Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

Development and Evaluation with AI Tools & Devices: Google Edge TPU for General-Purpose Computing

Διπλωματικη Εργασια

του

Χρόνη Σάκου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων
Αθήνα, Ιούνιος 2022

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

Development and Evaluation with AI Tools & Devices: Google Edge TPU for General-Purpose Computing

Διπλωματικη Εργασια

του

Χρόνη Σάκου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 30η Ιουνίου 2022.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Κωνσταντίνος Σιώζιος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Α.Π.Θ.

Αθήνα, Ιούνιος 2022

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

(Υπογραφή)

.....

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

Περίληψη

Ένας από τους ταχύτερα αναπτυσσόμενους τομείς έρευνας είναι η Τεχνητή Νοημοσύνη (ΤΝ), η οποία έχει φέρει την επανάσταση σε μια πληθώρα εφαρμογών. Τα σύγχρονα νευρωνικά δίκτυα (ΝΔ) απαιτούν μεγάλη υπολογιστική πολυπλοκότητα και, ως αποτέλεσμα, οι σύγχρονες CPU γενικού σκοπού δυσκολεύονται να επιτύχουν επαρκή απόδοση. Για το λόγο αυτό, υπάρχει η επιδίωξη για εφαρμογή περισσότερης ΤΝ σε ενσωματωμένα συστημάτα με μεγαλύτερες υπολογιστικές δυνατότητες. Προβλέποντας αυτή την τάση, η Google ανέπτυξε τις Tensor Processing Units (TPUs) για να επιταχύνει την εκτέλεση εφαρμογών ΤΝ τόσο σε κέντρα δεδομένων όσο και at the edge.

Σε αυτή τη διπλωματική εργασία, θα ασχοληθούμε με το Edge TPU. Το Edge TPU είναι ένα μικρό ολοκληρωμένο κύκλωμα που επιτρέπει την ανάπτυξη εφαρμογών ΤΝ at the edge. Το Edge TPU είναι ικανό να εκτελεί 4 τρισεκατομμύρια πράξεις ανά δευτερόλεπτο, χρησιμοποιώντας 2 Watt ισχύος. Ωστόσο, η αρχιτεκτονική και το σύνολο εντολών τέτοιων επιταχυντών ΤΝ, εμφανίζουν διάφορες προκλήσεις και περιορισμούς. Σε αυτή τη διπλωματική εργασία, στοχεύουμε στην επίλυση των προκλήσεων που συνοδεύουν τους επιταχυντές ΤΝ, σχετικά με εφαρμογές γενικού σκοπού, προτείνοντας τη δική μας μεθοδολογία για την κατασκευή δικτύων συμβατών με το Edge TPU.

Αρχικά, πραγματοποιήσαμε εκτενές benchmarking στο TPU με στόχο να να αξιολογήσουμε τις δυνατότητές του. Τα αποτελέσματα που προέκυψαν αποκαλύπτουν σημαντική επιτάχυνση για το Google Edge TPU σε σύγκριση με τον επεξεργαστή ARM A53 και άλλες ενσωματωμένες συσκευές. Συνολικά, επιτυγχάνει σημαντική επιτάχυνση για μεσαίου και μεγάλου μεγέθους συνελικτικά νευρωνικά δίκτυα (ΣΝΔ) και απλά τεχνητά νευρωνικά δίκτυα (ΤΝΔ) ή προσαρμοσμένα μοντέλα που αποτελούνται κυρίως από πολλαπλασιασμούς πινάκων. Η πράξη του πολλαπλασιασμού πινάκων βελτιώνεται κατά 4 φορές σε σύγκριση με την κβαντισμένη σε 8-bit εκτέλεση στον ARM A53 και 7 φορές σε σχέση με την εκτέλεση σε 32-bit κινητής υποδιαστολής. Επίσης, για κλασικές εφαρμογές ψηφιακής επεξεργασίας σήματος (ΨΕΣ), το Edge TPU παρέχει έως και 6 φορές καλύτερη απόδοση από τον ARM A53.

Λέξεις Κλειδιά

Τεχνητή νοημοσύνη, ενσωματωμένα συστήματα, Edge TPU, συνελικτικά νευρωνικά δίκτυα,

Abstract

One of the fastest growing ground-based areas of research is Artificial Intelligence (AI), which has revolutionized a variety of application domains. Modern artificial neural networks (ANNs) impose increased computational complexity, and as a result, general-purpose CPUs struggle to provide sufficient performance. For this reason, developers are forced to integrate AI into more power efficient, AI microchips and accelerators. Anticipating this trend, Google provides the Tensor Processing Units (TPUs) to accelerate AI inference in data-centers and at the edge.

In this thesis, targeting embedded AI, we focus on the Edge TPU. The Edge TPU is a small Application-Specific Integrated Circuit (ASIC) that delivers high performance, enabling the deployment of high accuracy AI at the edge. It is a dedicated hardware that enables the parallelization of certain computations in order to achieve faster inference of them. The Edge TPU processor is capable of performing 4 Trillion Operations Per Second (TOPS), using 0.5 Watt for each TOPS. However, the architecture of such an AI-specific accelerator imposes hardware challenges and limitations for non-AI workloads for general-purpose computing. In this thesis, our goal is to provide solutions to this challenge by proposing a custom methodology for building Edge TPU compatible networks for general-purpose calculations. Moreover, we propose a solution for overcoming the barrier of the 8-bit-only operations on the TPU in a way that, we support both element-wise and matrix multiplications for larger bit-widths without significant decrease in performance.

Initially, we perform benchmarking on the TPU to explore and evaluate its capabilities, including both pre-trained and custom networks. Overall, the Edge TPU provides remarkable speedup for medium- and large-sized CNNs and MLPs, as well as for custom models dominated by matrix multiplications. The matrix multiplication operations are improved up to 4x compared to the 8-bit quantized ARM execution and up to 7x for 32-bit floating point. Moreover, for classic Digital Signal Processing (DSP) operations, such as the Sobel Edge Detector and Image Binning, the Edge TPU provides up to 6x better performance than ARM A53.

Keywords

Artificial Intelligence, Embedded Systems, Edge TPU, Convolutional Neural Networks, AI Accelerators, GEMM

Ευχαριστίες

Αρχικά, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντα καθηγητή Δημήτριο Σούντρη, ο οποίος με εμπιστεύτηκε και μου έδωσε την ευκαιρία να εκπονήσω τη διπλωματική μου εργασία στο Εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων (MicroLab) του ΕΜΠ. Επίσης, θα ήθελα να ευχαριστήσω τους δύο στενούς συνεργάτες μου, τον μεταδιδακτορικό ερευνητή Γεώργιο Λεντάρη και τον υποψήφιο διδάκτορα Βασίλειο Λέων για τη πολύτιμη βοήθεια και τη καθοδήγησή τους καθ΄ όλη τη διάρκεια της διπλωματικής μου εργασίας. Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου για την υποστηρίξη τους καθ΄ όλη την ακαδημαϊκή μου πορεία.

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

Τεχνητή Νοημοσύνη

Η τεχνητή νοημοσύνη (ΤΝ) είναι ένας ευρύς κλάδος της επιστήμης των υπολογιστών που ασχολείται με την κατασκευή έξυπνων μηχανών ικανών να εκτελούν εργασίες που συνήθως απαιτούν ανθρώπινη νοημοσύνη. Η κύρια εφαρμογή της τεχνητής νοημοσύνης είναι η μηχανική μάθηση (ΜΜ). Ο στόχος της μηχανικής μάθησης είναι να επιτρέψει στις μηχανές να μαθαίνουν από πρότερες εμπειρίες τους μέσα από μεγάλο αριθμό δεδομένων. Η μηχανική μάθηση συνθέτει και ερμηνεύει πληροφορίες για την ανθρώπινη κατανόηση, σύμφωνα με προκαθορισμένες παραμέτρους, συμβάλλοντας στην εξοικονόμηση χρόνου, στη μείωση των σφαλμάτων, στη δημιουργία προληπτικών ενεργειών και στην αυτοματοποίηση διαδικασιών σε πληθώρα εφαρμογών της καθημερινότητας. Οι κινητήριες δυνάμεις για την επίτευξη του στόχου της αυτοματοποίησης είναι τα Νευρωνικά Δίκτυα (ΝΔ). Τα νευρωνικά δίκτυα αντικατοπτρίζουν τη συμπεριφορά του ανθρώπινου εγκεφάλου, επιτρέποντας στους υπολογιστές να αναγνωρίζουν μοτίβα και να επιλύουν κοινά προβλήματα στους τομείς της τεχνητής νοημοσύνης, της μηχανικής και της βαθιάς μάθησης.

Τα νευρωνικά δίκτυα στοχεύουν σε λειτουργικότητα που μοιάζει με εκείνη του ανθρώπινου εγκεφάλου και βασίζονται σε έναν απλό τεχνητό νευρώνα: μια μη γραμμική συνάρτηση (όπως max(0, value)) ενός σταθμισμένου αθροίσματος εισόδων. Αυτοί οι τεχνητοί νευρώνες οργανώνονται σε στρώματα, με τις εξόδους ενός στρώματος να γίνονται οι είσοδοι του επόμενου στην ακολουθία. Οι δύο φάσεις των νευρωνικών δικτύων ονομάζονται εκπαίδευση και πρόβλεψη και αναφέρονται στην ανάπτυξη έναντι της παραγωγής. Ο προγραμματιστής επιλέγει τον αριθμό των στρωμάτων και τον τύπο του ΝΔ και η εκπαίδευση καθορίζει τα βάρη. Τρία είδη νευρωνικών δικτύων είναι δημοφιλή στις μέρες μας:

1. Πολυεπίπεδοι αισθητήρες (ΠΑι): κάθε νέο στρώμα/επίπεδο είναι ένα σύνολο μη γραμμικών λειτουργιών σταθμισμένου αθροίσματος όλων των εξόδων (πλήρως συνδεδεμένων) από ένα προηγούμενο στρώμα, το οποίο επαναχρησιμοποιεί τα βάρη.

2. Συνελικτικά Νευρωνικά Δίκτυα (ΣΝΔ): Κάθε επόμενο στρώμα είναι ένα σύνολο μη γραμμικών συναρτήσεων σταθμισμένων αθροισμάτων χωρικά κοντινών υποσυνόλων εξόδων από το προηγούμενο στρώμα, το οποίο επίσης επαναχρησιμοποιεί τα βάρη.

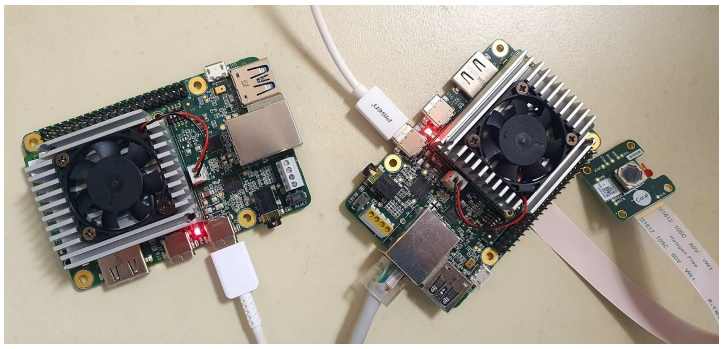
3. Περιοδικό Νευρωνικό Δίκτυο (ΠΝΔ): Κάθε επόμενο στρώμα είναι μια συλλογή μη γραμμικών λειτουργιών σταθμισμένων αθροισμάτων εξόδων και της προηγούμενης κατάστα-

σης. Το πιο δημοφιλές ΠΝΔ είναι η μακροπρόθεσμη-βραχυπρόθεσμη μνήμη (LSTM). Η ίδεα πίσω από το LSTM έχει να κάνει με την απόφαση του τι να ξεχάσει και τι να περάσει ως κατάσταση στο επόμενο στρώμα. Τα βάρη επαναχρησιμοποιούνται σε χρονικά βήματα.

Τα τελευταία χρόνια, η τεχνητή νοημοσύνη έχει φέρει την επανάσταση σε πληθώρα εφαρμογών της καθημερινότητας, από την ιατροφαρμακευτική περίθαλψη, την αυτόνομη οδήγηση ως και τις διαστημικές εφαρμογές. Τα νευρωνικά δίκτυα παρουσιάζουν υψηλή υπολογιστική πολυπλοκότητα, κάτι που δυσκολεύει τους επεξεργαστές γενικού σκοπού να ανταποκριθούν στις απαιτήσεις. Επίσης, καθώς οι σύγχρονες εφαρμογές απαιτούν λήψη αποφάσεων σε πραγματικό χρόνο, η επεξεργασία και ανάλυση δεδομένων μετακινείται at the edge, δηλαδή όλο και πιο κοντά στους αισθητήρες συλλογής των δεδομένων. Ως εκ τούτου, δημιουργείται η ανάγκη για ανάπτυξη πιο εξειδικευμένων και καινοτόμων επεξεργαστών και επιταχυντών τεχνητής νοημοσύνης όπως είναι οι GPUs, οι VPUs και τα FPGAs. Προβλέποντας αυτή την τάση, η Google ανέπτυξε τις Tensor Processing Units (TPUs) για να επιταχύνει την εκτέλεση εφαρμογών μηχανικής μάθησης. Η TPU είναι ένας επιταχυντής που προσφέρει υψηλή απόδοση με μικρό φυσικό και ενεργειακό αποτύπωμα, επιτρέποντας την ανάπτυξη εφαρμογών ΤΝ τόσο σε κέντρα δεδομένων όσο και at the edge.

Google Edge TPU

Το Edge TPU είναι ένα μικρό ολοκληρωμένο κύκλωμα σχεδιασμένο από τη Google για να παρέχει μηχανική μάθηση υψηλής απόδοσης σε συσκευές χαμηλής κατανάλωσης ισχύος. Λειτουργεί συμπληρωματικά με το Cloud TPU που παρέχει τις υποδομές για cloud-to-edge εφαρμογές τεχνητής νοημοσύνης. Το TPU προσφέρεται μέσω της Coral, η οποία είναι μια πλατφόρμα που παρέχει προϊόντα για τεχνητή νοημοσύνη at the edge. Η Coral παρέχει TPUs είτε σε μορφή πρωτοτύπων, τύπου development board είτε έτοιμα προϊόντα παραγωγής. Όσον αφορά τα πρωτότυπα, υπάρχουν δύο development boards, το Dev Board και το Dev Board Mini που είναι αυτόνομοι υπολογιστές και περιλαμβάνουν το System on Module. Ο επιταχυντής TPU είναι ενσωματωμένος στο SoM μαζί με κεντρικές μονάδες επεξεργασίας τύπου ARM, διεπαφές εισόδων-εξόδων και διαφόρων μεγέθων block μνήμης. Προσφέρεται επίσης και ο USB accelerator που περιλαμβάνει το TPU και μπορεί να συνδεθεί σε οποιοδήποτε host pc μέσω USB3. Αναφορικά με τα προϊόντα παραγωγής, υπάρχει μεγάλη ποικιλία από PCIe devices τα οποία μπορούν να συνδεθούν σε slots οποιασδήποτε μητρικής πλακέτας και δύνανται να ενσωματώνουν περισσότερες από μία TPU. Στην εικόνα που ακολουθεί (Σχήμα 1), βλέπουμε τα development boards που υπάρχουν διαθέσιμα στο εργαστήριο και τα οποία χρησιμοποιήσαμε στην παρούσα διπλωματική.



(α′) Dev Boards

(β′) Dev Board Mini

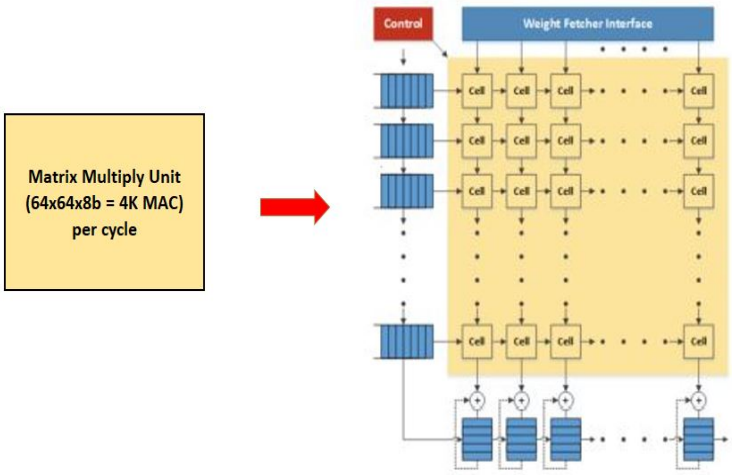
Σχήμα 1: Πρωτότυπα προϊόντα της Coral στο Microlab

Edge TPU Architecture

Η αρχιτεκτονική του Edge TPU επιταχύνει την εκτέλεση νευρωνικών δικτύων για σύγχρονες εφαρμογές TN, βασιζόμενη σε μια συστολική συστοιχία που εκτελεί λειτουργίες σε μορφή τανυστών, σε αντίθεση με τις περισσότερες εφαρμογές TN που λαμβάνουν ως εισόδους τανυστές και ενημερώνουν επαναληπτικά τις παραμέτρους και τα βάρη από προηγούμενα αποτελέσματα. Θέτοντας τανυστές ως προεπιλεγμένες εισόδους και εξόδους, καθιστά την αρχιτεκτονική του TPU θεμελιωδώς διαφορετική από τις συμβατικές αρχιτεκτονικές των CPU και GPU που κάνουν υπολογισμούς σε διανυσματικά ζεύγη δεδομένων.

