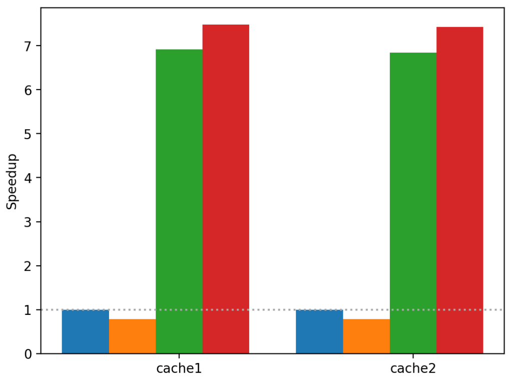


(b) Speedup

Figure 8.13: Comparing jit implementations with our code generation scheme on Table 8.3 on Edge.

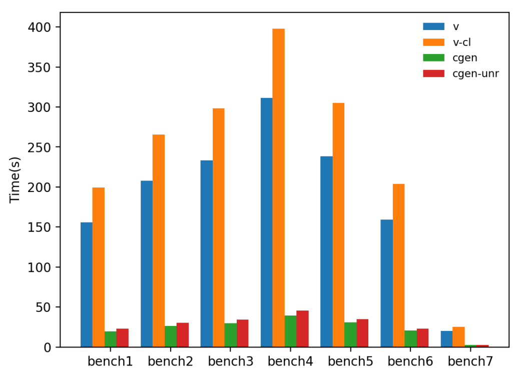


(a) Time

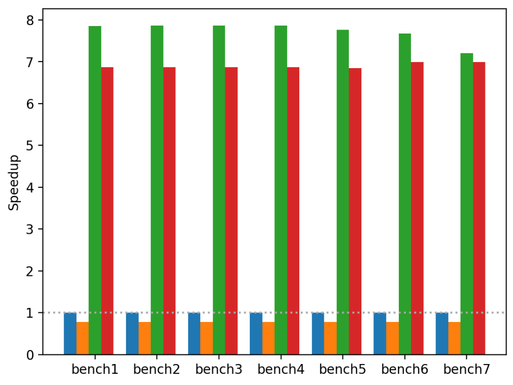


(b) Speedup

Figure 8.14: Comparing jit implementations with our code generation scheme on Table 8.5 on Edge.



(a) Time



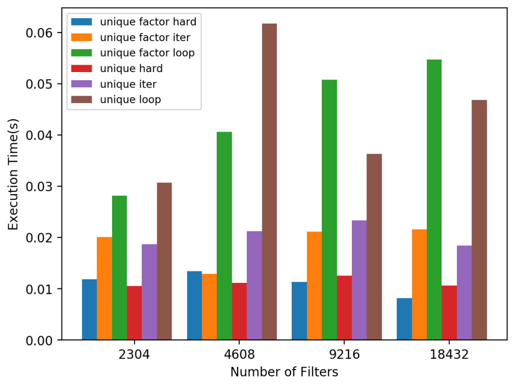
(b) Speedup

Figure 8.15: Comparing jit implementations with our code generation scheme on Table 8.4 on Edge.

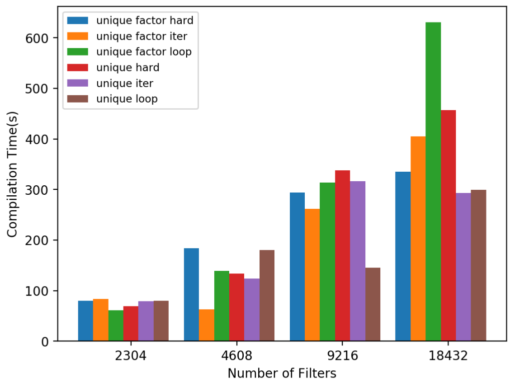
8.8.7 Unique Filters

Each unique filter is performed by its own function. As the number of filters increases we expect the compilation time to also increase. We plot the compilation time and execution time against the number of unique filters present. The compilation step is a heavy load for embedded machines with limited memory. That is why it was not possible to increase the number of filters as much as in the other implementations. The unique filter method can still be a viable solution if the compilation is performed offline for a given model, on a more powerful machine. Then only the compiled libraries are deployed. Since we have no indirect references and only a few kernel functions that can be optimized, we expect the performance to increase significantly from the vanilla convolution 8.17.

The code produced performed really well, but we found that the icc compiler produces the fastest code on x86 platforms and for this section we will use these results. This will be discussed more extensively on the compilers evaluation section. The differences of the performance of the different implementations on the Desktop are marginal. That being said the hardcoded implementations were consistently slightly faster than their respective counterparts. The factorized implementation performs slightly worse than the unfactorized which can be attributed to the same reasons discussed in the asmjit evaluation. For the above reasons to compare with the other optimizations we will be using the hard-coded non factorized variant. In Fig. 8.17, 8.18 and 8.19 it is evident how much this method improves the cluster convolution.