Το ASIC του Edge TPU βασίζεται σε μια συστολική συστοιχία (systolic array) πολλαπλασιαστών και συσσωρευτών, οι διαστάσεις των οποίων δεν έχουν αποκαλυφθεί ακόμα, αλλά σύμφωνα με την Q-Engineering [5], εκτιμάται ως μια συστοιχία μεγέθους 64x64 με συχνότητα ρολογιού τουλάχιστον f_{clk}=480MHz. Μία μεμονωμένη TPU είναι ικανή να εκτελέσει 4 τρισεκατομμύρια πράξεις ανά δευτερόλεπτο (TOPS), χρησιμοποιώντας 0,5 Watt ισχύος για κάθε TOPS (2 TOPS ανά Watt). Τρέχοντας στα 480 MHz μπορεί θεωρητικά να εκτελέσει 64 x 64 x 480.000.000 = 2 τρισεκατομμύρια πολλαπλασιασμούς και προσθέσεις ανά δευτερόλεπτο. Ή αν κοιτάξουμε σε μεμονωμένες πράξεις, είναι 4 τρισεκατομμύρια (4 TOPS). Η κατανάλωση ισχύος του chip φτανει τα 1,5 Watt, ενώ συνολικα όλο το SoM καταναλώνει περί τα 4,5 Watt ισχύος.

Όσον αφορά τη μνήμη, το Edge TPU ενσωματώνει μεγάλη μνήμη εντός του chip για να διατηρεί τα ενδιάμεσα αποτελέσματα που επαναχρησιμοποιούνται αργότερα. Αυτό επιτρέπει μεγαλύτερη ταχύτητα εκτέλεσης σε σύγκριση με τη φόρτωση των δεδομένων των παραμέτρων από εξωτερική μνήμη. Με άλλα λόγια, η SRAM του TPU χρησιμοποιείται ως "cache", παρόλο που είναι μια scratchpad μνήμη που έχει εκχωρηθεί από τον μεταγλωτιστή. Σε αντίθεση με τους συμβατικούς επεξεργαστές, το Edge TPU χρησιμοποιεί ένα σετ εντολών τύπου CISC και οποιαδήποτε λειτουργία συντελείται μέσω του κεντρικού υπολογιστή. Επίσης, οι μονάδες TPU υποστηρίζουν μόνο λειτουργίες σε περιορισμένο επίπεδο ακρίβειας, όσο επαρκεί για να ικανοποιήσει τις απαιτήσεις των σύγχρονων εφαρμογών TN, μειώνοντας σημαντικά τόσο το κόστος τους όσο και τις ενεργειακές απαιτήσεις.



Σχήμα 2: TPU: μονάδες MAC σε συστολική συστοιχία

Εργαλεία & Frameworks

Η ανάπτυξη νευρωνικών δικτύων στη TPU βασίζεται σε ένα πολύ διαδεδομένο framework, το TensorFlow. Το TensorFlow είναι μια ελεύθερη πλατφόρμα ανοιχτού κώδικα, η οποία χρησιμοποιείται κυρίως για την ανάπτυξη, την εκπαίδευση και την υλοποίηση μοντέλων μηχανικής μάθησης. Είναι ικανό να λειτουργεί σε υψηλή κλίμακα και σε ετερογενή περιβάλλοντα. Το TensorFlow επιτρέπει στους προγραμματιστές να δημιουργούν δομές ροής δεδομένων που περιγράφουν τον τρόπο με τον οποίο τα δεδομένα μετακινούνται μέσω ενός γράφου ή μιας σειράς κόμβων. Κάθε κόμβος στον γράφο αντιπροσωπεύει μια μαθηματική λειτουργία και κάθε σύνδεση ή ακμή μεταξύ κόμβων είναι ένας πολυδιάστατος πίνακας δεδομένων ή ένας τανυστής.

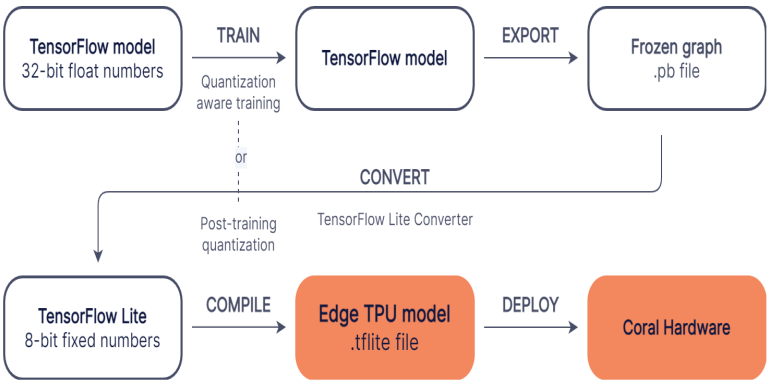
Η διαδικασία που ακολουθείται για την εφαρμογή ενός μοντέλου μηχανικής μάθησης στο TPU είναι απλή και περιλαμβάνει τα ακόλουθα βήματα. Αρχικά, αναπτύσσεται ο κώδικας σε TensorFlow, δηλαδη χτίζεται και εκπαιδεύεται το νευρωνικό δίκτυο. Στη συνέχεια, το μοντέλο μετατρέπεται σε format Tensorflow Lite, μεταγλωτίζεται από τον Edge TPU Compiler και τέλος εκτελείται στη TPU με χρήση του TF Lite API. Το TensorFlow Lite είναι ένα framework που επιτρέπει την ταχύτερη ανάπτυξη και εκτέλεση εφαρμογών μηχανικής μάθησης σε ενσωματωμένες συσκευές. Ένα μοντέλο TensorFlow Lite αντιπροσωπεύεται σε μια ειδική φορητή μορφή γνωστή ως FlatBuffers. Αυτό παρέχει πολλά πλεονεκτήματα σε σχέση με τη μορφή buffer πρωτοκόλλου του TensorFlow, όπως μειωμένο μέγεθος (μικρό αποτύπωμα κώδικα) και ταχύτερη εξαγωγή συμπερασμάτων (τα δεδομένα έχουν άμεση πρόσβαση χωρίς επιπλέον βήμα ανάλυσης) που επιτρέπει στο TensorFlow Lite να εκτελείται αποτελεσματικά σε συσκευές με περιορισμένους υπολογιστικούς πόρους και περιορισμένη μνήμη.

Συμβατότητα με το Edge TPU

Προκειμένου να μπορεί να εκτελεστεί και να επιταχυνθεί ένα μοντέλο από το TPU πρέπει να ικανοποιούνται κάποιες προϋποθέσεις. Αρχικά, είναι σημαντικό να τονίσουμε ότι το TPU υποστηρίζει πράξεις μόνο για 8-bit ακέραιους αριθμούς, κάτι που σημαίνει πως το μοντέλο μας πρέπει να κβαντιστεί κατά την μετατροπή σε .tflite μέσω του TF Lite Converter. Η κβάντιση είναι μια μορφή βελτιστοποίησης, για τη μείωση του μεγέθους του μοντέλου και του χρόνου εκτέλεσης με ελάχιστη απώλεια ακρίβειας. Με τη χρήση του converter τα μοντέλα αποθηκεύονται σε μορφή SavedModel και δημιουργούνται είτε χρησιμοποιώντας API υψηλού επιπέδου τύπου Keras είτε χαμηλού επιπέδου (από τα οποία δημιουργούνται concrete functions). Ωστόσο, για να πραγματοποιηθεί η κβάντιση πρέπει να ορίσουμε ενα representative dataset, δηλαδή ένα σύνολο δεδομένων ενδεικτικών των αναμενόμενων εισόδων και εξόδων του εκάστοτε μοντέλου. Αυτό γίνεται ώστε κατά την κβάντιση και αποκβάντιση οποιουδήποτε μοντέλου, το TPU να λάβει την σωστή είσοδο αλλά και να μας επιστρέψει τη σωστή έξοδο.

Στο δεύτερο στάδιο επεξεργασίας ενός μοντέλου, ώστε να είναι συμβατό με το TPU, επιστρατεύεται ο Edge TPU Compiler. Ο Compiler αντιστοιχίζει τα operations ενός δικτύου σε εκείνα που υποστηρίζει το TPU, αν αυτό είναι εφικτό, καθώς υποστηρίζονται μόνο ορισμένα operations για ΤΝ, όπως είναι η συνέλιξη, το max pooling, ο πολλαπλασιασμός πινάκων, κ.α. Οι μη υποστηριζόμενες λειτουργίες εκτελούνται από τον ARM επεξεργαστή, κάτι που συνεπάγεται επιπρόσθετο χρόνο στην εκτέλεση ενός δικτύου. Επίσης, ο compiler είναι αυτός που προ-αποφασίζει πόση από την διαθέσιμη SRAM του TPU θα αναθέσει σε κάθε μοντέλο που μεταγλωττίζεται για αυτό, ενώ στην περίπτωση που η χωρητικότητα της δεν μπορεί να εξυπηρετήσει ένα δίκτυο, τότε γίνεται χρήση και της εξωτερικής μνήμης, κάτι που ωστόσο σημαίνει πρόσθετη καθυστέρηση.

Το Σχήμα 3 απεικονίζει τη βασική διαδικασία για τη δημιουργία ενός μοντέλου που είναι συμβατό με το Edge TPU. Το μεγαλύτερο μέρος της ροής εργασίας χρησιμοποιεί τυπικά εργαλεία TensorFlow. Μόλις έχουμε ένα μοντέλο TensorFlow Lite, τότε χρησιμοποιούμε τον Edge TPU Compiler για να δημιουργήσουμε ένα αρχείο .tflite που είναι συμβατό με το Edge TPU.



Σχήμα 3: Διαδικασία δημιουργίας μοντέλου συμβατό με το Edge TPU

Αρχικό benchmarking

Πραγματοποιήσαμε μία εκτενή συγκριτική αξιολόγηση που περιλαμβάνει διάφορους τύπους νευρωνικών δικτύων και εφαρμογών, καθώς και δίκτυα που ποικίλουν σε πολυπλοκότητα. Η αξιολόγησή μας περιλαμβάνει αποτελέσματα που συλλέχθηκαν είτε από τη βιβλιογραφία είτε από δικές μας εφαρμογές. Όσον αφορά τα ενσωματωμένα συστήματα που αξιολογήσαμε, είναι τα εξής:

TPU: 2 Coral TPU dev boards (regular and mini)
GPU: NVIDIA's Jetson Nano (Cortex-A57 + 128-core Maxwell GPU) [6]
VPU: Intel's NCS2 (host machine with i7 CPU + Myriad X VPU) [7]
FPGA: Xilinx's Zynq FPGAs (Zynq-7020, ZCU104) [8]
CPU: ARM CPUs (ενσωματωμένες στα προαναφερθέντα boards, με συχνότητα ρολογιού 1.3-1.5 GHz)

Μετρικές

Η **καθυστέρηση** είναι ο χρόνος που απαιτείται για την εκτέλεση μίας επανάληψης. Μετράται σε μονάδες χρόνου χρησιμοποιώντας τη συνάρτηση time() της python. Ο μετρητής υπολογίζει το χρόνο που περνάει από τη στιγμή που καλείτε ο interpreter έως ότου να επιστρέψει την έξοδο της εκτέλεσης. Κατά τη διάρκεια της κλήσης, στο παρασκήνιο εκτελούνται όλοι οι απαραίτητοι υπολογισμοί, ενώ καμία άλλη λειτουργία δεν μπορεί να διετελεστεί αν δεν ολοκληρωθεί η κλήση του interpreter. Η καθυστέρηση μετράται σε χιλιοστά του δευτερολέπτου (ms).

Το **throughput** είναι ο ρυθμός με τον οποίο ένα σύστημα μπορεί να επεξεργαστεί τις εισόδους που λαμβάνει. Ορίζεται ως το πλήθος μετρήσεων/υπολογισμών ανά δεδομένο χρόνο. Δεν είναι ένα μέτρο για το πόσο συχνά γίνεται μια διεργασία, αλλά είναι μια μέτρηση του όγκου τους. Στην περίπτωσή μας, θεωρούμε ότι το throughput είναι αντιστρόφως ανάλογο του χρόνου εκτέλεσης (καθυστέρηση) μιας επανάληψης. Αυτό οφείλεται στο γεγονός ότι ο interpreter καλείται για μία είσοδο κάθε φορά. Το throughput μετράται σε καρέ ανά δευτερόλεπτο (FPS).

$$throughput = \frac{1}{latency} \tag{1}$$

Πίνακας 1: Αποτελέσματα benchmarking [1–4]

Neural Network	**ARM A-53**[1] **(ms)**	**TPU DevB**[2] **(ms)**	**TPU DevB**[2] **(FPS)**	**Nvidia Jetson Nano**[3] **(FPS)**	**Intel NCS2**[4] **+ i7 (FPS)**	**Zynq ZCU104**[5] **(FPS)**
Inception V1	392	4.1	244	76	93	202
Inception V4	3157	102	10	11	10	30.5
MobileNet V1	164	2.4	417	80	119	344
MobileNet V2	122	2.6	385	60	75	284
SSD MobileNet V2	282	14	71	39	58	86
ResNet-50 V1	1763	56	18	21	29	93
SqueezeNet	232	2	500	104	287	305
Vgg16	4595	343	3	12		21.5
Vgg19	5538	357	3	10		18.5

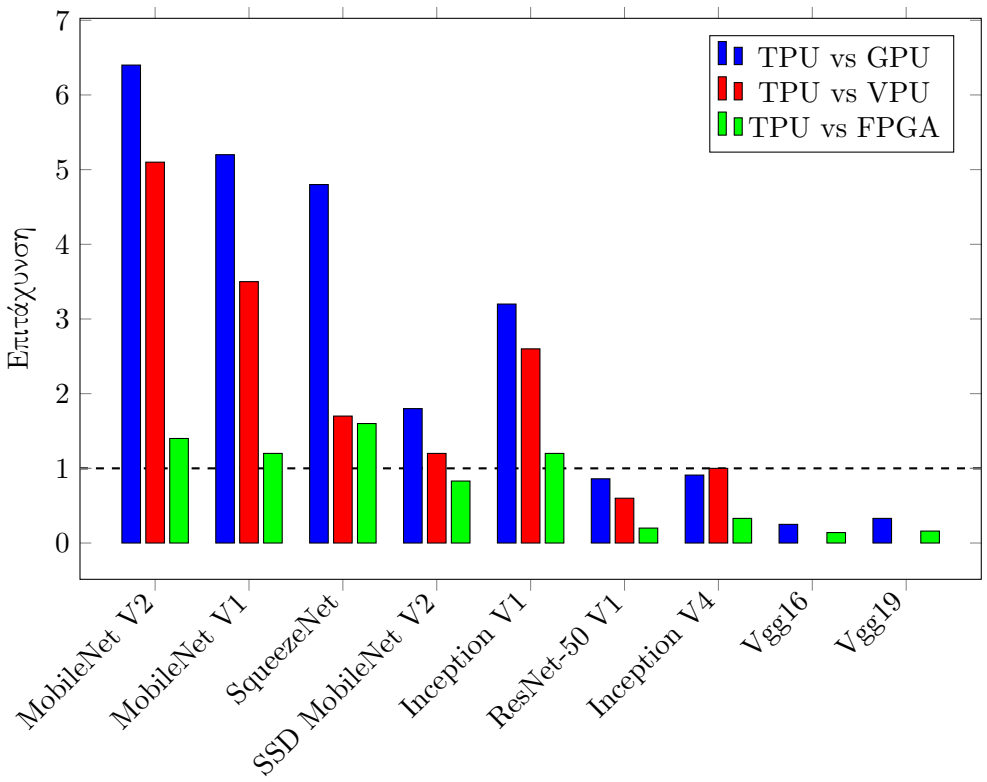
[1] Embedded CPU: Quad-core Cortex-A53 @ 1.5GHz [1]

[2] Dev Board: Quad-core Cortex-A53 @ 1.5GHz + Edge TPU @ 500 MHz @ 2 Watts [1]

[3] Nvidia GPU: Quad-core ARM Cortex-A57 @ 1.5GHz @ 10 Watts, 4 GB LPDDR4 [2]

[4] Intel VPU: Intel Neural Compute Stick 2 + Movidius Myriad X VPU @ 700MHz [3]

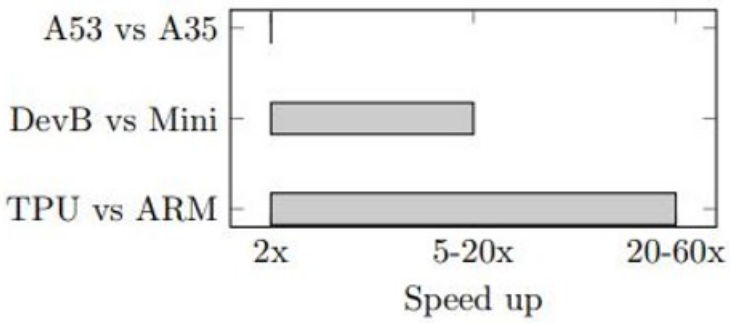
[5] FPGA: Quad-core Cortex-A53 @ 1.5GHz + Dual-core Cortex-R5 + 2 GB DDR4 + 504k LUTs + 1728 DSP [4]



Σχήμα 4: Επιτάχυνση στο TPU σε σχέση με άλλους επεξεργαστές

Το Σχήμα 4 απεικονίζει τα αποτελέσματα από την αρχική αξιολόγηση που πραγματοποιήσαμε σε νευρωνικά δίκτυα και εφαρμογές της βιβλιογραφίας, τα οποία επιβεβαιώσαμε αμφότεροι στα δικά μας πειράματα. Ουσιαστικά, το σχήμα απεικονίζει την επιτάχυνση που παρουσίασε το Edge TPU σε σύγκριση με τους επεξεργαστές των υπόλοιπων ενσωματωμένων συστημάτων που αξιολογήσαμε. Τα μοντέλα που παρουσιάζονται στο σχήμα έχουν ταξινομηθεί ανά μέγεθος, προκειμένου να αξιολογήσουμε την επίδρασή του μεγέθους στην απόδοση των εκάστοτε επεξεργαστών.

Αυτό που παρατηρούμε είναι, ότι για ΣΝΔ μικρού και μεσαίου μεγέθους, ο επιταχυντής Edge TPU επιτυγχάνει 3-7 φορές μικρότερο χρόνο εκτέλεσης από τη Nvidia GPU με την μισή, τουλάχιστον, κατανάλωση ενέργειας. Σε σύγκριση με την Intel Myriad X VPU, οι συντελεστές επιτάχυνσης κυμαίνονται μεταξύ 2-5 για τα ίδια ΣΝΔ μικρού και μεσαίου μεγέθους, ενώ για τα ίδια δίκτυα το Edge TPU αποδίδει εξίσου καλά με το Zynq ZCU104 FPGA. Για δίκτυα μεγαλύτερου μεγέθους και υψηλότερης πολυπλοκότητας, όπως το ResNet, το Inception V4 και το Vgg, το Edge TPU είτε δεν αποδίδει τόσο καλά, είτε επιτυγχάνει παρόμοια αποτελέσματα με τις άλλες ενσωματωμένες μονάδες επεξεργασίας.



Σχήμα 5: Σύγκριση του Dev Board με το Dev Board Mini

Κατά τη διάρκεια της παρούσας διπλωματικής, είχαμε επίσης την ευκαιρία να συγκρίνουμε το Dev Board της Coral με το Dev Board Mini, τόσο ως προς τους επιταχυντές TPU όσο και ως προς τους επεξεργαστές ARM. Το Σχήμα 5 παρουσιάζει τα συγκριτικά αποτελεσμάτα για τους επεξεργαστές TPU και ARM των δύο board. Παρόλο που και οι δύο συσκευές φιλοξενούν το ίδιο τσιπ Edge TPU, το Dev Board Mini φαίνεται να είναι 5-20 φορές πιο αργό από το μεγάλο board. Αυτό υποδηλώνει ότι η απόδοση επηρεάζεται σημαντικά τόσο από τον επεξεργαστή ARM, που συνοδεύει το εκάστοτε TPU, όσο και την αρχιτεκτονική της μνήμης. Λαμβάνοντας υπόψη ότι η CPU ARM A53 είναι δύο φορές πιο γρήγορη από την ARM A35, μπορούμε να υποθέσουμε ότι η μνήμη RAM LPDDR3 στο TPU Mini συμβάλλει στην προστιθέμενη καθυστέρηση.

Μεθοδολογία για εκτέλεση στο Edge TPU

Η εκτέλεση ενός μοντέλου ΤΝ στον επιταχυντή Edge TPU [9] βασίζεται στο **TensorFlow Lite Interpreter API**. Ο TensorFlow Lite Interpreter έχει σχεδιαστεί για να είναι γρήγορος και ευέλικτος. Χρησιμοποιεί μια σειρά στατικών γράφων και έναν προσαρμοσμένο, λιγότερο δυναμικό κατανεμητή μνήμης για να εξασφαλίσει μικρότερο φόρτο εργασίας, ταχύτερη αρχικοποίηση και λιγότερη καθυστέρηση εκτέλεσης. Ωστόσο, για να γίνει ακόμα πιο εύκολη η ανάπτυξη κώδικα στο TPU, η Coral δημιούργησε μια βιβλιοθήκη wrapper, το **Pycoral API**, για να χειριστεί πολλούς κώδικες που απαιτούνται κατά την εκτέλεση επαναλήψεων με το TensorFlow Lite. Το Pycoral API είναι χτισμένο πάνω στο TensorFlow Lite Python API για να απλοποιήσει τον κώδικα κατά την εκτέλεση ενός μοντέλου στο Edge TPU και να παρέχει προηγμένες δυνατότητες, όπως η εκτέλεση σε πολλαπλές Edge TPU με pipeline.

Η εκτέλεση και επιτάχυνση μοντέλων ΤΝ στο Edge TPU με το Pycoral API και τον TensorFlow Lite Interpreter περιλαμβάνει τα ακόλουθα βήματα, όπως φαίνεται στο Σχήμα 6. Αυτά τα βήματα είναι σταθερά για οποιοδήποτε μοντέλο θέλουμε να εκτελέσουμε στον επιταχυντή TPU. Διαφοροποίηση μπορεί να προκύψει ανάλογα με τα δεδομένα εισόδου.

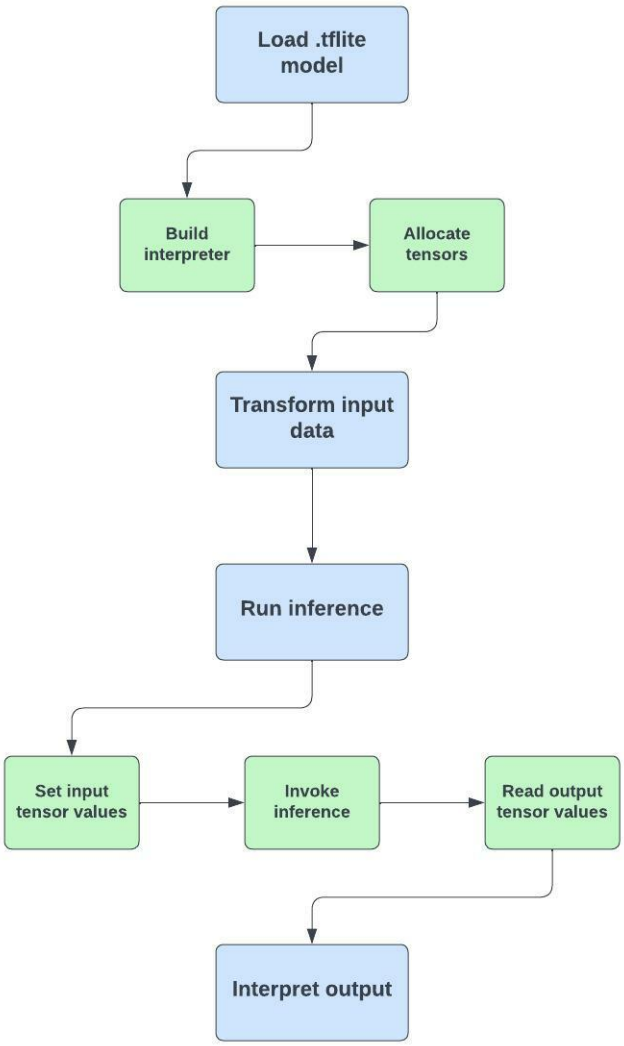
1. Φόρτωση του μοντέλου .tflite, που περιέχει τον γράφο εκτέλεσης του μοντέλου, στη μνήμη.

2. Μετατροπή των δεδομένων εισόδου. Τα ακατέργαστα δεδομένα εισόδου για το μοντέλο δεν ταιριάζουν με τη μορφή δεδομένων εισόδου που αναμένονται από το TF Lite. Συνήθως, είναι απαραίτητο να αλλάξουμε το μέγεθος μιας εικόνας ή τη μορφή της ώστε να είναι συμβατή με το μοντέλο.

3. Επαναληπτική εκτέλεση του μοντέλου μέσω TensorFlow Lite API. Περιλαμβάνει τα ακόλουθα βήματα:

   - Δημιουργία του Interpreter για το υπάρχον μοντέλο.
   - Κατανομή των τανυστών.
   - Ορισμός των τιμών των εισόδων του μοντέλου βάσει της χβάντισης που πραγματοποιήσαμε.
   - Κλήση του Interpreter.
   - Λήψη και διάβασμα των τιμών των εξόδων του μοντέλου.

4. Ερμηνεία των δεδομένων εξόδου με τέτοιο τρόπο που να είναι κατανοητά για την εκάστοτε εφαρμογή.



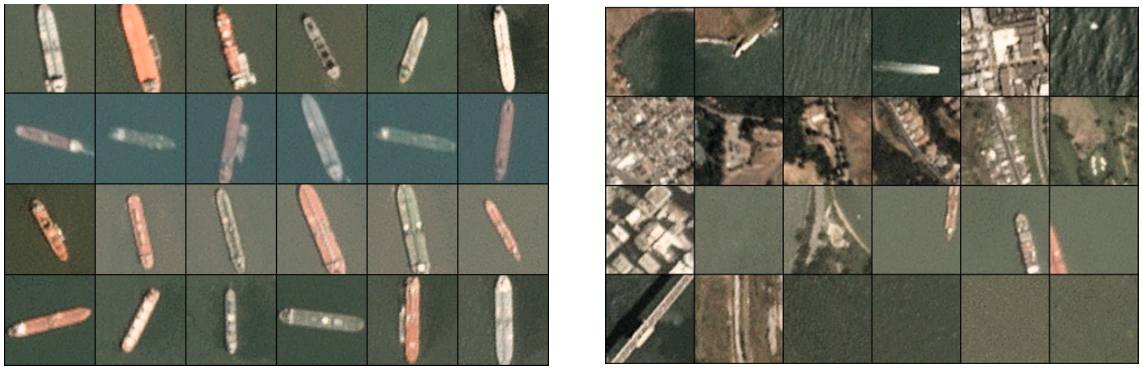
Σχήμα 6: Διαδικασία εκτέλεσης στο TPU

ΣΝΔ για Αναγνώριση Πλοίων

Σε αυτήν την ενότητα, θα παρουσιάσουμε την εφαρμογή ενός μοντέλου ανίχνευσης αντικειμένων σε εικόνες στο Edge TPU. Θα χρησιμοποιήσουμε ένα συνελικτικό νευρωνικό δίκτυο (ΣΝΔ) που εκτελεί ανίχνευση πλοίων σε δορυφορικές εικόνες προκειμένου να αξιολογήσουμε την απόδοση του επιταχυντή TPU σε μια πραγματική εφαρμογή. Επίσης, στο ίδιο πλαίσιο, θα παρουσιάσουμε μια δοκιμαστική εφαρμογή ανίχνευσης πλοίων σε εικόνες μεγάλης κλίμακας από παραθαλλάσιες περιοχές στην Καλιφόρνια.

Το νευρωνικό δίκτυο που θα χρησιμοποιήσουμε έχει ήδη παρουσιαστεί σε προηγούμενα έργα του Microlab [10–12] του ΕΜΠ. Είναι ένα ΣΝΔ βάθους 6 επιπέδων, που αποτελείται από 4 δισδιάστατα επίπεδα συνέλιξης + 4 επίπεδα max pooling και 2 πλήρως συνδεδεμένα στρώματα (FC layers) ακολουθούμενα από ένα στρώμα softmax για έξοδο.

Το συνελικτικό νευρωνικό μας δίκτυο έχει εκπαιδευτεί σε ένα σύνολο δεδομένων που προέρχεται από το kaggle [13] και αποτελείται από εικόνες που εξάγονται από δορυφορικές λήψεις στο διάστημα, που συλλέχθηκαν πάνω από τις περιοχές San Francisco Bay και San Pedro Bay της Καλιφόρνια. Το σύνολο δεδομένων περιλαμβάνει 4000 εικόνες 80x80 RGB με ετικέτα για κατηγοριοποίηση ως ΄πλοίο΄ είτε ως όχι πλοίο΄. Παραδείγματα εικόνων από τις δύο κατηγορίες φαίνονται στο Σχήμα 7.



(α΄) Κατηγορία ΄πλοίο΄ (β΄) Κατηγορία ΄όχι πλοίο΄

Σχήμα 7: Κατηγοριοποίηση του Dataset

```
Edge TPU Compiler version 16.0.384591198
Input: ship_tf_quant.tflite
Output: ship_tf_quant_edgetpu.tflite

Operator                       Count      Status

SOFTMAX                        1          Mapped to Edge TPU
RESHAPE                        1          Mapped to Edge TPU
CONV_2D                        4          Mapped to Edge TPU
FULLY_CONNECTED                2          Mapped to Edge TPU
QUANTIZE                       2          Mapped to Edge TPU
MAX_POOL_2D                    4          Mapped to Edge TPU
```

Σχήμα 8: Αρχείο καταγραφής του Edge TPU Compiler

Για να εξετάσουμε την ακρίβεια του μοντέλου μας, θα τρέξουμε 2000 επαναλήψεις στο TPU για εικόνες του συνόλου δεδομένων. Συγκεκριμένα, φορτώνουμε 1000 εικόνες και από τις δύο κατηγορίες και αναμένουμε πρόβλεψη για κάθε μία από αυτές. Η ακρίβεια και η απόδοση του μοντέλου μας, είτε εκτελείται στη CPU είτε στο TPU, κβαντισμένο ή όχι, παρουσιάζεται στον Πίνακα 2. Σχετικές εργασίες για το ίδιο σύνολο δεδομένων που αναπτύχθηκαν στον επιταχυντή Xilinx Virtex 7 XC7VX485T FPGA έχουν δείξει παρόμοια απόδοση (χρόνος εκτέλεσης ίσος με 0.687 ms) για υψηλότερη κατανάλωση ενέργειας (5 Watts) [14], ενώ το ίδιο ΣΝΔ για ανίχνευση πλοίων έχει επιτύχει 725 FPS σε Xilinx Zynq Z-7020 FPGA [15] στα 4 Watts ισχύος [12].

Πίνακας 2: Απόδοση & Ακρίβεια

Μοντέλο	Είδος χβάντισης	Inference	Ακρίβεια	Καθυστέρηση (ms)	Throughput (FPS)
TFLite	καμία	CPU	87.8 %	36	28
TFLite	uint8	CPU	99.4 %	22	45.5
TFLite Compiled	uint8	Edge TPU	99.4 %	0.5-1	1000-2000

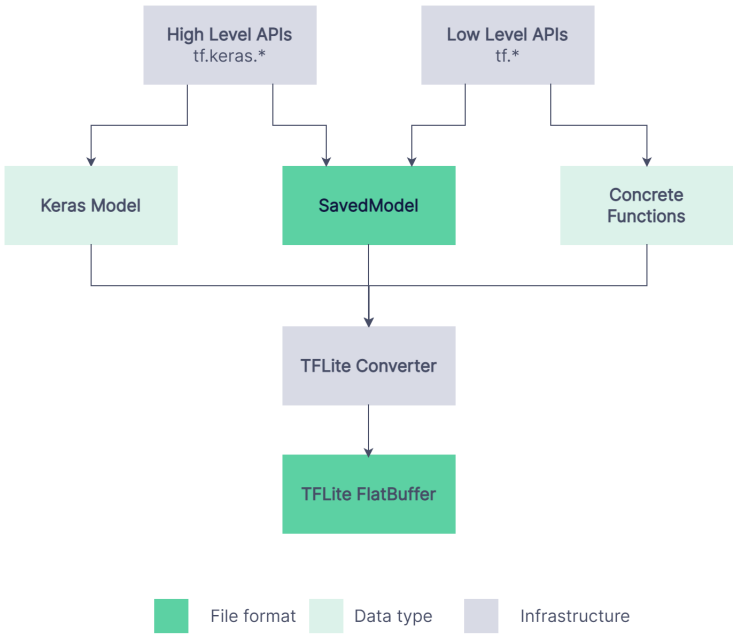
Χρησιμοποιώντας το ίδιο μοντέλο, θα παρουσιάσουμε μια δοκιμαστική εφαρμογή για ανίχνευση πλοίων. Στόχος μας είναι να εντοπίσουμε περιοχές ενδιαφέροντος σε εικόνες μεγάλης κλίμακας από παραθαλλάσιες περιοχές. Για το σκοπό αυτό, δημιουργήσαμε μια συνάρτηση που χωρίζει μια εικόνα σε μικρότερες εικόνες, επιλεγμένου πλάτους και ύψους με προκαθορισμένη επικάλυψη, τόσο οριζόντια όσο και κάθετα, έτσι ώστε να μην χάνουμε καμία σημαντική πληροφορία κατά τη διάρκεια αυτής της διαδικασίας. Για παράδειγμα, μεγάλες εικόνες όπως στο Σχήμα 9, με ανάλυση 2844x1828, μπορούν να χωριστούν σε **12460** 80x80 εικόνες με επικάλυψη ίση με 60 pixel, όπως στο Σχήμα 10.

Σχήμα 9: Long Beach bay area (2844ξ1828)

Σχήμα 10: Αποτέλεσμα εφαρμογής για ανίχνευση πλοίων

Το μοντέλο μας επιταχύνθηκε με το Edge TPU και μπορεί να παράξει πρόβλεψη για οποιαδήποτε εικόνα εισόδου 80x80 σε 0,5-1 χιλιοστό του δευτερολέπτου (ms). Έτσι, με throughput 1000 FPS, μπορεί να επιστρέψει ακριβείς προβλέψεις για εικόνες πλοίων για 12460 εισόδους σε λιγότερο από 12,5 δευτερόλεπτα.

Υπολογισμοί Γενικού Σκοπού



Σχήμα 11: Διαδικασία μετατροπής σε TF Lite

Σε αυτή την ενότητα, θα παρουσιάσουμε την υλοποίησή μας για υπολογισμούς γενικού σκοπού στο Edge TPU. Έχουμε ήδη συζητήσει ότι η κατασκευή ενός συμβατού μοντέλου

για το TPU μπορεί να γίνει με δύο τρόπους, είτε με APIs υψηλού επιπέδου είτε με χαμηλού επιπέδου (Σχήμα 11). Στην πρώτη περίπτωση, μπορούμε να κατασκευάσουμε οποιοδήποτε μοντέλο στο TensorFlow χρησιμοποιώντας τη βιβλιοθήκη Keras, η οποία προσφέρει πρόσβαση σε πολλές υλοποιήσεις δομικών στοιχείων νευρωνικών δικτύων που χρησιμοποιούνται συνήθως. Από την άλλη πλευρά, ο TF Lite Converter μπορεί να μετατρέψει και οποιοδήποτε function ή operation που έχει τη μορφή TensorFlow γράφου. Ο γράφος προκύπτει με τη χρήση του tf.function, ενός εργαλείου, δηλαδή, που μετατρέπει οποιοδήποτε σύνολο πράξεων σε μία συνάρτηση, την οποία μπορούμε να καλούμε με διαφορετικά ορίσματα κάθε φορά. Για κάθε όρισμα, η συνάρτηση αποθηκεύει έναν γράφο και με τη χρήση ενός wrapper, όπως είναι το Concretefunction, μετατρέπεται σε μορφή .tflite.