(a) Execution Time on Desktop icc



(b) Compilation Time on Desktop icc

Figure 8.16: Execution and compilation time vs the number of filters

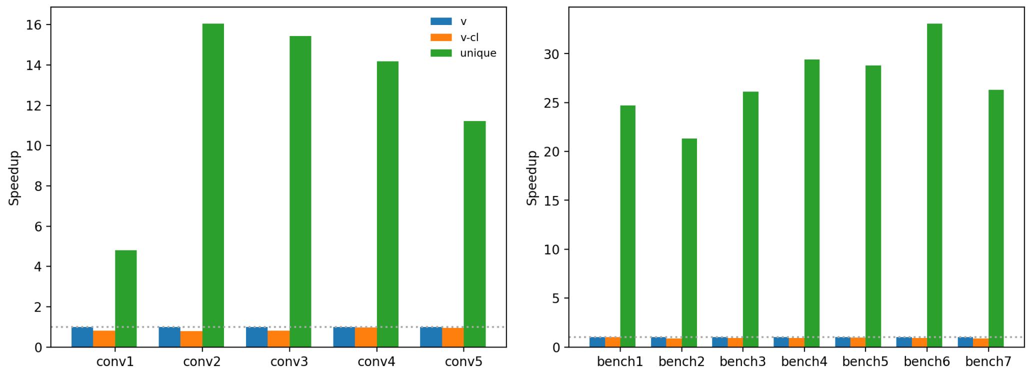


Figure 8.17: Comparison of unique filters on Table 8.3 and 8.4 on Desktop

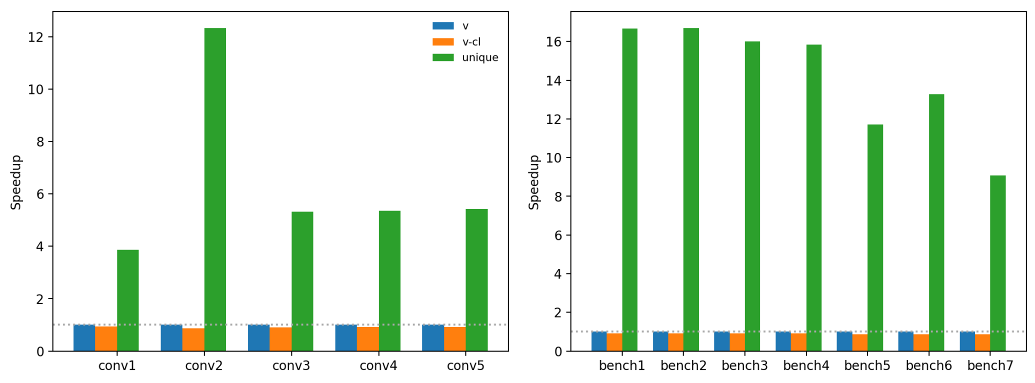


Figure 8.18: Comparison on Table 8.3 and 8.4 on Server

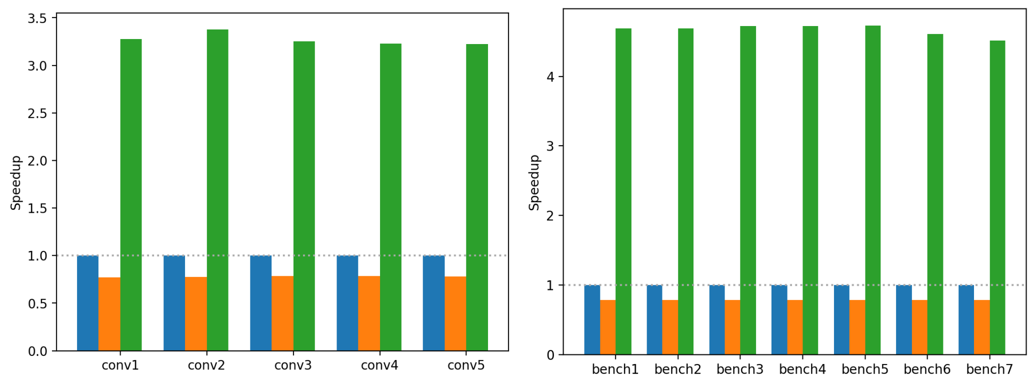


Figure 8.19: Comparison on Table 8.3 and 8.4 on Edge

8.8.8 Total comparison with all compilers

In this section we will evaluate the different convolution optimizations targeting cluster convolution and how they perform relative to each other depending on which compiler we used. We use convolution sizes from Table 8.4 since we do not observe any differences with the other benchmarks. While comparing the performance it is useful to take into account Tables 8.11, 8.12 and 8.13 in order to correlate performance and vectorization. On these tables we have documented whether we encountered single and packed (vectorized) FMA instructions when disassembling each implementation.

Our first observation is that the compilers are able to consistently vectorize the same implementations across all machines and that these implementations consistently performed better. Secondly every compiler in every platform was able to vectorize the unique filters and code generation implementations. Thus they consistently perform better than other implementations regardless of platform or compiler. From the above we can argue that they are the most versatile implementations for best performance.

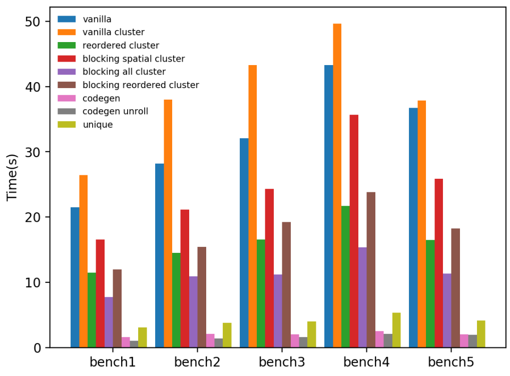
When looking each compiler individually and in contrast to the others we first oberve that Icc usually both vectorizes code more efficiently and vectorizes pieces of code that the other compilers tested here do not. Its usefulness lies to the fact that it can vectorize more complex code cases that the other compilers. In Table 8.11 we see that icc vectorizes all code cases while the other compilers vectorize some of them. The benefits of icc being able to vectorize the code can be direclty observed in the better performance of the respective convolution variants over the ones produced by the other compilers (Fig 8.20 and Fig 8.21). This is more evident when looking the performance of blocking implementations. With icc, they perform substantially better than the normal cluster convolution. When using clang, we observe a moderate improvement in performance. Lastly, when compiled with gcc, they yield just a slight performance boost. The performance gap is bigger on the blocked reordered implementation. When looking through its machine code we find that icc is using a lot of gather instructions that collect the indirect references of the codebooks into a vector to then perform the FMA with. We believe that this is the main contributing factor to its better performance against the other compilers.

Clang is not able to vectorize all clustering implementations like icc, but vectorizes more than gcc, notably blocking Alg. 12 and 13. For the best performing implementations clang seems to favor the normal and unrolled codegen implementations but surprisingly is not able to optimize the unique filter implementations as well as gcc and icc.

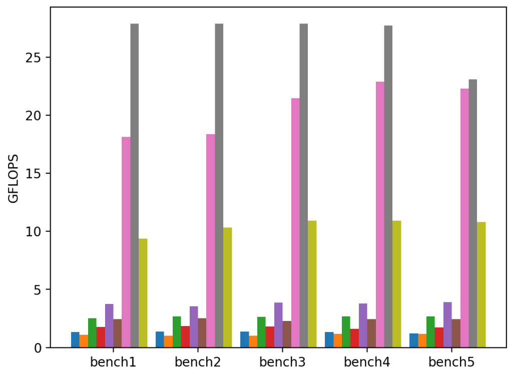
Code generated by gcc, besides the codegen and unique filter implementations, does not perform well compared to the other compilers. If we take a look at Tables 8.11, 8.12 and 8.13 we can see why. Gcc does not manage to vectorize any clustering implementation except for the the codegen and unique filter.

On the Edge device (Fig. 8.22) we observe that gcc does not handle blocking implementations very well and clang handles them moderately well which is the same trend as in Server and Desktop. We also observe that on the Edge machine even though FMA instructions are used, vectorization does not yield the same improvements as in the other machines. This can be attributted to a number of things. One possible reason could be that the vectorized instructions no being as optimized in performance as the x86-64 counterparts. It is worth noting that the SIMD registers of the Edge CPU have a vector length of only 4 compared to 8 and 16 of Desktop and Server. For that reason we expect it to be able to perform less operations in parallel. This claim is further reinforced by the fact that with clang the naive codegen implementation performs better that the unrolled. We suspect that this is due to the fact that the smaller SIMD registers of the A53 CPU do not let the unrolling implementation to reach its full potential. Moreover the Edge machine is not as as powerfull as Desktop and Server in terms of CPU speed. Lastly, it could be due to the compilers not being able to produce optimized vectorized code the arm64 architecture as in x86-64. This is highly unlikely though since we observed the same trends on the implementations on different machines.

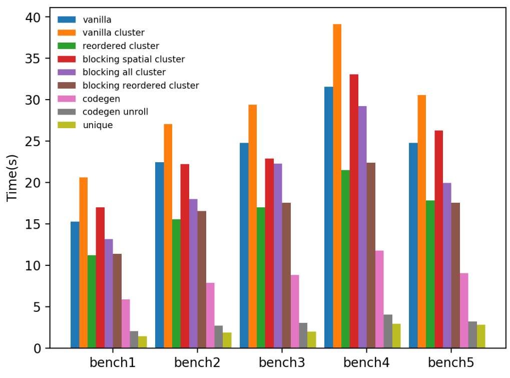
For the unvectorized implementations on the Edge machine (Fig. 8.23) we observe that the unrolling codegen and unique filters are the only implementations that significantly increase performance.