Listing 3.4: Concrete function

```
@tf.function
def add(x1, x2):
  return tf.add(x1,x2)

tensor = add.get_concrete_function(tf.ones([4, 1]), tf.ones([4,
   1]))
```

Σχήμα 12: Concrete function

Πρόσθεση & Πολλαπλασιασμός

Υπό αυτό το πρίσμα, θα δείξουμε πως μπορούμε να εκτελέσουμε πρόσθεση και πολλαπλασιασμό στο TPU. Όπως γνωρίζουμε, το TPU επιτυγχάνει μέγιστη απόδοση για μοντέλα που είναι πλήρως κβαντισμένα σε 8-bit ακέραιους αριθμούς. Αυτό σημαίνει ότι τόσο οι είσοδοι όσο και οι έξοδοι τους, πρέπει να είναι 8-bit ακέραιοι αριθμοί, δηλαδά να ανήκουν στο εύρος από 0 έως 255.

Στην μία περίπτωση, από τη πρόσθεση δύο 8-bit ακεραίων προκύπτει πάντα ένας 9-bit ακέραιος αριθμός, ο οποίος, όπως αναφέραμε, πρέπει να αναπαρασταθεί ως 8-bit. Αυτό σημαίνει ότι, ενώ όλα τα δυνατά αποτελέσματα μίας πρόσθεσης είναι 511, αυτά πρέπει να αντιστοιχιστούν σε 256 θέσεις, δηλαδή στη συγκεκριμένη περίπτωση κάθε αποτέλεσμα αντιστοιχίζεται με 2 πιθανά. Αντίστοιχα, στον πολλαπλασιασμό, το αποτέλεσμα που προκύπτει είναι ένας 16-bit ακέραιος αριθμός, με 65026 πιθανά αποτελέσματα που πρέπει να αντιστοιχιστούν σε 256 θέσεις. Άρα σε κάθε περίπτωση, το αποτελεσμα από την εκτέλεση στο TPU θα είναι μεταξύ 0 και 256 και πρεπει με το κατάλληλο scale να μας δώσει το επιθυμητό αποτέλεσμα, ή έστω μία προσέγγιση αυτού. Στην περίπτωση της πρόσθεσης, ο παράγοντας αυτός είναι το 2, ενώ στον πολλαπλασιασμό το 255. Εύκολα αναλογιζόμαστε την μείωση της ακρίβειας των πράξεων στο TPU ακόμα και για 8-bit αριθμούς. Στην περίπτωση αριθμών περισσότερων bit, η μείωση αυτή θα είναι ακόμα πιο σημαντική.

Πολλαπλασιασμός Πινάκων

Η πράξη του πολλαπλασιασμού πινάκων συνδυάζει και πολλαπλασιασμό αλλά και πρόσθεση. Συγκεκριμένα, ο πολλαπλασιασμός NxN πινάκων 8-bit ακέραιων αριθμών απαιτεί να εκτελεστούν N πολλαπλασιασμοί και N-1 προσθέσεις. Επομένως, σύμφωνα με όσα αναλύσαμε προηγουμένως, το αποτέλεσμα του πολλαπλασιασμού πινάκων πρέπει να πολλαπλασιαστεί με το κατάλληλο παράγοντα scale για να λάβουμε το επιθυμητό αποτέλεσμα. Σε αυτή τη περίπτωση, οι πολλαπλασιασμοί των στοιχείων γίνονται ταυτόχρονα, άρα ο παράγοντας θα είναι 255, ενώ οι προσθέσεις γίνονται σειριακά, οπότε ο παράγοντας scale για την πρόσθεση θα είναι N. Αρα για NxN πίνακες, η έξοδος του TPU πρέπει να πολλαπλασιαστεί με τον παράγοντα 255 επί N.

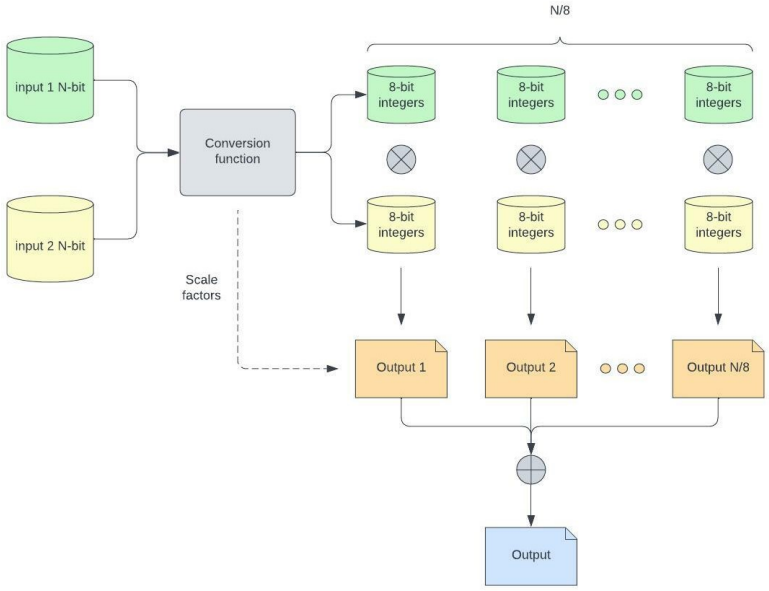
$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in} + b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj} \tag{2}$$

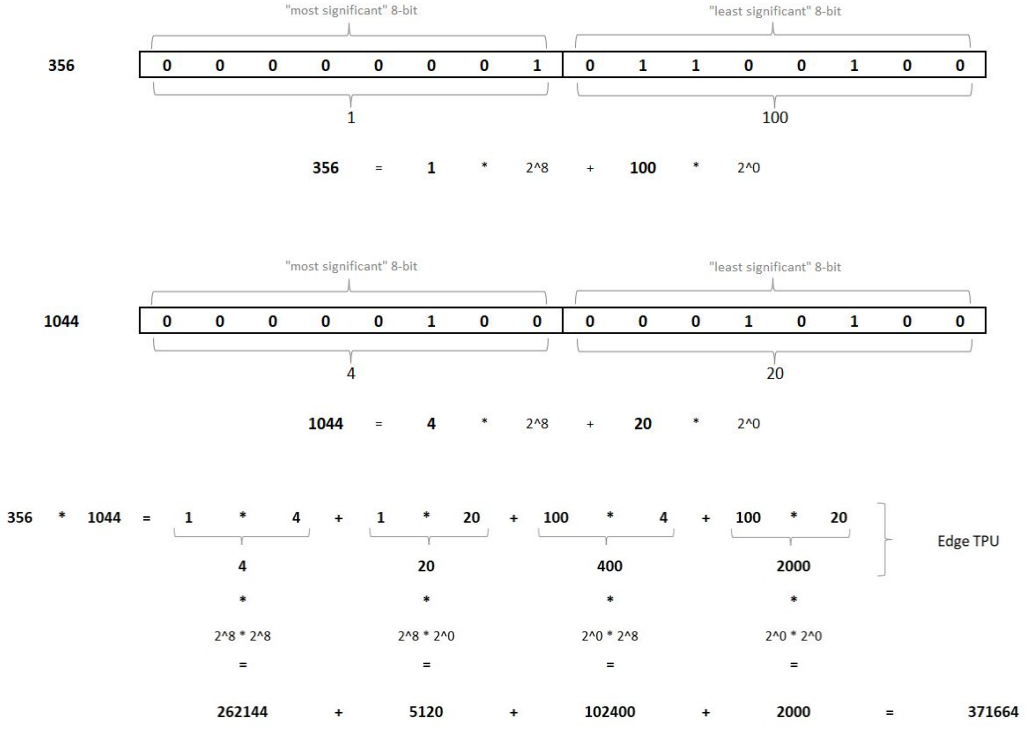
Πράξεις με N-bit αριθμούς

Και στις δύο εφαρμογές που παρουσιάσαμε προηγουμένως, παρατηρήσαμε ότι η εκτέλεση αλγεβρικών πράξεων με μέγεθος bit ίσο με 8 ή μεγαλύτερο, προσθέτει σημαντικό σφάλμα στους υπολογισμούς. Για αυτό το λόγο, θέλουμε να προτείνουμε μία δική μας λύση έτσι ώστε να πραγματοποιήσουμε τόσο element-wise πράξεις όσο και πολλαπλασιασμό πινάκων χωρίς να θυσιάζουμε σημαντικά την ακρίβεια. Έτσι, **η πρότασή μας θα έχει να κάνει με το σπάσιμο ακεραίων N-bit σε πολλαπλούς ακέραιους αριθμούς μεγέθους 8-bit.**

Για αυτό το σκοπό αναπτύξαμε μια συνάρτηση που σπάει αριθμούς σε 2 μέρη κάθε φορά, μέχρις ότου όλα τα μέρη να έχουν μέγεθος 8-bit. Για να αναγνωρίσουμε κάθε μέρος του αριθμού που σπάμε, πραγματοποιούμε δεξιά ολίσθηση για το πρώτο μέρος και modulo διαίρεση για το δεύτερο. Η διαδικασία που αναφέραμε παρουσιάζεται στο ακόλουθο παράδειγμα (Σχήμα 13 & 14).



Σχήμα 13: Πράξεις N-bit στο TPU

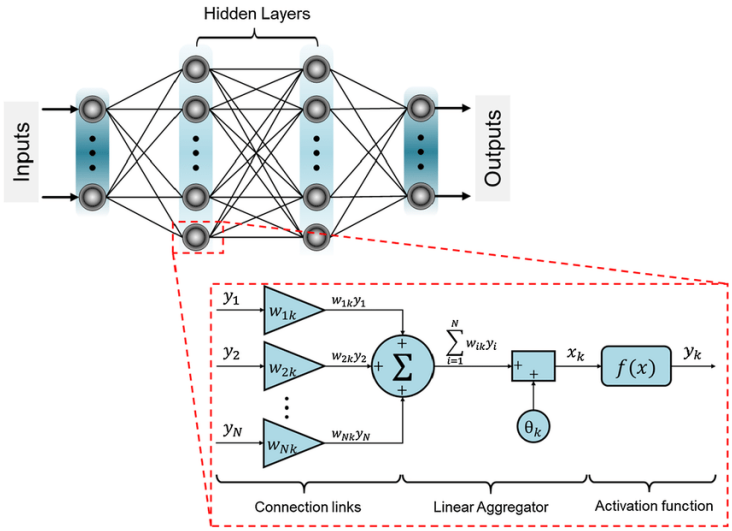


Σχήμα 14: Παράδειγμα: Σπάσιμο 16-bit αριθμών σε 8-bit

Εναλλακτικές υλοποιήσεις

Ως μέρος της αξιολόγησης του TPU και με στόχο να λάβουμε συμπεράσματα για την αρχιτεκτονική του επεξεργαστή, υλοποιήσαμε κάποια εναλλακτικά δίκτυα. Αφενός, θα αξιολογήσουμε την έκτελεση Multi Layer perceptron δικτύων (Σχήμα 15) στο TPU, τα οποία

χρησιμοποιούνται πολύ συχνά σε δίκτυα για classification και αναπαριστώνται ως πολλαπλασιασμός πινάκων. Αφετέρου, δημιουργήσαμε δικά μας μοντέλα που περιλαμβάνουν διαφορετικό αριθμό πράξεων και εισόδων/εξόδων (Σχήμα 16), που αν και δεν έχουν κάποια ρεαλιστική εφαρμογή θα μας βοηθήσουν να κατανοήσουμε την επίδραση αυτών των παραμέτρων στον χρόνο εκτέλεσης στο TPU.



Σχήμα 15: Fully connected νευρωνικό δίκτυο

Listing 3.12: Multiple muls in parallel

```
@tf.function
def parallel(x1,x2,x3,x4,...)
    :
  y1 = tf.mul(x1,x2)
  y2 = tf.mul(x3,x4)
  ...
  return [y1,y2,...]
```

Listing 3.13: Multiple muls in single output

```
@tf.function
def single_output(x1,x2,x3,x4
    ):
  y1 = tf.mul(x1,x2)
  y2 = tf.mul(x3,x4)
  y3 = tf.add(y1,y2)
  return y3
```

Listing 3.14: Multiple connected muls

```
@tf.function
def connected_muls(x1, x2):
  y1 = tf.mul(x1,x2)
  y2 = tf.mul(y1,x1)
  y3 = tf.mul(y2,x1)
  ...
  yN = tf.mul(y(N-1),x1)
return yN
```

Σχήμα 16: Δίκτυα με πολλαπλές πράξεις και εισόδους/εξόδους

Πειραματικά αποτελέσματα

Σε αυτή την ενότητα, θα παρουσιάσουμε τα πειραματικά αποτελέσματα για τις προτεινόμενες υλοποιήσεις στο Edge TPU.

Πρόσθεση & Πολλαπλασιασμός

Το Σχήμα 17 συγκρίνει τον μέσο χρόνο εκτέλεσης ανά πλήθος αριθμών μεταξύ της CPU ARM A53 και του Edge TPU για την πράξη του πολλαπλασιασμού. Τα αποτελέσματα οδηγούν σε πέντε παρατηρήσεις για το Edge TPU.

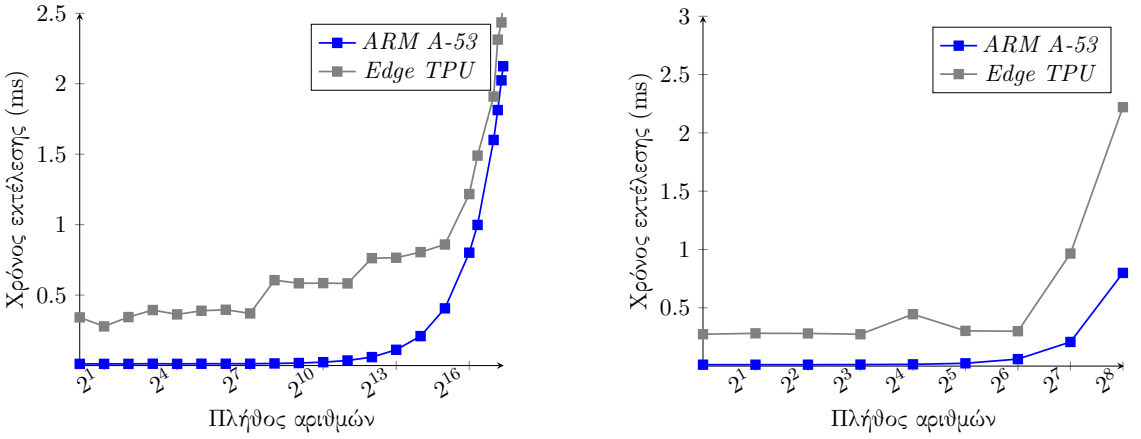
1. Η εκτέλεση στο TPU απαιτεί κατ' ελάχιστο 0,25-0,3 ms ως χρόνο εκτέλεσης, που αφορά τη μεταφορά δεδομένων και την επικοινωνία με τη μνήμη.

2. Οι υπολογισμοί γραμμικής άλγεβρας που κυριαρχούνται από πράξεις στοιχείου προς στοιχείο δεν αποδίδουν καλά στις TPU, επειδή δεν εκμεταλλεύονται τα πλεονεκτήματα της συστολικής συστοιχίας του επεξεργαστή.

3. Οι πράξεις με τανυστές διαστάσεων NxN είναι πιο αποδοτικές και πιο γρήγορες σε σχέση με τις Nx1.

4. Το ASIC του Edge TPU βασίζεται σε μια συστολική σειρά πολλαπλασιαστών και συσσωρευτών, που μπορούν να εκτελέσουν μια σειρά λειτουργιών παράλληλα. Η εκτίμησή μας για συστοιχία διαστάσεων 64x64 φαίνεται να επιβεβαιώνεται με βάση τα αποτελέσματά μας, καθώς ο χρόνος εκτέλεσης για συστοιχίες NxN αυξάνεται εκθετικά για διαστάσεις μεγαλύτερες από 64x64.

5. Το Edge TPU δεν μπορεί να εξυπηρετήσει πράξεις πινάκων διαστάσεων μεγαλύτερες από 2^8x2^8.



Σχήμα 17: Nx1 vs NxN πολλαπλασιασμοί

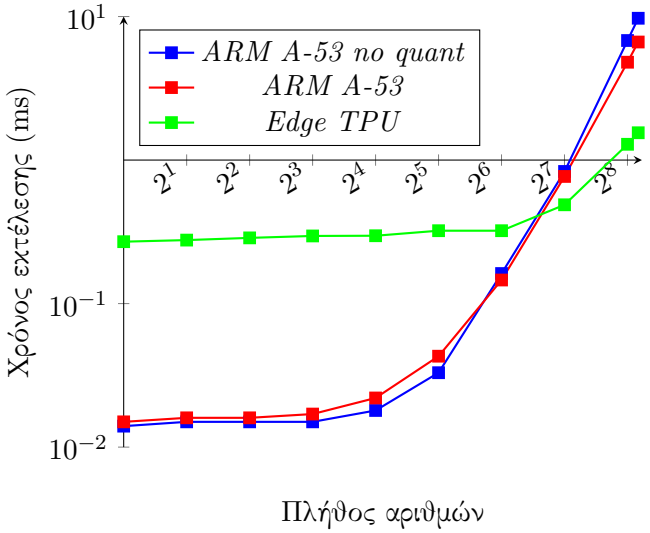
Όπως έχουμε ήδη αναφέρει, εκτελώντας αλγεβρικές πράξεις στο TPU προϋποθέτει κάποιους συμβιβασμούς όσον αφορά την ακρίβεια, ακόμη και αν οι είσοδοι είναι 8-bit ακέραιοι αριθμοί. Επομένως, στον Πίνακα 3 παρουσιάζουμε το μέσο σχετικό σφάλμα για την πράξη του πολλαπλασιασμού και της πρόσθεσης για εισόδους 8, 16 και 32-bit. Το μέγεθος της δειγματοληψίας που οδηγεί σε αυτά τα αποτελέσματα είναι 170000 δείγματα.