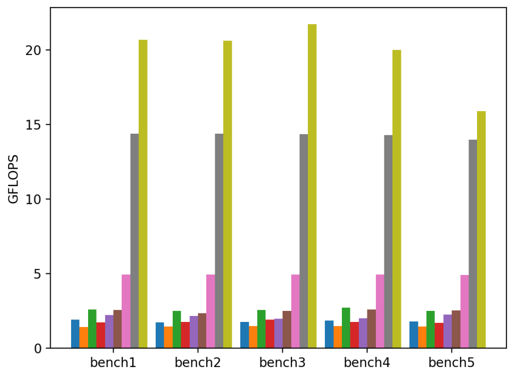
(a) Time on clang



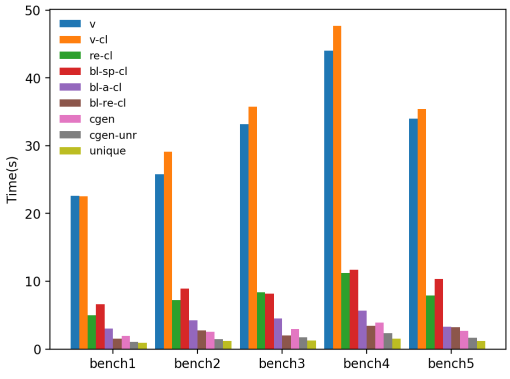
(b) FLOPS on clang



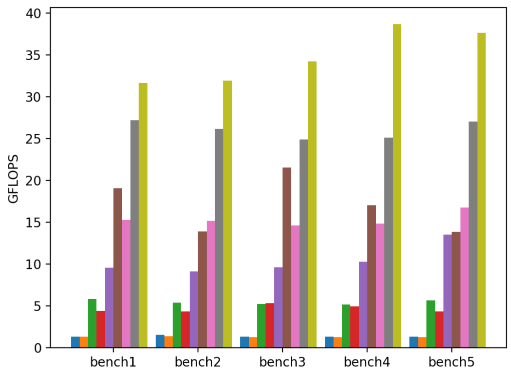
(c) Time on gcc



(d) FLOPS on gcc

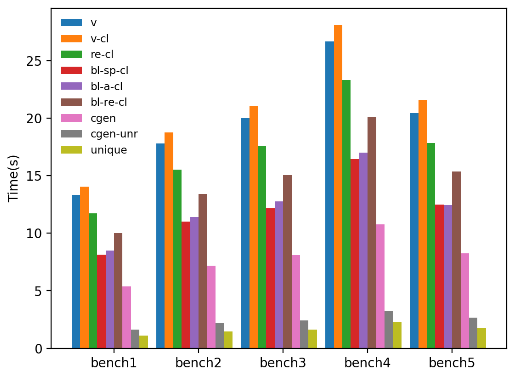


(e) Time on icc

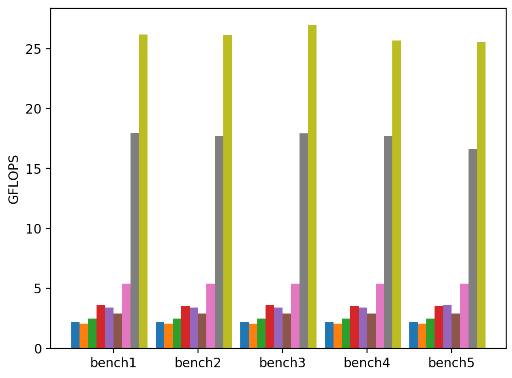


(f) FLOPS on icc

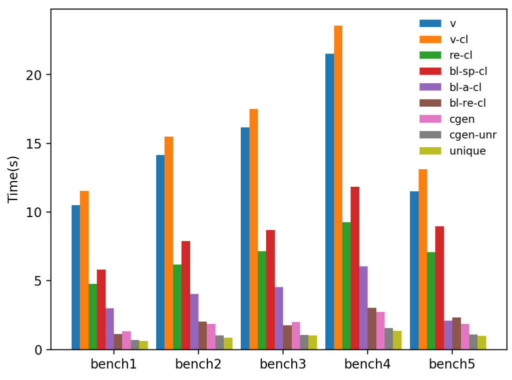
Figure 8.20: Performance with all the compilers on Desktop Table 8.4



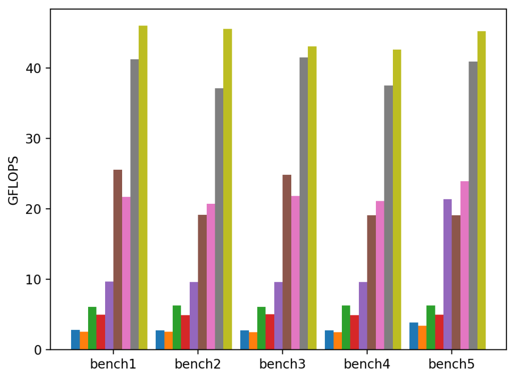
(a) Time on gcc



(b) FLOPS on gcc

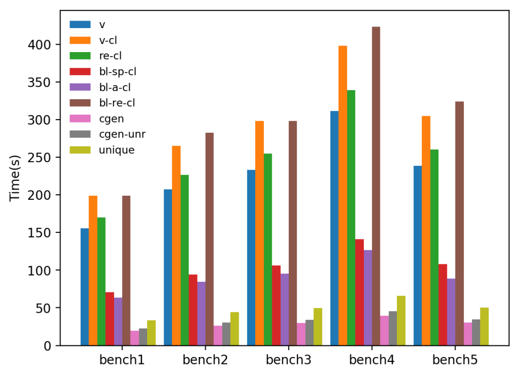


(c) Time on icc

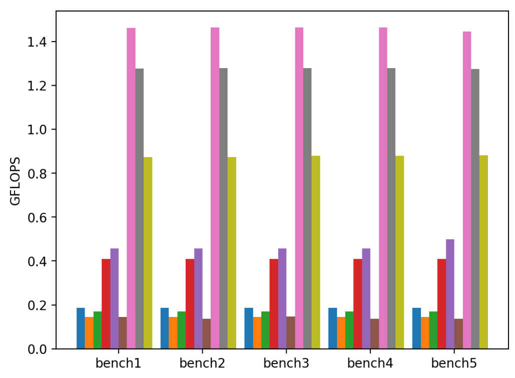


(d) FLOPS on icc

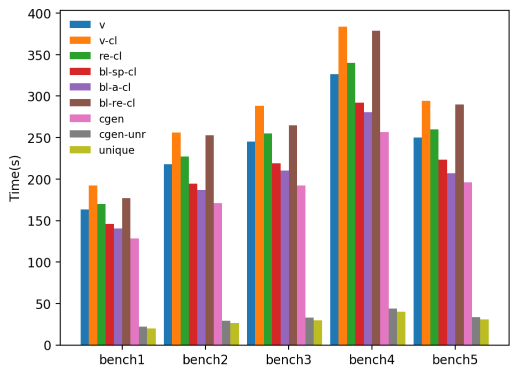
Figure 8.21: Performance with all the compilers on Server Table 8.4