Πίνακας 3: Μέσο σχετικό σφάλμα για N-bit *πολλαπλασιασμούς*

Πλήθος bit	**8-bit ακέραιοι**	**16-bit ακέραιοι**	**32-bit ακέραιοι**
Πολλαπλασιασμός			
MRE	2.9 %	4.6 %	4.6 %
Πρόσθεση			
MRE	0.28 %	1.1 %	1.1 %

Πολλαπλασιασμός Πινάκων

Το συμπέρασμα στο οποίο καταλήγουμε σχετικά με τον πολλαπλασιασμό πινάκων στο TPU, είναι ότι για διαστάσεις τανυστή εισόδου μικρότερες από 64x64, ο χρόνος εκτέλεσης στο Edge TPU είναι μία (1) τάξη μεγέθους μεγαλύτερη από αυτή του επεξεργαστή ARM, ενώ για μεγαλύτερες διαστάσεις, η συστολική συστοιχία επιτυγχάνει μεγάλη παραλληλοποίηση που οδηγεί σε επιτάχυνση σε σχέση με τον ARM, 4 φορές ταχύτερα από το κβαντισμένο μοντέλο 8-bit και έως και 7 φορές πιο γρήγορα από το μοντέλο κινητής υποδιαστολής 32 bit στον ίδιο επεξεργαστή ARM.



Σχήμα 18: NxN πολλαπλασιασμός πινάκων

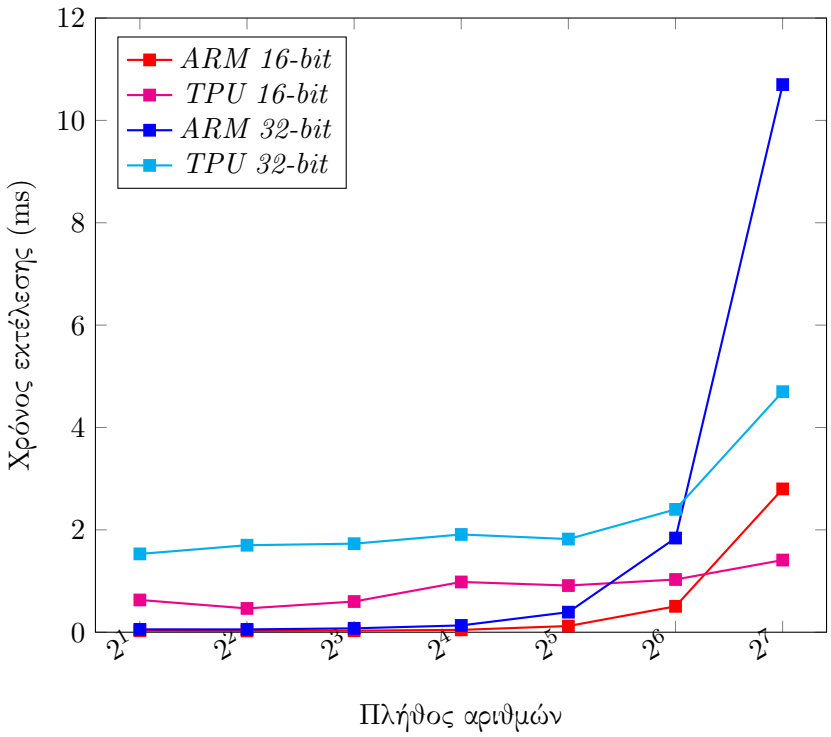
Αντίχτοιχα με τις element-wise πράξεις, η εκτέλεση πολλαπλασιασμού πινάκων στον επιταχυντή TPU θυσιάζει την ακρίβεια στο βωμό της απόδοσης. Ο Πίνακας 4 εμφανίζει το μέσο σχετικό σφάλμα των πολλαπλασιασμών πινάκων συγκριτικά με τις διαστάσεις του τανυστή εισόδου. Όσο μεγαλύτερο είναι το μέγεθος του πίνακα, τόσο περισσότερες προσθέσεις εκτελούνται, επομένως μεγαλύτερη απώλεια ακρίβειας. Ωστόσο, όσο μεγαλύτερο είναι το μέγεθος του πίνακα, τόσο μικρότερη συνεισφορά έχει κάθε πρόσθεση στο αποτέλεσμα, άρα και μικρότερη απώλεια ακρίβειας. Περιττό να πούμε ότι η μείωση της ακρίβειας στον πολλαπλασιασμό είναι υψηλότερη από την πρόσθεση, οπότε στο τέλος προτιμάται μεγαλύτερο μέγεθος πίνακα.

Πίνακας 4: Μέσο σχετικό σφάλμα για 8-bit *πολλαπλασιασμό πινάκων*

Μέγεθος πίνακα	Μέσο σφάλμα	Μέγιστο σφάλμα
1x1	2.27 %	100.0 %
2x2	0.95 %	100.0 %
4x4	0.52 %	21.1 %
8x8	0.45 %	3.5 %
16x16	0.41 %	1.6 %
32x32	0.41 %	1.4 %
64x64	0.40 %	1.14 %
128x128	0.39 %	0.99 %
256x256	0.40 %	0.95 %

Πράξεις με N-bit αριθμούς

Όσον αφορά την πρόταση μας για υλοποίηση υπολογισμών N-bit με σπάσιμο σε 8-bit, αφενός, για απλούς πολλαπλασιασμούς και προσθέσεις δεν παρατήρουμε κάποια επιτάχυνση από το TPU, αφετέρου, όμως, στον πολλαπλασιασμό πινάκων βλέπουμε σημνατική επιτάχυνση για μεγάλες διαστάσεις πινάκων, όπως φαίνεται στο Σχήμα 19.

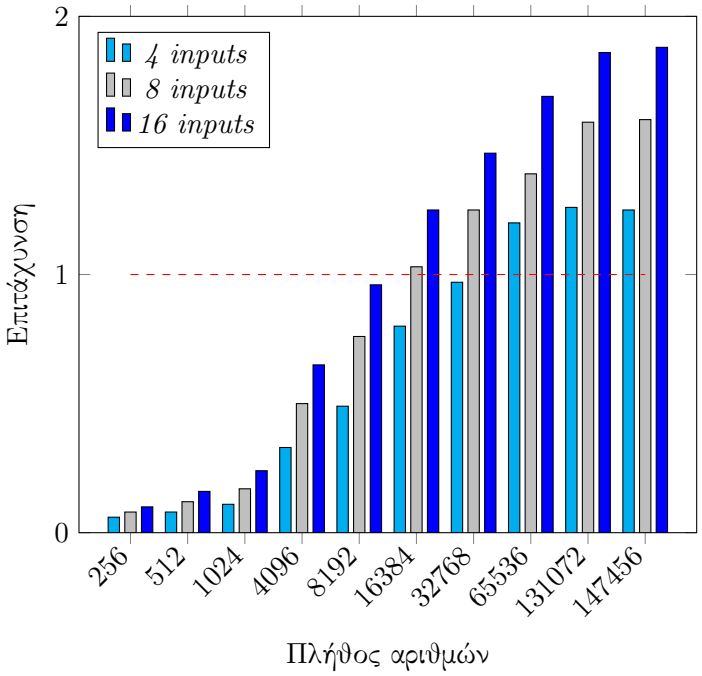


Σχήμα 19: N-bit πολλαπλασιασμοί πινάκων

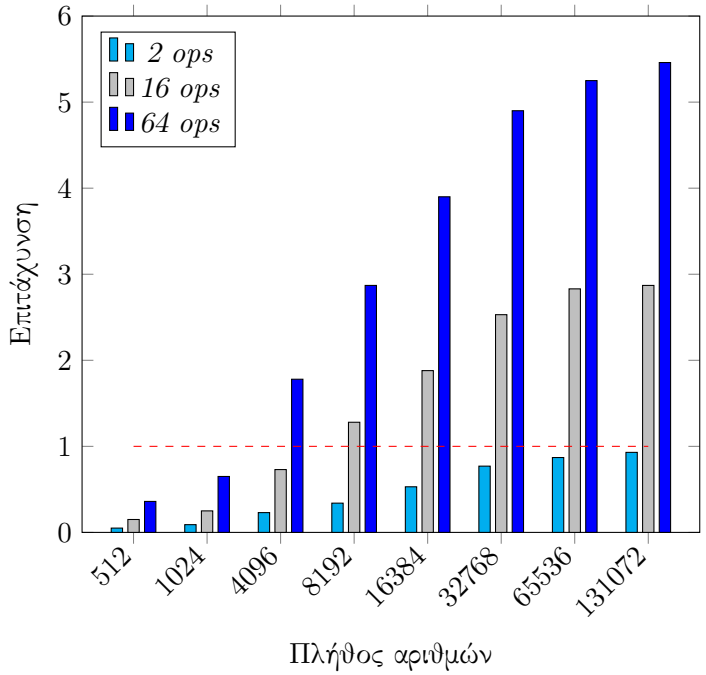
Αναφορικά με τα αποτελέσματα για το μέσο σχετικό σφάλμα των υλοποιήσεων που προτείνουμε, στην πράξη του πολλαπλασιασμού παρατήρουμε μείωση της απώλειας ακρίβειας κατά 45% για 16-bit αριθμούς, από 4,6% σε 2,5%, με μικρή αύξηση στον χρόνο εκτέλεσης. Στους πολλαπλασιασμούς πινάκων N-bit, από την άλλη, παρατηρούμε το ίδιο μέσο σφάλμα (0.4%) για πίνακες μεγάλων διαστάσεων όμοια με την υλοποίηση για 8-bit.

Εναλλακτικές υλοποιήσεις

Συνεχίζοντας, θα αξιολογήσουμε τα αποτελέσματα που προέκυψαν από τα custom μοντέλα που υλοποιήσαμε. Αρχικά, θα συγκρίνουμε δύο παρόμοια μοντέλα με τη διαφορά ότι το ένα από τα δύο δίνει μια έξοδο μονο. Αυτό που παρατηρούμε είναι ότι παρόλο που και οι δύο υλοποιήσεις έχουν τον ίδιο αριθμό από operations, αν συμπεριλάβουμε και τις κβαντίσεις, αυτή με τη μία έξοδο, όχι μόνο εκτελείται ταχύτερα από την άλλη, αλλά επιτυγχάνει επιτάχυνση από το Edge TPU, όπως φαίνεται στο Σχήμα 20. Το δεύτερο μοντέλο που υλοποιήσαμε, μας βοηθάει να συμπεράνουμε πως το πλήθος των πράξεων που εκτελούνται όλες μάζι, χωρίς να μεσολαβεί πρόσβαση στη μνήμη, επιτυγχάνει σημαντική επιτάχυνση στο TPU, έως και 6 φορές ταχύτερα σε σχέση με τον ARM (Σχήμα 21). Όσο μεγαλύτερο το πλήθος των πράξεων και όσο μεγαλύτερες οι διαστάσεις των τανυστών, τόσο μεγαλύτερη η επιτάχυνση.



Σχήμα 20: Edge TPU επιτάχυνση vs ARM



Σχήμα 21: Edge TPU επιτάχυνση vs ARM

Εφαρμογές Ψηφιακής Επεξεργασίας Εικόνων

Ανιχνευτής ακμών Sobel

Ολοκληρώνοντας την παρούσα διπλωματική εργασία, θα παρουσιάσουμε την υλοποίηση δύο εφαρμογών ψηφιακής επεξεργασίας εικόνων στο TPU. Πρώτα, θα αναφερθούμε στον ανιχνευτή ακμών Sobel. Ανήκει στη κατηγορία των γραμμικών φίλτρων και πραγματοποιεί ανίχνευση ακμών σε ασπρόμαυρες εικόνες. Η διαδικασία με την οποία πραγματοποιεί ανίχνευση ακμών έχει να κάνει με την συνέλιξη της εικόνας με δύο πυρήνες διαστάσεων 3x3, ένας για τον οριζόντιο άξονα και ένας για τον κατακόρυφο (Σχήμα 22). Οι πυρήνες είναι έτσι ορισμένοι ώστε να επισημαίνουν περιοχές της εικόνας που τα γειτνιάζοντα pixel παρουσιάζουν μεγάλη διαφορά στις τιμές τους.

$\mathbf{G_x}$

-1	0	+1
-2	0	+2
-1	0	+1

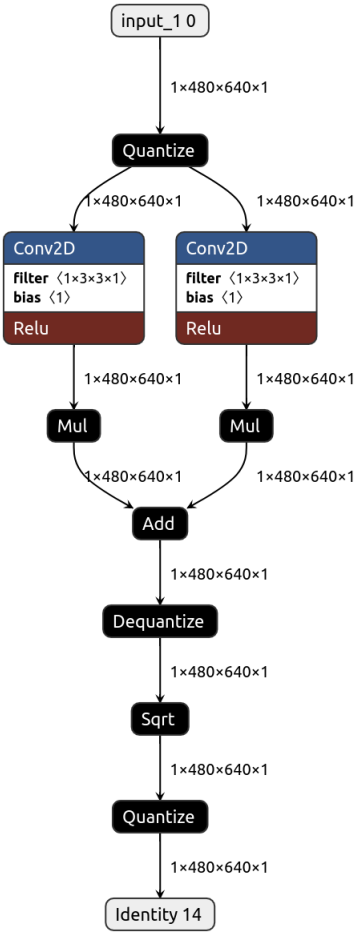
$\mathbf{G_y}$

-1	-2	-1
0	0	0
+1	+2	+1

Σχήμα 22: Πυρήνες Sobel

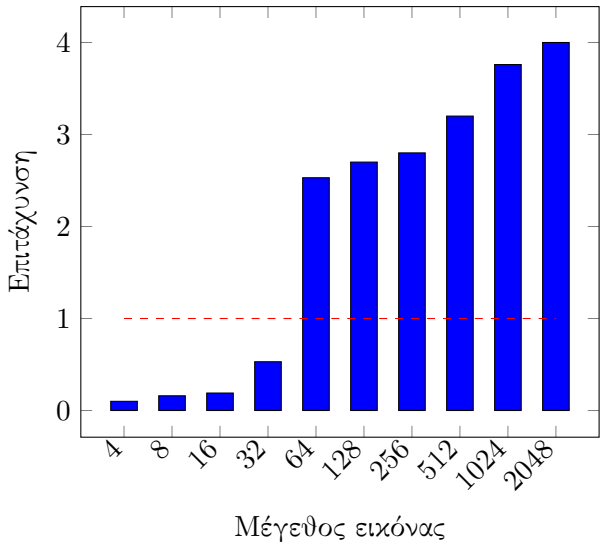
$$G = \sqrt{G_x^2 + G_y^2} \tag{3}$$

Η υλοποίηση στο TPU περιλαμβανει δύο συνελικτικά επίπεδα, ένα για κάθε κατεύθυνση και πράξεις γραμμικής άλγεβρας για να παραχθεί η τελική εικόνα. Οι πράξεις αυτές είναι ο πολλαπλασιασμός, η πρόσθεση και η τετραγωνική ρίζα. Το πρόβλημα αυτής της υλοποίησης είναι ότι η πράξη της τετραγωνικής ρίζας δεν υποστηρίζεται από το TPU, κάτι που σημαίνει ότι θα εκτελεστεί στη CPU, δηλαδή θα προσθέσει σημαντική καθυστέρηση στον χρόνο εκτέλεσης.

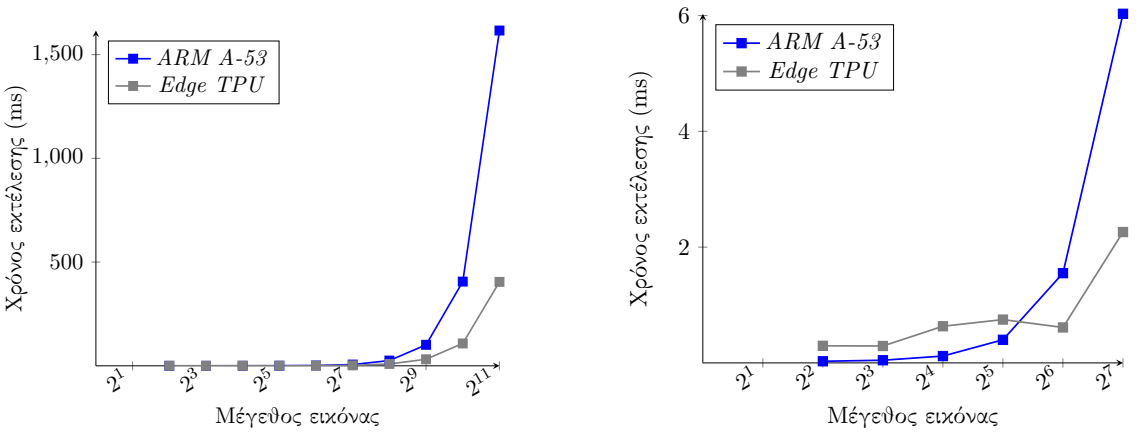


Σχήμα 23: Sobel TF Lite δίκτυο

Από τα πειραματικά αποτελέσματα, συμπεραίνουμε, ότι ο Sobel operator εκτελείται έως και 4 φορές πιο γρήγορα στο TPU σε σχέση με τον ARM για 2K εικόνες. Η εκτέλεση παραγματοποιείται σε 450 ms. Σε αυτό το σημείο, να τονίσουμε ότι αν δεν λαμβάναμε υπ΄ όψιν την πράξη της τετραγωνικής ρίζας, που εκτελείται στη CPU, ο χρόνος εκτέλεσης θα ήταν ο μισός.



Σχήμα 24: Επιτάχυνση του Sobel Operator στο Edge TPU vs ARM A-53



(α′) Συνολικά αποτελέσματα

(β′) Μικρότερα μεγέθη

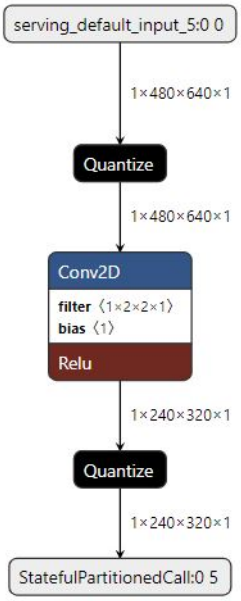
Σχήμα 25: Χρόνος εκτέλεσης Sobel operator

Binning

Το Binning είναι άλλη μία ευρέως διαδεδομένη εφαρμογή επεξεργασίας ψηφιακής εικόνας. Συνδυάζει γειτνιάζοντα pixel σε μια εικόνα και τα αντικαθιστά με ένα μόνο pixel ως τη μέση τιμή τους. Το αποτέλεσμα είναι να μειώνει την ανάλυση και το μέγεθος των εικόνων. Τέτοια φίλτρα εφαρμόζονται σε υλοποιήσεις που απαιτούν αυξημένη απόδοση και μειωμένη κατανάλωση ισχύος. Βρίσκει εφαρμογή σε τομείς όπως η αυτόνομη οδήγηση, στους κλάδους υγείας και στο διάστημα. Σε τέτοιες εφαρμογές, η ακρίβεια δεν παίζει πάντα τόσο σημαντικό ρόλο, όσο η ταχύτητα μετάδοσης και επεξεργασίας των δεδομένων.

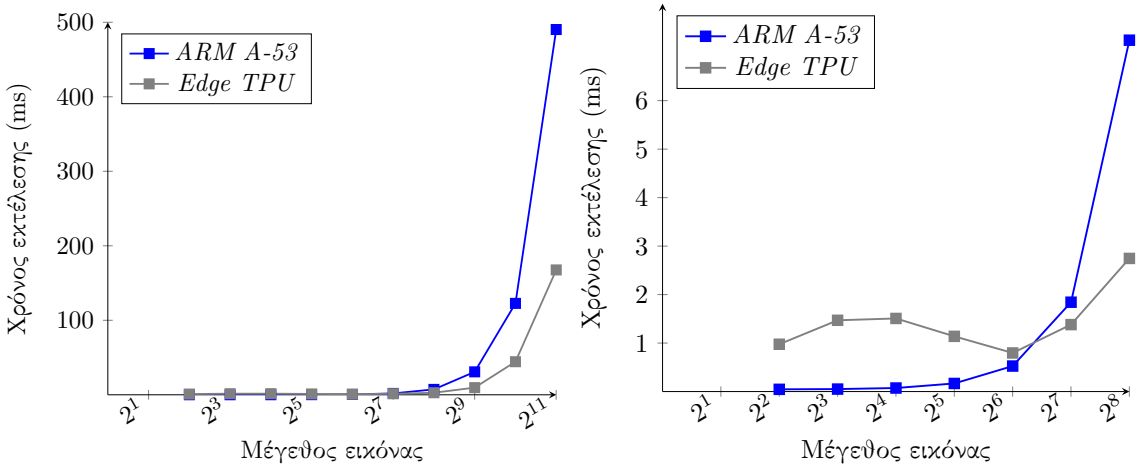
Η υλοποίηση που πραγματοποιήσαμε αποτελείται από ένα επίπεδο συνέλιξης. Όμως, το binning αφορά την επεξεργασία RGB εικόνων, δηλαδή εικόνων 3 καναλιών. Γι΄ αυτό το λόγο, αξιολογήσαμε δύο διαφορετικά σενάρια. Το πρώτο προϋποθέτει την παράλληλη επεξεργασία

και των 3 καναλιών σε μία εκτέλεση, άρα τρία (3) συνελικτικά στρώματα, ενώ το δεύτερο έχει να κάνει με σειριακό φιλτράρισμα των καναλιών, δηλαδή ένα (1) convolutional layer και 3 επαναλήψεις.



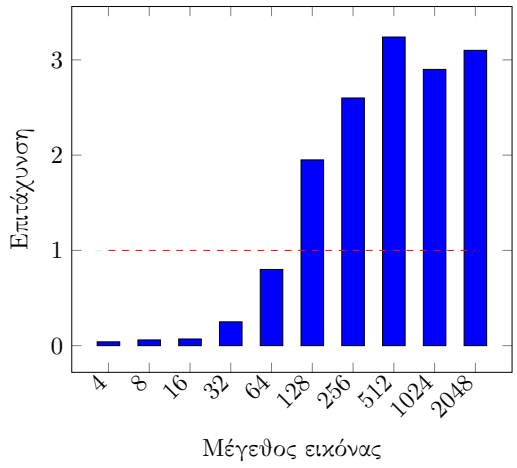
Σχήμα 26: Binning TF Lite δίκτυο

Τα πειραματικά αποτελέσματα δείχνουν ότι και οι δύο υλοποιήσεις, τελικά, έχουν παρόμοιο χρόνο εκτέλεσης για πραγματικά μεγέθη εικόνων και ειδικότερα για μεγέθη μεγαλύτερα από 512x512. Ωστόσο, για μικρότερα μεγέθη, παρατηρούμε ότι η παράλληλη υλοποίηση και των 3 επιπέδων συνέλιξης είναι σημαντικά ταχύτερη. Ο λόγος είναι, ότι για μικρότερα μεγέθη η εκτέλεση στο TPU έχει overhead, το οποίο τριπλασιάζεται όταν εκτελούμε 3 διαφορετικές επαναλήψεις. Κάτι τέτοιο δεν ισχύει στην παράλληλη υλοποίηση.

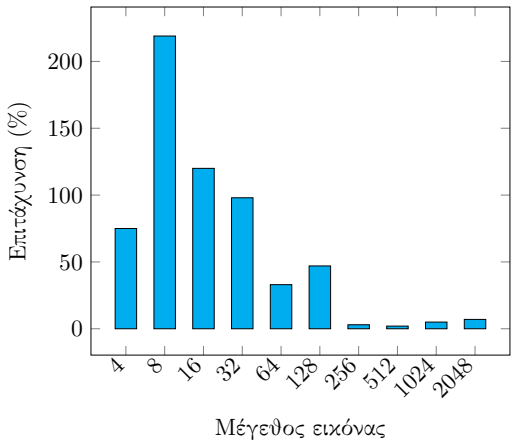


(α′) Συνολικά αποτελέσματα **(β′)** Μικρότερα μεγέθη

Σχήμα 27: Χρόνος εκτέλεσης Binning



(α′) Επιτάχυνση του Edge TPU vs ARM A-53



(β′) 1 επίπεδο vs 3 επίπεδα

Σχήμα 28: Σύγκριση υλοποιήσεων Binning

Συμπεράσματα

Συνοπτικά, τα πειραματικά αποτελέσματα αποκάλυψαν σημαντική βελτίωση της απόδοσης για το Google Edge TPU σε σύγκριση με τον επεξεργαστή ARM A53 και άλλους ενσωματωμένους επεξεργαστές. Αρχικά, τα πειράματά μας επαλήθευσαν την εκτιμήσεις ότι η αρχιτεκτονική του TPU βασίζεται σε systolic array διαστάσεων 64x64. Συνολικά, το Edge TPU επιτυγχάνει σημαντική επιτάχυνση για υπολογιστικά απαιτητικούς φόρτους εργασίας, όπως μεσαίου και μεγάλου μεγέθους ΣΝΔ και MLPs ή προσαρμοσμένα μοντέλα που κυριαρχούνται από πολλαπλασιασμούς πινάκων. Συγκριτικά με τη GPU Jetson Nano και την Intel Myriad X VPU, το Edge TPU επιτυγχάνει έως και 5 φορές καλύτερη απόδοση, ενώ μπορεί να αποδώσει εξίσου καλά με το Zynq ZCU104 FPGA. Η πράξη του πολλαπλασιασμού πινάκων βελτιώνεται έως και 4 φορές σε σύγκριση με την εκτέλεση κβαντισμένου μοντέλου 8-bit στον ARM και έως 7 φορές από το μοντέλο κινητής υποδιαστολής 32-bit. Τα αποτελέσματά μας δείχνουν επίσης ότι, πέρα από τα μικρά μεγέθη μοντέλων, ο επιταχυντής Edge TPU είναι πάντα ανώτερος από τον ARM A53 επιτυγχάνοντας έως και 100 φορές μεγαλύτερη απόδοση για μεσαία ΣΝΔ και έως 30 φορές για μεγάλα ΣΝΔ. Επιπλέον, για κλασικές εφαρμογές επεξεργασίας ψηφιακού σήματος (DSP), όπως ο ανιχνευτής ακμών Sobel και το Binning εικόνων, το Edge TPU παρέχει έως και 6 φορές καλύτερη απόδοση από τον ARM A53. Παρατηρήσαμε, περαιτέρω, ότι το πλεονέκτημα του Edge TPU έναντι του επεξεργαστή ARM μειώνεται μόλις το μέγεθος του μοντέλου φτάσει στο μέγεθος της μνήμης on-chip της πλατφόρμας Edge TPU. Το τελικό εύρημα των πειραμάτων μας, σχετικά με τον υπολογισμούς γενικού σκοπού, είναι ότι το TPU δεν αποδίδει καλά σε πράξεις ή μοντέλα που απαιτούν συχνή πρόσβαση στη μνήμη.

Η μελλοντική μας εργασία θα επικεντρωθεί στη δημιουργία των αναγκαίων framework με στόχο την ενσωμάτωσή του Edge TPU σε ετερογενή συστήματα για μελλοντικές διαστημικές εφαρμογές. Σε αυτό το πλαίσιο, εργαζόμαστε σε συνεργασία με τον Ευρωπαϊκό Οργανισμό Διαστήματος (ESA) για την ανάπτυξη του Edge TPU για μελλοντικές διαστημικές αποστολές.

Chapter 1

Introduction

Artificial intelligence (AI) is a wide-ranging branch of computer science concerned with building smart machines capable of performing tasks that typically require human intelligence. The major application of AI is machine learning (ML). The aim of ML is to allow machines to learn from experiences with large amounts of data without being programmed to do so. It synthesizes and interprets information for human understanding, according to pre-established parameters, helping to save time, reduce errors, create preventive actions and automate processes in large operations and companies. Workhorses to achieve the goal of automation are Neural Networks (NN). Neural networks reflect the behavior of the human brain, allowing computer programs to recognize patterns and solve common problems in the fields of AI, machine learning, and deep learning.

Neural networks target brain-like functionality and are based on a simple artificial neuron: a nonlinear function (such as max(0, value)) of a weighted sum of the inputs. These artificial neurons are collected into layers, with the outputs of one layer becoming the inputs of the next one in the sequence. The two phases of neural networks are called training (or learning) and inference (or prediction), and they refer to development versus production. The developer chooses the number of layers and the type of NN, and training determines the weights. Three kinds of neural networks [16, 17] are popular today:

1. Multi-Layer Perceptrons (MLP): Each new layer is a set of nonlinear functions of weighted sum of all outputs (fully connected) from a prior one, which reuses the weights.

2. Convolutional Neural Networks (CNN): Each ensuing layer is a set of of nonlinear functions of weighted sums of spatially nearby subsets of outputs from the prior layer, which also reuses the weights.

3. Recurrent Neural Networks (RNN): Each subsequent layer is a collection of nonlinear functions of weighted sums of outputs and the previous state. The most popular RNN is Long Short-Term Memory (LSTM). The art of the LSTM is in deciding what to forget and what to pass on as state to the next layer. The weights are reused across time steps.

In the recent years, artificial intelligence and machine learning applications have become increasingly popular and are being implemented to almost every aspect of our life. The neural networks pose great computational complexity and modern general-purpose CPUs struggle to deal with the demands. Also, as applications came online requiring real-time data processing and analysis, machine learning inference analysis is moved to the edge. Therefore, the increase in AI/ML workloads, along with breakdowns of several trends including Moore's Law, has prompted an explosion of embedded AI [18]. With embedded AI, devices have the ability to run AI models at the device level and then directly use the results to perform an appropriate task or action. These edge devices feature multiple processing units and specialized accelerators that promise even greater computational and machine learning capabilities. The hardware used for ML ranges from a single core chip to multi-core neural processing systems. All specialized processors are different from each other on the basis of hardware architecture and on-chip parallel processing.

Nowadays, four different processors are used for Machine Learning; starting from Graphics Processing Units (GPUs) [19] and Vision Processing Units (VPUs) [20, 21] we move to Field Programmable Gate Arrays (FPGAs) [22, 23] and Tensor Processing Units (TPUS) [24,25]. Our work will focus on evaluating novel AI accelerators, like the Google's Edge Tensor Processing Unit (Edge TPU) in comparison to the processors mentioned before, like Nvidia Jetson Nano GPU, Intel Myriad X VPU and Zynq ZCU104 MPSoc from related works presented in [26, 27], [10, 11, 28–32], [33] and [34].

The theory and the technology of AI accelerators [26,35] have been around for decades, but it has only been within the last years that the technology has been commercialized. The migration of processing from the cloud to the edge to embedded devices, which translates to a higher demand for heavy compute power at the local level, is also driving market growth.

Neural network accelerators, like the Edge TPU, offer power efficiency orders of magnitude better than that of conventional vector processors (e.g., Graphics Processing Units) for the same workloads. Despite the differences among micro-architectures, most accelerators are essentially matrix processors that take tensors-matrices as inputs, generate tensors-matrices as outputs, and provide operators that facilitate neural network computations. Graphics processing units are used to be just domain-specific accelerators. However, after intensive research into high-performance algorithms, architectures, and the availability of frameworks like CUDA and OpenCL, GPUs have been transformed into high-performance, general-purpose vector processors. Therefore, a similar revolution is expected to take place with AI accelerators, thus leading to a broader spectrum of applications.

However, democratizing these accelerators for non-AI/ML workloads will require tackling some issues:

1. The micro-architectures and instructions of NN accelerators are optimized for NN workloads, instead of general tensor algebra. These auxiliary NN accelerators focus on latency per inference, but not yet on delivering computation throughput compara-

ble to GPUs. Naively mapping conventional tensor algorithms to AI/ML operations will lead to sub-optimal performance.

2. Because many AI/ML applications are error tolerant, NN accelerators typically trade accuracy for area/energy-efficiency; when such a trade-off produces undesirable results, additional mechanisms are needed to make adjustments.

3. The programming interfaces of existing NN accelerators are specialized for developing AI/ML applications. Existing frameworks expose very few details about the hardware/software interfaces of NN accelerators, so programmers are unable to customize computation and the application can suffer from significant performance overhead due to adjusting the parameters/data bound to the supported ML models.

4. Tensor algorithms are traditionally time-consuming, so programmers have tailored compute kernels in favor of scalar/vector processing. Such tailoring makes applications unable to take advantage of tensor operators without revisiting algorithms.

1.1 Thesis Motivation & Contribution

The motivation of this diploma thesis is to explore and evaluate Edge TPU's capabilities for general-purpose computing and to provide solutions to imposed challenges. Our contribution to this thesis includes, at first, an extensive benchmarking on the TPU, including both pre-trained and custom networks, like our CNN for Ship Detection. Moreover, we provide solutions to the imposed challenges by proposing a custom methodology for building Edge TPU compatible networks for general-purpose calculations. We also propose a solution for overcoming the barrier of the 8-bit-only operations on the TPU by breaking N-bit algrebraic computations in 8-bit parts. Finally, we examine and implement applications for digital signal processing (DSP) on the Edge TPU and provide an overall evaluation for AI and general-purpose computing.

Chapter 2

Google Edge Tensor Processing Unit

2.1 Introduction to Edge TPU

The Edge TPU [36] is a small Application-Specific Integrated Circuit (ASIC) designed by Google that provides high performance machine learning inferencing for low-power devices. It delivers high performance in a small physical and power footprint, enabling the deployment of high accuracy AI at the edge. Edge TPU complements Cloud TPU and Google Cloud services [37–39] to provide an end-to-end, cloud-to-edge, hardware and software infrastructure for facilitating the deployment of customers' AI-based solutions. Edge TPU can be used for a growing number of industrial use-cases such as predictive maintenance, anomaly detection, machine vision, robotics, voice recognition, and many more. It can be used in manufacturing, on-premises, healthcare, retail, smart spaces and transportation

The Edge TPU allows developers to deploy high-quality machine learning inferencing [40] at the edge, using various prototyping and production products from Coral. Coral is a platform of hardware components, software tools, and pre-compiled machine learning models, allowing you to create local AI in any form-factor.

2.1.1 Coral Devices

Coral [41] is offering TPUs in the form of either prototyping or production products. Regarding the prototyping products, there are two development boards, i.e., Dev Board and Dev Board Mini [42] (Figure 2.2), which are single-board computers featuring: ARM processors, the edge TPU ASIC, I/O interfaces and various types of memory. There is also a USB accelerator, which integrates the edge TPU and can be attached to a host machine with Linux, Windows, or MAC OS and x86-64, Armv7-32, or Armv8-64 system architecture. Regarding the production products, there is a great variety of PCI-based TPU accelerators that can be integrated into any existing systems. Table 2.1 reveals the

tech specifications of the coral development boards we examined in this thesis.