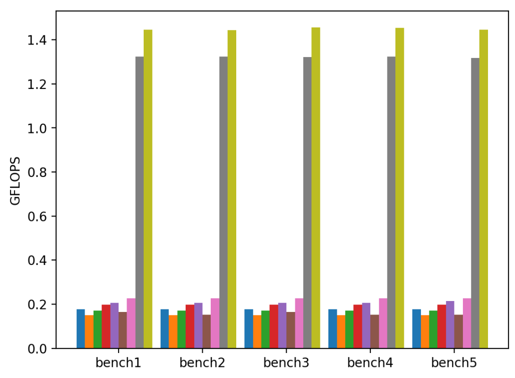
(a) Time on clang



(b) FLOPS on clang

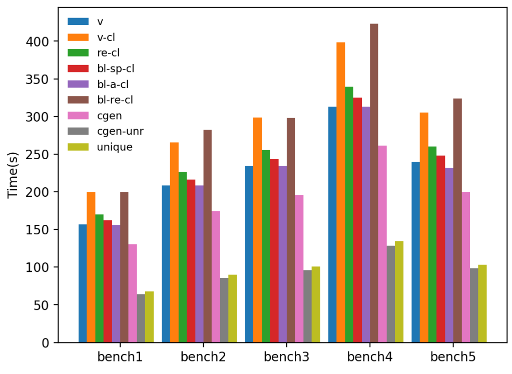


(c) Time on gcc

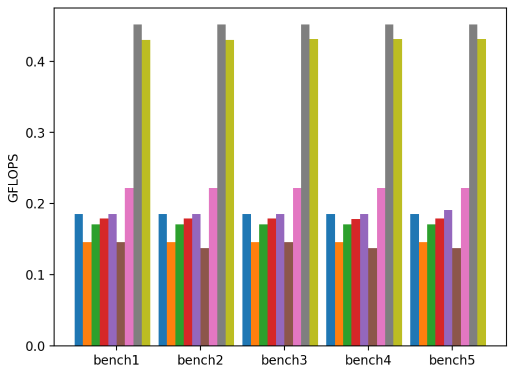


(d) FLOPS on gcc

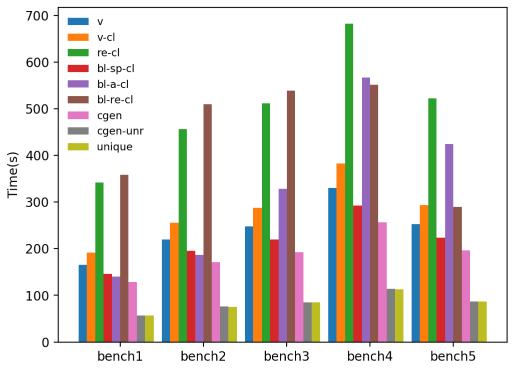
Figure 8.22: Performance with all the compilers on Edge Table 8.4



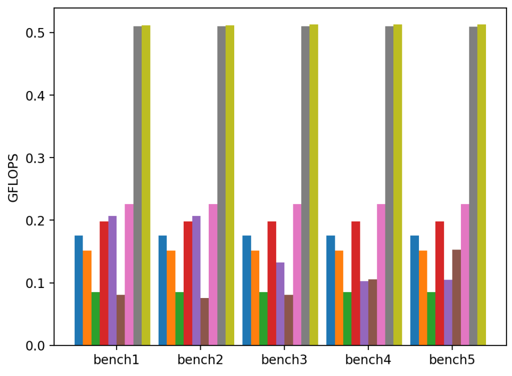
(a) Time on clang



(b) FLOPS on clang



(c) Time on gcc



(d) FLOPS on gcc

Figure 8.23: Performance without vectorization with all the compilers on Edge Table 8.4

Table 8.11: Existence of single and vector FMA instructions on the Desktop machine.

Impl.	Type		Clang	GCC	ICC
Vanilla	Normal	FMA	yes	yes	yes
		vector	no	no	yes
	Cluster	FMA	yes	yes	yes
		vector	no	no	yes
Reorder	Normal	FMA	yes	yes	yes
		vector	yes	yes	yes
	Cluster	FMA	yes	yes	yes
		vector	no	no	yes
Blocking Spatial	Normal	FMA	yes	yes	yes
		vector	yes	no	yes
	Cluster	FMA	yes	yes	yes
		vector	yes	no	yes
Blocking All	Normal	FMA	yes	yes	yes
		vector	yes	no	yes
	Cluster	FMA	yes	yes	yes
		vector	yes	no	yes
Blocking Reordered	Normal	FMA	yes	yes	yes
		vector	yes	yes	yes
	Cluster	FMA	yes	yes	yes
		vector	no	no	yes
Codegen	Cluster	FMA	yes	yes	yes
		vector	yes	no	yes
Codegen Unroll	Cluster	FMA	yes	yes	yes
		vector	yes	yes	yes
Unique	Cluster	FMA	yes	yes	yes
		vector	yes	yes	yes

Table 8.12: Existence of single and vector FMA instructions on the Server machine.

Impl.	Type		GCC	ICC
Vanilla	Normal	FMA	yes	yes
		vector	no	yes
	Cluster	FMA	yes	yes
		vector	no	yes
Reorder	Normal	FMA	yes	yes
		vector	yes	yes
	Cluster	FMA	yes	yes
		vector	no	yes
Blocking Spatial	Normal	FMA	yes	yes
		vector	no	yes
	Cluster	FMA	yes	yes
		vector	no	yes
Blocking All	Normal	FMA	yes	yes
		vector	no	yes
	Cluster	FMA	yes	yes
		vector	no	yes
Blocking Reordered	Normal	FMA	yes	yes
		vector	yes	yes
	Cluster	FMA	yes	yes
		vector	no	yes
Codegen	Cluster	FMA	yes	yes
		vector	no	yes
Codegen Unroll	Cluster	FMA	yes	yes
		vector	yes	yes
Unique	Cluster	FMA	yes	yes
		vector	yes	yes

Table 8.13: Existence of single and vector FMA instructions on the Edge machine.

Impl.	Type		Clang	GCC
Vanilla	Normal	FMA	yes	yes
		vector	no	no
	Cluster	FMA	yes	yes
		vector	no	no
Reorder	Normal	FMA	yes	yes
		vector	yes	yes
	Cluster	FMA	yes	yes
		vector	no	no
Blocking Spatial	Normal	FMA	yes	yes
		vector	yes	no
	Cluster	FMA	yes	yes
		vector	yes	no
Blocking All	Normal	FMA	yes	yes
		vector	yes	no
	Cluster	FMA	yes	yes
		vector	yes	no
Blocking Reordered	Normal	FMA	yes	yes
		vector	yes	yes
	Cluster	FMA	yes	yes
		vector	no	no
Codegen	Cluster	FMA	yes	yes
		vector	yes	no
Codegen Unroll	Cluster	FMA	yes	yes
		vector	yes	yes
Unique	Cluster	FMA	yes	yes
		vector	yes	yes

8.8.9 Parallel implementation

In this section we evaluate how well the two best implementations for cluster convolution scale in parallel and we compare it with traditional GEMM based vanilla convolution. For the Server machine (Fig. 8.25) we can see that the clustering implementations have comparable performance with the GEMM based implementation. We observe as the performance increases as the number of threads increase. For 16 threads we see JIT implementations dropping in performance, while the GEMM convolutions increase slows down. Then for 32 threads the performance increases yet again. This behavior can be attributed to bad cache usage. We can assume that at 16 threads the working set becomes too big, and we either have cache misses because the data do not fit in cache or due to the cache spilling to the second socket. Then at 32 threads where the second socket CPU is utilized the cache is utilized better. Nonetheless there is a lot of room for improvement in the parallel section of the JIT implementations. On Desktop (Fig. 8.24) we can not really make a conclusion regarding the scalability because we have only 2 cores. However we must note that the JIT implementations perform better that the GEMM based ones.