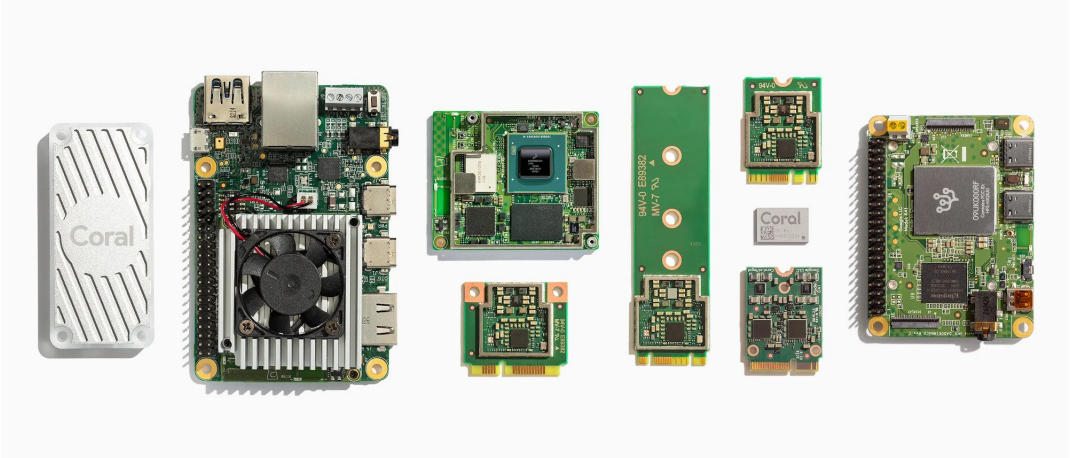
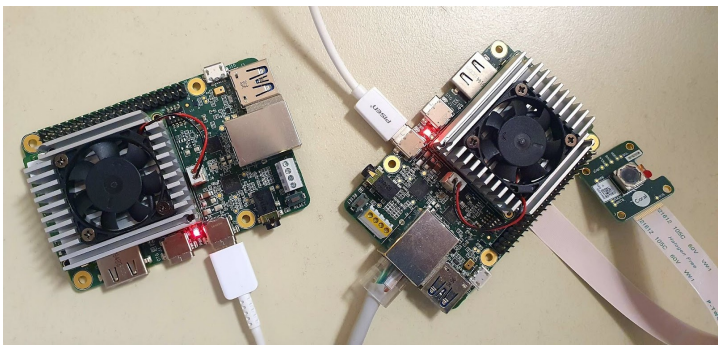
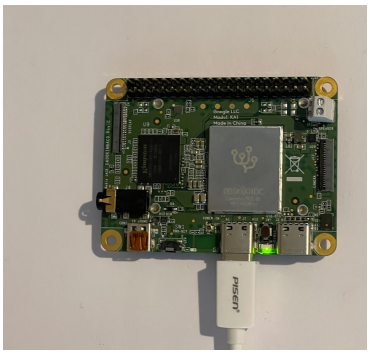


Figure 2.1: Coral products



(a) Dev Boards at NTUA



(b) Dev Board Mini at NTUA

Figure 2.2: Coral prototyping products at NTUA

Table 2.1: Tech specifications of Coral development boards

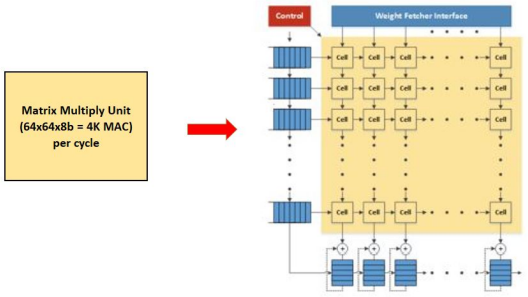
Specs	**Dev Board**	**Dev Board Mini**
CPU	NXP SoC (quad-core ARM Cortex-A53, Cortex-M4F)	MediaTek SoC (quad-core ARM Cortex-A35)
ML accelerator	Google Edge TPU coprocessor	Google Edge TPU coprocessor
RAM	1/4 GB LPDDR4	2 GB LPDDR3
CPU frequency	1500 Mhz	1300 Mhz

2.1.2 Edge TPU Architecture

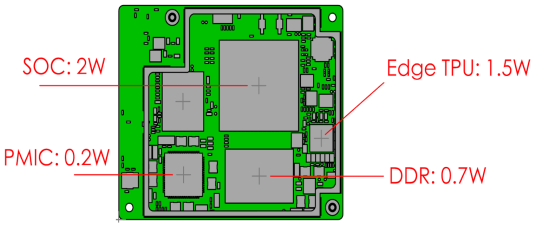
The Edge TPU microarchitecture accelerates Neural Network tasks for modern ML applications by creating a systolic array that performs operations on the units of tensors, in contrast to most NN applications that take tensor inputs and iteratively update parameters and weights from previous outcomes. Taking tensors as the default inputs and outputs makes the TPU architecture and its corresponding execution model fundamentally different from conventional CPU/GPU architectures that compute on scalar-vector data pairs.

The ASIC of Edge TPU is based on a systolic array of multipliers and accumulators, the dimensions of which have not been revealed yet, but according to Q-Engineering [5], it is estimated as an 64x64 array with f_{clk}=480MHz. An individual Edge TPU is capable of performing 4 trillion operations (tera-operations) per second (TOPS), using 0.5 watts for each TOPS (2 TOPS per watt). Running at 480 MHz it can theoretically perform 64 x 64 x 480.000.000 = 2 trillion mul-adds per second. Or if we look to individual operations, it is 4 trillion (4 TOPS).

In terms of memory, the Edge TPU incorporates large on-chip memory to hold the intermediate results that later iterations reuse. This enables faster inferencing speed compared to fetching the parameter data from an external memory. In other words, the TPU SRAM is used as "cache", even though it is a compiler-allocated scratchpad memory. Unlike conventional processors, the Edge TPU use a CISC-style instruction-set architecture and rely on the host program to issue instructions through the system interconnect. Also, TPU tensor units only support operations on a limited level of precision that is sufficient to satisfy the demands of modern ML applications while significantly reducing both TPU costs and energy requirements.



(a) TPU's Matrix Multiply Unit implemented by systolic array



(b) SoM: power draw of components

2.2 Tensorflow Tools and Frameworks

The Edge TPU is based on the well-known TensorFlow framework [43] for the development of the neural networks. **TensorFlow** is a free, open-source platform, that is primarily used to develop, train and deploy Machine Learning models. TensorFlow is capable of operating at high scale and in heterogeneous environments. TensorFlow allows

developers to create dataflow graphs—structures that describe how data moves through a graph, or a series of processing nodes. Each node in the graph represents a mathematical operation, and each connection or edge between nodes is a multidimensional data array, or tensor.

Building a model in TensorFlow is quite simple, you just inherit the Model class from the Keras library. **Keras** [44] is an open source library, built on top of TensorFlow. Its main purpose is to act as an interface for TensorFlow by providing a consistent and user-friendly API. This API offers access to numerous implementations of commonly used neural-network building blocks such as layers, objectives, activation functions, optimizers, as well as a host of tools to make working with image and text data easier and to simplify the coding necessary for writing deep neural network code.

When it comes to saving a trained model, TensorFlow is capable of keeping checkpoints of models during the training procedure or save models in a **SavedModel** format. Checkpoints capture the exact value of all parameters used by a model, allowing users to stop the training and resume it later on, on the same or on a different platform. Checkpoints do not contain any description of the computation defined by the model and thus are typically only useful when source code that will use the saved parameter values is available. The SavedModel format, on the other hand, contains both data describing the structure of the network as well as the weights of the different layers, ie. the parameter values (checkpoint). Therefore, trained models can be loaded on any platform that supports TensorFlow and be used for inference.

TensorFlow operations are run eagerly; they are executed by Python, operation by operation, and return results back to Python. While eager execution has several unique advantages, it comes at the expense of performance and deployability. Graph execution [45], on the other hand, enables portability outside Python and tends to offer better performance. In short, graphs and functions are extremely useful and allow TensorFlow run fast, run in parallel, and run efficiently on multiple devices. Graphs are data structures that contain a set of tf.Operation objects, which represent units of computation; and tf.Tensor objects, which represent the units of data that flow between operations. You create and run a graph in TensorFlow by using tf.function, either as a direct call or as a decorator. tf.function is a transformation tool that takes a regular function as input and returns a Function. A Function is a Python callable that builds TensorFlow graphs from the Python function. Each time a Function is invoked with a set of arguments that can't be handled by any of its existing graphs, Function creates a new tf.Graph specialized to those new arguments. If you call the TF Function with an input signature it has already seen before, it will reuse the concrete function it generated earlier. The type specification of a tf.Graph's inputs is known as its input signature. The Function stores the tf.Graph corresponding to that signature in a **ConcreteFunction**. A ConcreteFunction is a wrapper around a tf.Graph.

2.2.1 TensorFlow Lite Framework

All inferencing with the Edge TPU is based on the TensorFlow Lite APIs (Python or C/C++). **TensorFlow Lite** [46] is a production ready, cross-platform framework that enables on-device machine learning by helping developers run their models on embedded and edge devices. TensorFlow Lite can convert a pre-trained model in TensorFlow to a TFLite format that can be optimized for speed or storage and later be deployed on mobile devices & embedded systems to make the inference at the Edge.

A TensorFlow Lite model is represented in a special efficient portable format known as FlatBuffers. This provides several advantages over TensorFlow's protocol buffer model format such as reduced size (small code footprint) and faster inference (data is directly accessed without an extra parsing/unpacking step) that enables TensorFlow Lite to execute efficiently on devices with limited compute and memory resources.

A TensorFlow Lite model can optionally include metadata that has human-readable model description and machine-readable data for automatic generation of pre- and post-processing pipelines during on-device inference. Generating a TensorFlow Lite model can occur in the following ways:

- Use an existing TensorFlow Lite model.
- Create a TensorFlow Lite model, using the TensorFlow Lite Model Maker to create a model with custom dataset.
- Convert a TensorFlow model into a TensorFlow Lite model.

TensorFlow Lite Converter

The TensorFlow Lite converter takes a TensorFlow model and generates a TensorFlow Lite model (an optimized FlatBuffer format identified by the .tflite file extension). During conversion, optimizations such as quantization can be applied to reduce model size and latency with minimal or no loss in accuracy. In TensorFlow 2, models are stored using the SavedModel format and are generated either using the high-level tf.keras.* APIs (Keras model) or the low-level tf.* APIs (from which you generate concrete functions). As a result, there are three options to convert a TensorFlow 2 model using tf.lite.TFLiteConverter, as follows:

```
tf.lite.TFLiteConverter.from_saved_model(): Converts a
    SavedModel.
tf.lite.TFLiteConverter.from_keras_model(): Converts a Keras
    model.
tf.lite.TFLiteConverter.from_concrete_functions(): Converts
    concrete functions.
```

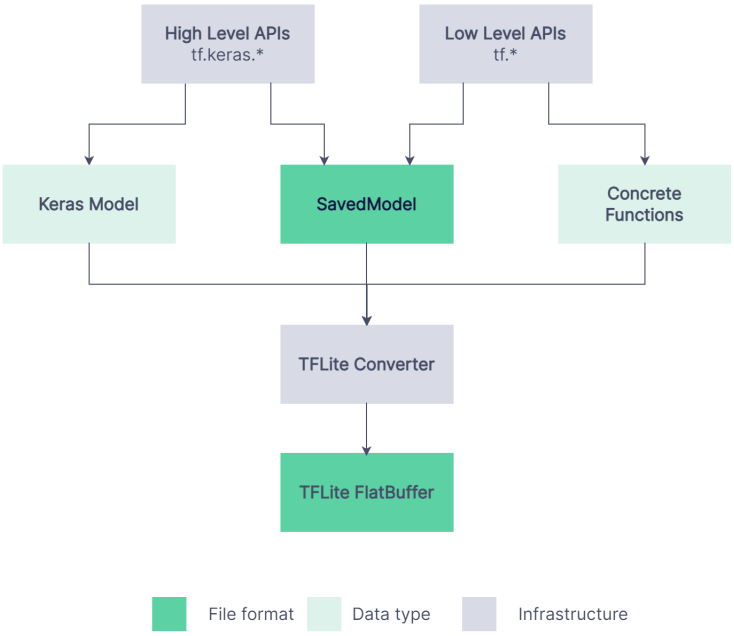


Figure 2.4: TFLite conversion workflow

2.2.2 TensorFlow Model Compatibility Overview

In order for the Edge TPU to provide high-speed neural network performance with a low-power cost, the Edge TPU supports a specific set of neural network operations and architectures. We will describe what types of models are compatible with the Edge TPU and how to create them, either by compiling custom TensorFlow models or retraining an existing model with transfer-learning.

The Edge TPU supports only TensorFlow Lite models that are fully 8-bit quantized and then compiled specifically for the Edge TPU. Models cannot be trained directly with TensorFlow Lite; instead must be converted from a TensorFlow file (such as a .pb file) to a TensorFlow Lite file (a .tflite file), using the TensorFlow Lite converter.

Figure 2.5 illustrates the basic process to create a model that's compatible with the Edge TPU. Most of the workflow uses standard TensorFlow tools. Once we have a TensorFlow Lite model, then we use our Edge TPU compiler to create a .tflite file that's compatible with the Edge TPU.

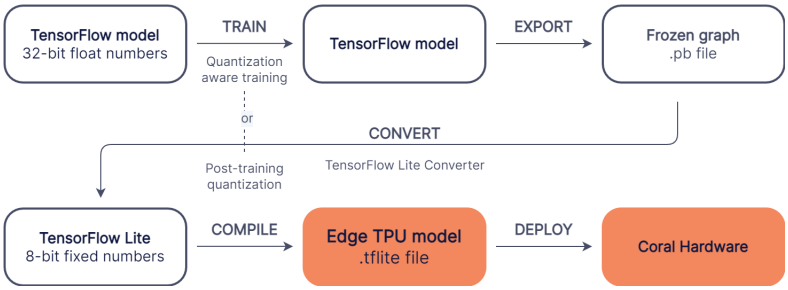


Figure 2.5: Edge TPU model creation workflow

However, it is not always necessary to create a model from scratch. Instead, there are existing TensorFlow models that are compatible with the Edge TPU that can be retrained with any dataset, using a technique called transfer learning (or "fine tuning").

Transfer learning allow developers to start with a model that's already trained for a related task and then perform further training to teach the model new classifications using a smaller training dataset. This can be achieved by removing the final layer that performs classification and training a new layer on top that recognize your new classes. Also, the Python and C++ APIs offer techniques for on-device accelerated transfer learning on image classification models.

2.2.3 Model Requirements

TensorFlow models build in order to take full advantage of the Edge TPU for accelerated inferencing, must meet the following basic requirements:

- Tensor parameters are **quantized** (8-bit fixed-point numbers; int8 or uint8).
- Tensor sizes are constant at compile-time, **no dynamic sizes**.
- Model parameters are constant at compile-time.
- Tensors are either 1-, 2-, or 3-dimensional. If a tensor has more than 3 dimensions, then only the 3 innermost dimensions may have a size greater than 1.
- The model uses only the operations supported by the Edge TPU (see 2.2.2.2 below).

Failure to meet these requirements could mean the model cannot compile for the Edge TPU at all, or only a portion of it will be accelerated. Also, passing a model to the Edge TPU Compiler that uses float inputs means that the compiler will leave a quantize op at the beginning of the graph, which will run on the ARM CPU and result in additional latency due to the data format conversion. Likewise, the output is dequantized at the end. However, as long as the tensor parameters are quantized, it's okay if the input and output tensors are float because they'll be converted on the CPU.

2.2.3.1 Supported Operations

When building any model architecture, it is necessary to know that only the operations in the linked table are supported by the Edge TPU. **Supported Operations**

Operations that are not listed in that table, usually work either an unsupported data type or the tensors have unsupported rank or cannot be supported due to their dynamic size and format. In this case, only a portion of the model will execute on the Edge TPU.

2.2.4 Quantization

Executing a model compatible with the Edge TPU, requires that all the 32-bit floating-point numbers (such as weights and activation outputs) are converted to the nearest 8-bit fixed-point numbers. This makes the model smaller and faster. And although these 8-bit representations can be less precise, the inference accuracy of the neural network is not significantly affected. For compatibility with the Edge TPU, there are two types of quantization:

1. Quantization-aware training (for TensorFlow 1) uses "fake" quantization nodes in the neural network graph to simulate the effect of 8-bit values during training. Thus, this technique requires modification to the network before initial training. This generally results in a higher accuracy model (compared to post-training quantization) because it makes the model more tolerant of lower precision values, due to the fact that the 8-bit weights are learned through training rather than being converted later. Also, it is currently compatible with more operations than post-training quantization.

2. Post-training quantization [47] is a conversion technique that can reduce model size while also improving CPU and hardware accelerator latency, with little degradation in model accuracy. We can quantize an already-trained float TensorFlow model when we convert it to TensorFlow Lite format using the TensorFlow Lite Converter. There are several post-training quantization options to choose from. Here is a summary table of the choices and the benefits they provide:

Table 2.2: Post-training quantization options

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU

Full integer quantization [48] does not require any modifications to the network, so this technique can be used to convert a previously-trained network into a quantized model and get further latency improvements, reductions in peak memory usage, and compatibility with integer only hardware devices or accelerators.

However, for full integer quantization, we need to calibrate or estimate the range of all floating-point tensors in the model. This conversion process requires supplying a **representative dataset**; a dataset that is formatted the same as the original training dataset, using the same data range, and is of a similar style. This representative dataset allows the quantization process to measure the dynamic range of activations and inputs, which is critical to finding an accurate 8-bit representation of each weight and activation value. This dataset can be a small subset (around 100-500 samples) of the training or validation data. So, the 8-bit quantization approximates floating point values using the following equation:

$$r = (q - Z) \cdot S \tag{2.1}$$

Equation 2.1 is our quantization formula and the constants Z and S are our quantization parameters. For 8-bit quantization, q is quantized as an 8-bit integer. The constant S (for "scale") is an arbitrary positive real number, typically represented in software as a floating point quantity, like the real values r. The constant Z (for "zero-point") is of the same type as quantized values q, and is in fact the quantized value q corresponding to the real value 0. This allows us to automatically meet the requirement that the real value r = 0 be exactly represented by a quantized value. Quantization ranges are treated differently for weight quantization and activation quantization:

Per-axis or per-tensor weights are represented by int8 two's complement values in the range [-127, 127] with zero-point equal to 0. Per-tensor activations/inputs are represented by int8 two's complement values in the range [-128, 127], with a zero-point in range [-128, 127].