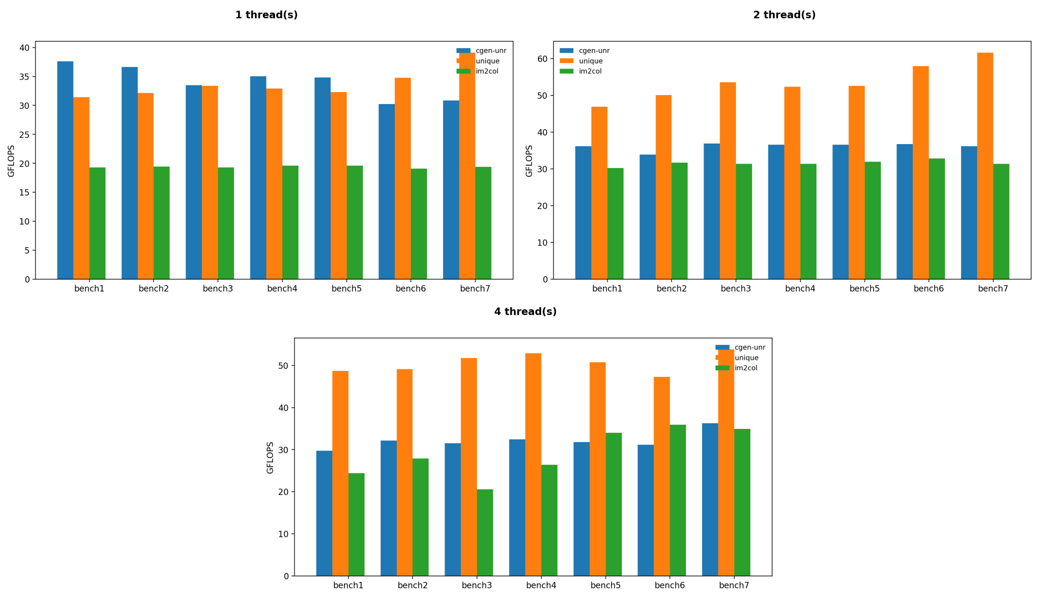


Figure 8.24: Parallel performance while threads increase on Desktop Table 8.4

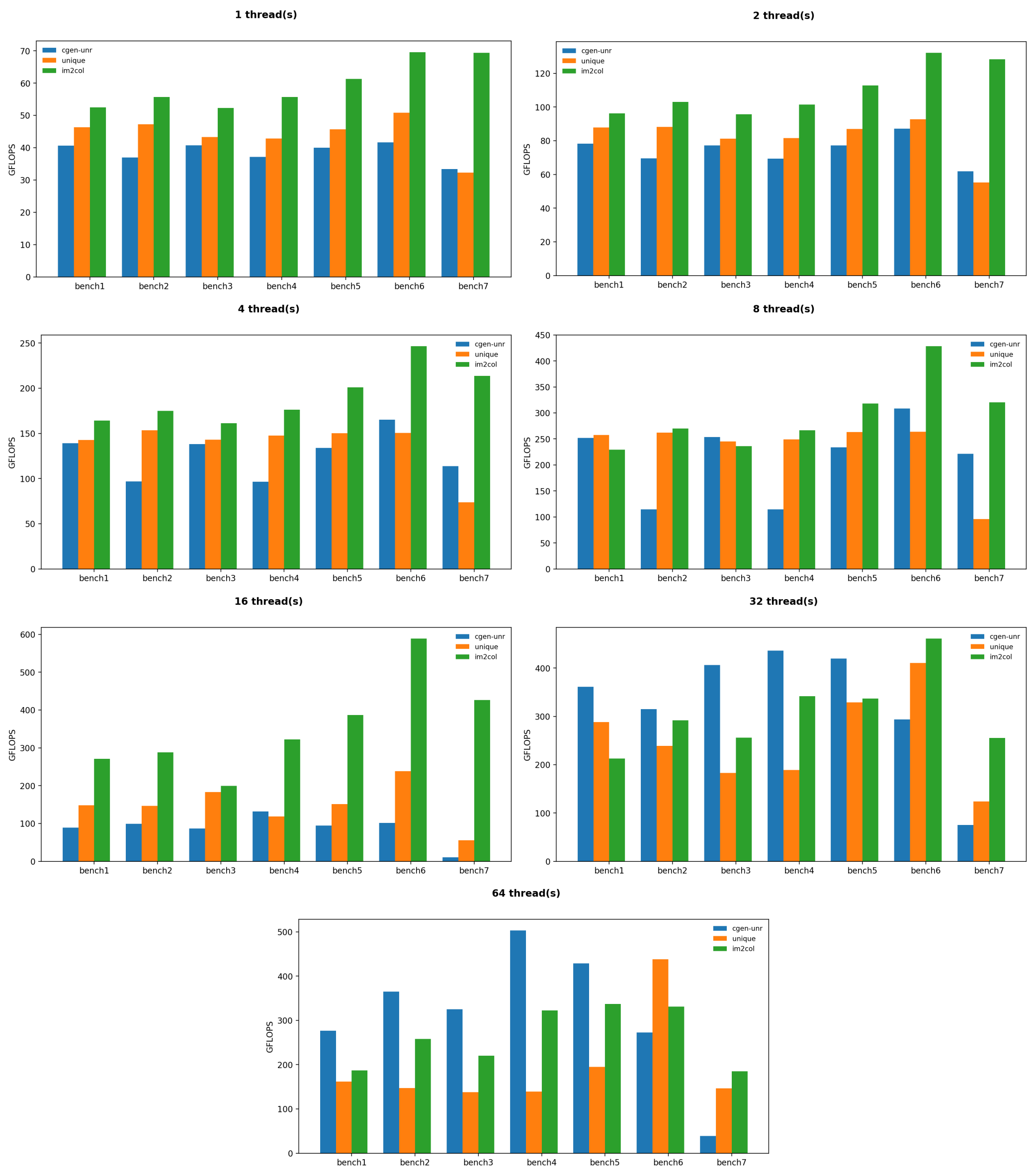


Figure 8.25: Parallel performance while threads increase on Server Table 8.4

Chapter 9

Conclusions and Future Work

In this Thesis we first investigated clustering as a quantization technique Convolutional Layers of DNNs and we observed that it yields good results for DNN model compression. However this method introduces time penalties due to the irregular way data is accessed. Indirect referencing in clustered convolutions is the main source of performance drop compared to the regular counterparts. It makes for poor vectorization potential and data locality.

Therefore we then focused on minimizing the performance penalties that clustering introduced. We experimented, inspired by various techniques present in previous academic works, and confirmed their usefulness. In order to hide the latency from indirect referencing, we utilized loop reordering and blocking techniques. Our results indicate that with restructuring and blocking the loops it is it is possible to achieve a considerable performance boost. These techniques seem to benefit a lot from the existence of SIMD vector instructions and thus are mainly useful on powerful machines when JIT compilation is not an option.

JIT compilation is the technique we investigated next. We tweaked the convolution implementation to utilize two available JIT frameworks and we received promising results. The frameworks lacked some features we needed so we developed our own code generation and execution library to give us the freedom we needed. With this freedom we managed to develop implementations that surpassed the performance of the previous JIT implementations. Using our JIT code generation library we we attempted to remove the indirect references altogether. We demonstrated that it is possible but also that it yields big performance boost for all platforms. The caveat is that it is a viable solution only when the number of unique filters is relatively small.

The JIT implementations had the most positive impact on the cluster convolution algorithm since they exploited data locality and vectorization the most. Even without vectorization, JIT implementations make substantial increases to baseline performance. Moreover they perform comparable to traditional GEMM based convolution implementations. Thus the JIT implementations can be deployed for performance benefits both on powerful machines with SIMD instructions as well as machines with limited resources and even no vectorization capabilities.

In conclusion, in this thesis we managed to compress DNN models by up to 6.4x while retaining their accuracy. With JIT compilation they also retained their original speed but without the memory overheads of GEMM convolutions.

Our work can be extended towards various directions. First the idea behind each convolution optimization can be implemented on the back propagation step of the DNN training process. The back propagation loops are similar to the forward convolution ones and indirect referencing will most likely have the same effects. Moreover it would be interesting to investigate the performance of the optimizations presented in this work on highly parallel environments (e.g. GPUs).

We used the code generation library to compile already known parameters to the convolution and completely unroll the kernel loops. This approach gives a lot of freedom for implementations and could be taken one step further by providing a more fine grained solution, by targeting specific common occurring kernel sizes or by performing more elaborate loop transforming optimizations.

Lastly, works that attempt to enforce fewer kernels to the models as a means of quantization compression already exist and our unique filters approach could be tweaked to work with them. These works address the problem of low memory capabilities of embedded devices and could be directly complemented by our approach since it targets the problem of poor computational performance that is also encountered on said devices.

Bibliography

[1] URL `https://github.com/shicai/MobileNet-Caffe`.

[2] Alireza Aghasi, Afshin Abdi, Nam Nguyen, and Justin Romberg. Net-Trim: Convex Pruning of Deep Neural Networks with Performance Guarantee. *arXiv e-prints*, art. arXiv:1611.05162, November 2016.

[3] Juan Manuel Martinez Caamaño. Easy::jit: A just-in-time compiler for C++. URL `https://github.com/jmmartinez/easy-just-in-time`.

[4] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An Analysis of Deep Neural Network Models for Practical Applications. *arXiv e-prints*, art. arXiv:1605.07678, May 2016.

[5] Matias Capeletto. Boost bimap. 2006. URL `https://www.boost.org/doc/libs/1_73_0/libs/bimap/`.

[6] Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing Neural Networks with the Hashing Trick. *arXiv e-prints*, art. arXiv:1504.04788, April 2015.

[7] François Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. *arXiv e-prints*, art. arXiv:1610.02357, October 2016.

[8] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv e-prints*, art. arXiv:1602.02830, February 2016.

[9] Miguel de Prado, Maurizio Denna, Luca Benini, and Nuria Pazos. QUENN: QUantization Engine for low-power Neural Networks. *arXiv e-prints*, art. arXiv:1811.05896, Nov 2018.

[10] Jia Deng, Wenjun Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. 2009.

[11] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv e-prints*, art. arXiv:1603.07285, March 2016.

[12] Athena Elafrou, Georgios Goumas, and Nektarios Koziris. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. *arXiv e-prints*, art. arXiv:1711.05487, November 2017.

[13] Agner Fog et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 93:110, 2011.

[14] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy Of High-Performance Deep Learning Convolutions On SIMD Architectures. *arXiv e-prints*, art. arXiv:1808.05567, Aug 2018.

[15] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. MorphNet: Fast & Simple Resource-Constrained Structure Learning of Deep Networks. *arXiv e-prints*, art. arXiv:1711.06798, November 2017.

[16] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Understanding the performance of sparse matrix-vector multiplication. pages 283–292, 03 2008. ISBN 978-0-7695-3089-5. doi: 10.1109/PDP.2008.41.

[17] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic Network Surgery for Efficient DNNs. *arXiv e-prints*, art. arXiv:1608.04493, August 2016.

[18] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv e-prints*, art. arXiv:1510.00149, Oct 2015.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv e-prints*, art. arXiv:1512.03385, December 2015.

[20] Yihui He, Xiangyu Zhang, and Jian Sun. Channel Pruning for Accelerating Very Deep Neural Networks. *arXiv e-prints*, art. arXiv:1707.06168, July 2017.

[21] Koen Helwegen, James Widdicombe, Lukas Geiger, Zechun Liu, Kwang-Ting Cheng, and Roeland Nusselder. Latent Weights Do Not Exist: Rethinking Binarized Neural Network Optimization. *arXiv e-prints*, art. arXiv:1906.02107, June 2019.

[22] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18:1527–54, 08 2006. doi: 10.1162/neco.2006.18.7.1527.

[23] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv e-prints*, art. arXiv:1704.04861, Apr 2017.

[24] Yannick Van Huele. URL `https://github.com/yvanhuele/yvanhuele.github.io`.

[25] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, art. arXiv:1502.03167, February 2015.

[26] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv e-prints*, art. arXiv:1408.5093, Jun 2014. URL `https://github.com/BVLC/caffe`.

[27] Petr Kobalicek. AsmJit: Complete x86/x64 JIT and AOT Assembler for C++. URL `https://github.com/asmjit`.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[29] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[30] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed Point Quantization of Deep Convolutional Networks. *arXiv e-prints*, art. arXiv:1511.06393, Nov 2015.

[31] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural Networks with Few Multiplications. *arXiv e-prints*, art. arXiv:1510.03009, October 2015.

[32] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. *arXiv e-prints*, art. arXiv:1411.4038, November 2014.

[33] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press. URL `https://projecteuclid.org/euclid.bsmsp/1200512992`.

[34] Moritz B. Milde, Daniel Neil, Alessandro Aimar, Tobi Delbruck, and Giacomo Indiveri. ADaPTION: Toolbox and Benchmark for Training Convolutional Neural Networks with Reduced Numerical Precision Weights and Activation. *arXiv e-prints*, art. arXiv:1711.04713, Nov 2017.

[35] Asit Mishra, Jeffrey J Cook, Eriko Nurvitadhi, and Debbie Marr. WRPN: Training and Inference using Wide Reduced-Precision Networks. *arXiv e-prints*, art. arXiv:1704.03079, April 2017.

[36] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional Neural Networks using Logarithmic Data Representation. *arXiv e-prints*, art. arXiv:1603.01025, March 2016.

[37] LLVM Project. Llvm passmanagerbuilder. . URL `https://llvm.org/doxygen/classllvm_1_1PassManagerBuilder.html`.

[38] LLVM Project. Llvm vectorizers. . URL `https://llvm.org/docs/Vectorizers.html`.

[39] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *arXiv e-prints*, art. arXiv:1506.01497, June 2015.

[40] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *arXiv e-prints*, art. arXiv:1801.04381, January 2018.

[41] Scipy. scipy.cluster.vq.kmeans. URL `https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.vq.kmeans.html`.

[42] Sungho Shin, Yoonho Boo, and Wonyong Sung. Fixed-point optimization of deep neural networks with adaptive step size retraining. *arXiv e-prints*, art. arXiv:1702.08171, February 2017.

[43] Sanghyun Son, Seungjun Nah, and Kyoung Mu Lee. Clustering convolutional kernels to compress deep neural networks. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, pages 225–240, Cham, 2018. Springer International Publishing.

[44] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for Simplicity: The All Convolutional Net. *arXiv e-prints*, art. arXiv:1412.6806, December 2014.

[45] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. *arXiv e-prints*, art. arXiv:1409.4842, September 2014.

[46] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[47] A. Vasuki and P.T. Vanathi. A review of vector quantization techniques. *Potentials, IEEE*, 25:39 – 47, 08 2006. doi: 10.1109/MP.2006.1664069.

[48] Sahar Voghoei, Navid Hashemi Tonekaboni, Jason G. Wallace, and Hamid R. Arabnia. Deep Learning at the Edge. *arXiv e-prints*, art. arXiv:1910.10231, Oct 2019.

[49] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning Structured Sparsity in Deep Neural Networks. *arXiv e-prints*, art. arXiv:1608.03665, August 2016.

[50] Junru Wu, Yue Wang, Zhenyu Wu, Zhangyang Wang, Ashok Veeraraghavan, and Yingyan Lin. Deep k-Means: Re-Training and Parameter Sharing with Harder Cluster Assignments for Compressing Deep Convolutions. *arXiv e-prints*, art. arXiv:1806.09228, June 2018.

[51] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. *arXiv e-prints*, art. arXiv:1804.03230, April 2018.

[52] Muhamad Yani, S Irawan, and M.T. S.T. Application of transfer learning using convolutional neural network method for early detection of terry's nail. *Journal of Physics: Conference Series*, 1201:012052, 05 2019. doi: 10.1088/1742-6596/1201/1/012052.

[53] Buse Yilmaz, Baris Aktemur, María Garzarán, Sam Kamin, and Furkan Kıraç. Autotuning runtime specialization for sparse matrix-vector multiplication. *ACM Transactions on Architecture and Code Optimization*, 03 2016. doi: 10.1145/2851500.

[54] Zhongzhi Yu, Yemin Shi, Tiejun Huang, and Yizhou Yu. Kernel Quantization for Efficient Network Compression. *arXiv e-prints*, art. arXiv:2003.05148, March 2020.

[55] Geng Yuan, Xiaolong Ma, Caiwen Ding, Sheng Lin, Tianyun Zhang, Zeinab S. Jalali, Yilong Zhao, Li Jiang, Sucheta Soundarajan, and Yanzhi Wang. An Ultra-Efficient Memristor-Based DNN Framework with Structured Weight Pruning and Quantization Using ADMM. *arXiv e-prints*, art. arXiv:1908.11691, August 2019.

[56] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. High Performance Zero-Memory Overhead Direct Convolutions. *arXiv e-prints*, art. arXiv:1809.10170, September 2018.

[57] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. Adaptive Quantization for Deep Neural Network. *arXiv e-prints*, art. arXiv:1712.01048, Dec 2017.

[58] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained Ternary Quantization. *arXiv e-prints*, art. arXiv:1612.01064, December 2016.