2.2.5 Edge TPU Compiler

The Edge TPU Compiler (edgetpu compiler) [49] is a command line tool that compiles a TensorFlow Lite model (.tflite file) into a file that's compatible with the Edge TPU. The compiler accepts the file path to one or more TensorFlow Lite models, known as the model argument. If we pass multiple models, they are co-compiled such that they can share the Edge TPU's RAM for parameter data caching.

After the TensorFlow model is trained and converted your to TensorFlow Lite with quantization, the final step is to compile it with the Edge TPU Compiler. Unless the model meets all the requirements listed at section 2.2.2.1, it can still compile, but only a portion of the model will execute on the Edge TPU. In the case where an unsupported operation occurs, the compiler partitions the graph into two parts. The first part of the graph that contains only supported operations is compiled into a custom operation that executes on the Edge TPU, and everything else executes on the ARM processor, as illustrated in Figure 2.6.

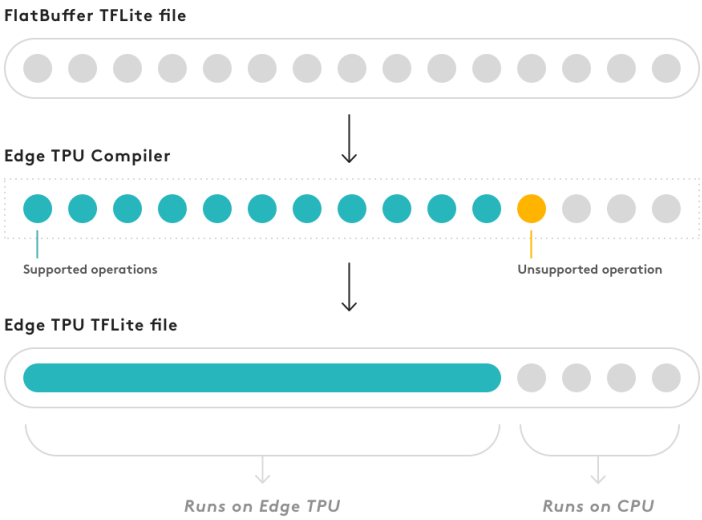


Figure 2.6: Edge TPU compiler supported and unsupported ops handling

By inspecting a compiled model, with a tool such as netron.app, we can see that it's still a TensorFlow Lite model except it now has a custom operation at the beginning of the graph. This custom operation is the only part of the model that is actually compiled and contains all the operations that run on the Edge TPU. The rest of the graph, that contains the unsupported operations, remains the same and runs on the CPU.

If part of a model executes on the CPU, we should expect a significantly degraded inference speed compared to a model that executes entirely on the Edge TPU. We cannot predict how much slower the model will perform in this situation. However, the percentage of operations that execute on the Edge TPU versus the CPU does not correspond to the overall performance impact; if even a small fraction of the model executes on the CPU, it can potentially slow the inference speed by an order of magnitude compared to a version of the model that runs entirely on the Edge TPU.

2.2.5.1 Data Caching

The Edge TPU has roughly 8 MB of SRAM that can cache the model's parameter data. However, a small amount of the RAM is first reserved for the model's inference executable, so the parameter data uses whatever space remains after that. Naturally, saving the parameter data on the Edge TPU RAM enables faster inferencing speed compared to fetching the parameter data from external memory. This Edge TPU "cache" is not actually traditional cache—it's compiler-allocated scratchpad memory. The Edge TPU Compiler adds a small executable inside the model that writes a specific amount of the model's parameter data to the Edge TPU RAM, if available, before running an inference.

When compiling models individually, the compiler gives each model a unique "caching token" (a 64-bit number). Then, at the execution stage, the Edge TPU runtime compares that caching token to the token of the data that is currently cached. If the tokens match, the runtime uses that cached data. If they don't match, it wipes the cache and writes

the new model's data instead. When models are compiled individually, only one model at a time can cache its data. When the system clears the cache and writes new model's data to cache, it delays the inference. So the first time a model runs is always slower. Any later inferences are faster because they use the cache that's already written. But if an application constantly switches between multiple models, this cache swapping adds significant overhead to the application's overall performance. The solution to this matter comes from co-compilation.

2.2.5.2 Co-compilation

To speed up performance when continuously running multiple models on the same Edge TPU, the compiler supports co-compilation. Essentially, co-compiling allows multiple models to share the Edge TPU RAM to cache their parameter data together, eliminating the need to clear the cache each time we run a different model.

When we pass multiple models to the compiler, each compiled model is assigned the same caching token. So when we run any co-compiled model for the first time, it can write its data to the cache without clearing it first.

The amount of RAM allocated to each model is fixed at compile-time, and it is prioritized based on the order the models appear in the compiler command. The cache space is first allocated to the first model's data, as much as can fit. If space remains after that, cache is given to second model's data. If some of the model data cannot fit into the Edge TPU RAM, then it must instead be fetched from the external memory at run time. If we co-compile several models, it is possible some models don't get any cache, so they must load all data from external memory. That is slower than using the cache, but when running the models in quick succession, this could still be faster than swapping the cache every time you run a different model.

2.2.5.3 Performance

The Edge TPU Compiler assigns a fixed amount of cache space for every model's parameter data at compile-time. If we co-compile two models that use less than the available on-chip caching memory, then the external memory usage will be zero. However, if there is not enough space for both models' parameter data, then the rest must be fetched from the external memory, as we see in Figure 2.7. So to achieve maximum performance gains it is vital to never read from external memory and never rewrite the Edge TPU RAM.

Figure 2.7: Edge TPU RAM parameter data fitting

When deciding whether to use co-compilation, you should run the compiler with all your models to see whether they can fit all parameter data into the Edge TPU RAM, by reading the compiler output. If they cannot all fit, then the most-frequently-used model should be passed to the compiler first, so it can cache all its parameter data. On the other hand, if they cannot all fit and they switch rarely, then perhaps co-compilation is not beneficial because the time spent reading from external memory is more costly than periodically rewriting the Edge TPU RAM.

2.2.5.4 Segmentation

One other available option offered by the edgetpu compiler is model segmentation. If a model is unable to fit into the Edge TPU cache or the overall throughput for the model is a bottleneck in the program, we can improve your performance by segmenting your model into separate subgraphs that run in a pipeline on separate Edge TPUs. During segmentation, the compiler outputs each segment as a separate .tflite file with an enumerated filename, so the model can be executed in a pipeline using multiple Edge TPUs.

However, we should consider what number of segments will achieve our performance goals, based on the size of the model and whether we are trying to fit a large model entirely into the Edge TPU cache, to reduce latency, or we are trying to increase your model's throughput.

2.3 Methodology for Edge TPU Assessment

The evaluation of the Edge TPU as an AI accelerator, the comparison to other embedded devices, as well as the trade-off analysis are performed according to NTUA's assessment methodology. The proposed methodology we followed in this work included:

1. Study the literature to create an initial pool of AI/ML benchmarks and datasets and a pool of bibliography results from the competitive embedded devices.

2. Perform initial TPU benchmarking and experimentation with the TPU with "black box" testing to derive first results and experience by assuming new users on purpose. We will also perform some preliminary comparisons and define the details for the next steps of the methodology.

3. Apply extended benchmarking and testing on Edge TPU, comparisons and trade-offs analysis with custom designs and experiments. The block diagram of the methodology is presented in Figure 2.8.

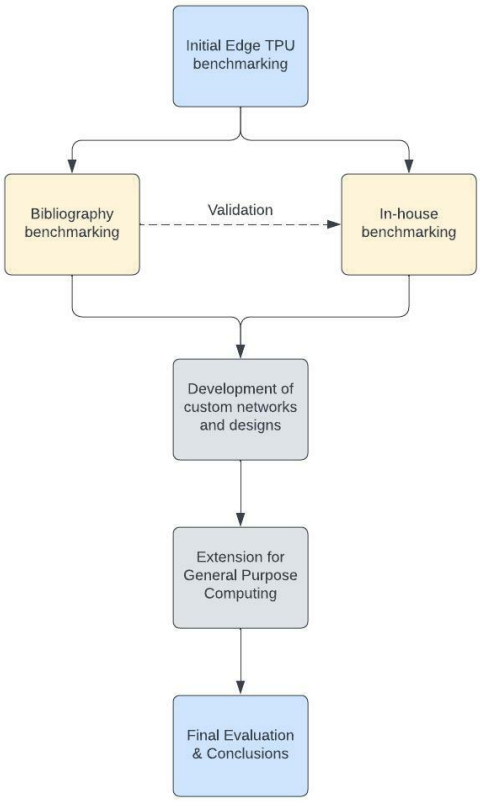


Figure 2.8: Methodology for the assessment of the Edge TPU

2.4 Initial Benchmarking

We performed an in-depth benchmarking involving several types of neural networks and applications, as well as varying network complexity. Our benchmarking assessment involves results collected from the bibliography and in-house inference runs. Regarding the embedded devices, we employ the following:

TPU: 2 Coral TPU dev boards (regular and mini)
GPU: NVIDIA's Jetson Nano (Cortex-A57 + 128-core Maxwell GPU) [6]
VPU: Intel's NCS2 (host machine with i7 CPU + Myriad X VPU) [7]
FPGA: Xilinx's Zynq FPGAs (Zynq-7020, ZCU104) [8]
CPU: ARM CPUs (embedded in the aforementioned boards, with frequency around 1.3-1.5 GHz)

2.4.1 Bibliography

Table 2.3 below compares the time spent to perform a single inference on the Edge TPU for several popular models with varying applications, such as image classification, object detection, pose estimation and image segmentation. All models running on both CPU and Edge TPU are the TensorFlow Lite versions of pre-trained models offered by Coral, representing a small selection of model architectures that are compatible with the Edge TPU. Om the other hand, table 2.4 compares the average throughput between the Edge TPU and other embedded devices mentioned before.

Metrics

Latency is the amount of time it takes to perform one inference. It is measured in units of time using the time() function of python. The timer counts the time it takes the TensorFlow Lite to invoke the interpreter. During invoking, heavy computation is done in the background and no other function on this object can be called while the invoke() call has not finished. Latency is measured in milliseconds (ms).

Throughput is the rate at which a system can process inputs. It is an amount of measurements per a given time. It is not a measure of how recent they are, rather just a measurement of their volume. In our case, we consider that throughput is inversely proportional to latency. This is due to the fact that the interpreter is invoked with one input at a time. Throughput is measured in frames per second (FPS).

$$throughput = \frac{1}{latency} \tag{2.2}$$

On the right hand side of Table 2.3 we can see the acceleration factors between the Edge TPU and the ARM A-53 CPU on varying neural network architectures.

Table 2.3: Bibliography benchmarking results (latency) [1]

Neural Network	**Input size**	**ARM A-53**[1] **(ms)**	**TPU DevB**[2] **(ms)**	**TPU USB**[3] **(ms)**	**Accel factor**
Unet Mv2	128x128	190.7	5.7	3.3	33
DeepLab V3	513x513	1139	241	52	5
DenseNet	224x224	1032	25	20	41
Inception V1	224x224	392	4.1	3.4	96
Inception V2	224x224		20.8	13.4	
Inception V3	299x299		59	42.8	
Inception V4	299x299	3157	102	85	31
Inception-Resnet V2	299x299	2852	69	57	41
MobileNet V1	224x224	164	2.4	2.4	68
MobileNet V2	224x224	122	2.6	2.6	47
SSD MobileNet V1	224x224	353	11	6.5	32
SSD MobileNet V2	300x300	282	14	7.6	20
ResNet-50 V1	224x224	1763	56	49	31
ResNet-50 V2	299x299	1875	59	50	32
ResNet-152 V2	299x299	5499	151	128	36
SqueezeNet	224x224	232	2	2	116
Vgg16	224x224	4595	343	296	13
Vgg19	224x224	5538	357	308	16
EfficientNet-S	224x224	705	5.5	5	128
EfficientNet-M	240x240	1081	10.6	9	102
EfficientNet-L	300x300	2717	30.5	25	89

[1] Embedded CPU: Quad-core ARM Cortex-A53 @ 1.5GHz

[2] Dev Board: Quad-core ARM Cortex-A53 @ 1.5GHz + Edge TPU @ 500MHz @ 2 Watts

[3] Desktop CPU + USB TPU: Single 64-bit Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz + Edge TPU @ 500MHz

Table 2.4: Bibliography benchmarking results (throughput) [1–4]

Neural Network	**TPU USB[1] (FPS)**	**ARM A-53[2] (FPS)**	**TPU DevB[3] (FPS)**	**Nvidia Jetson Nano[4] (FPS)**	**Intel NCS2[5] + i7 (FPS)**	**Zynq ZCU104[6] (FPS)**
Inception V1	294	2.55	244	76	93	202
Inception V4	12	0.32	10	11	10	30.5
MobileNet V1	417	6.1	417	80	119	344
MobileNet V2	385	8.2	385	60	75	284
SSD MobileNet V2	132	3.6	71	39	58	86
ResNet-50 V1	20	0.57	18	21	29	93
SqueezeNet	476	24.3	500	104	287	305
Vgg16	3	0.22	3	12		21.5
Vgg19	3	0.18	3	10		18.5
Tiny Yolo V3				25	46	123

[1] CPU + USB TPU: Single 64-bit Intel Xeon Gold 6154 @ 3.00GHz + Edge TPU @ 500MHz [1]

[2] Embedded CPU: Quad-core Cortex-A53 @ 1.5GHz [1]

[3] Dev Board: Quad-core Cortex-A53 @ 1.5GHz + Edge TPU @ 500 MHz @ 2 Watts [1]

[4] Nvidia GPU: Quad-core ARM Cortex-A57 @ 1.5GHz @ 10 Watts, 4 GB LPDDR4 [2]

[5] Intel VPU: Intel Neural Compute Stick 2 + Movidius Myriad X VPU @ 700MHz [3]

[6] FPGA: Quad-core Cortex-A53 @ 1.5GHz + Dual-core Cortex-R5 + mali-400 GPU + 2 GB DDR4 + 504k LUTs + 1728 DSP [4]

Table 2.5: Acceleration factor between embedded devices and Edge TPU

Neural Network	**Model size (MB)**	**TPU DevB/ Jetson Nano**	**TPU DevB/ Intel NCS2**	**TPU DevB/ Zynq ZCU104**
MobileNet V2	4.1	6.4	5.1	1.4
MobileNet V1	4.5	5.2	3.5	1.2
SqueezeNet	5	4.8	1.7	1.6
SSD MobileNet V2	6.7	1.8	1.2	0.83
Inception V1	7	3.2	2.6	1.2
ResNet-50 V1	25	0.86	0.6	0.2
Inception V4	43	0.91	1	0.33
Vgg16	138	0.25		0.14
Vgg19	143	0.33		0.16

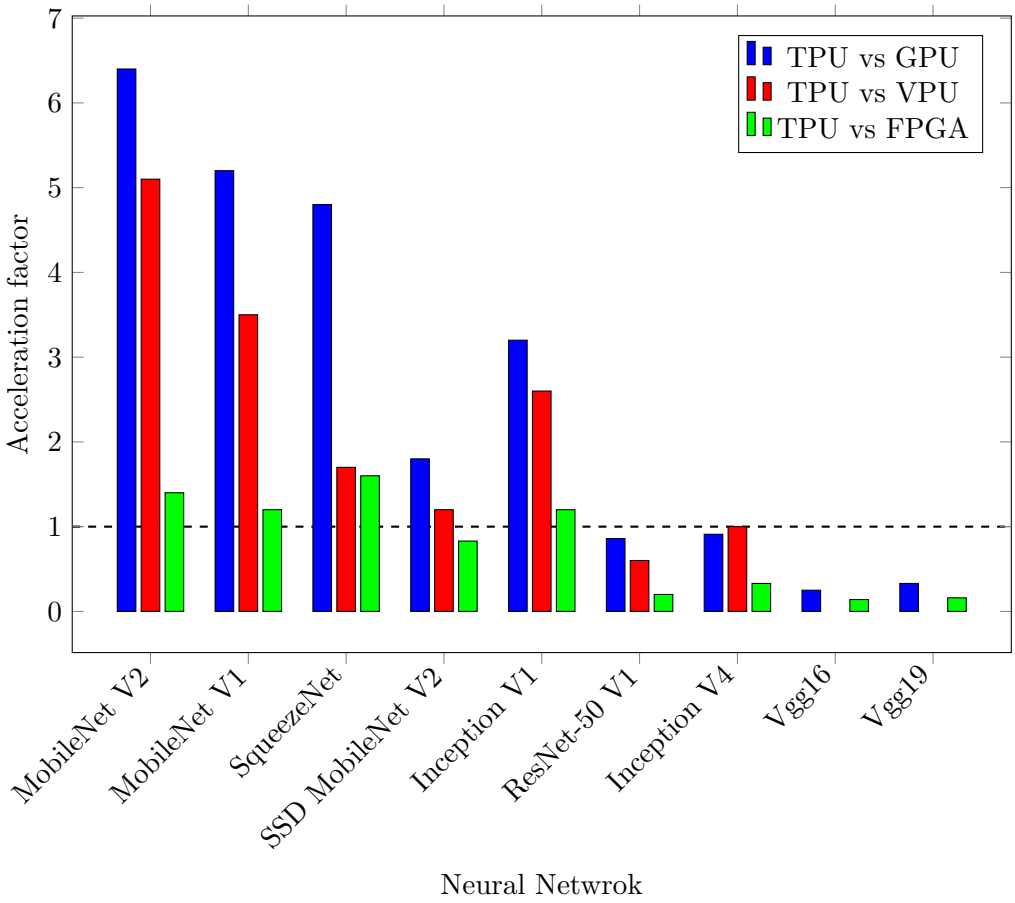


Figure 2.9: Acceleration of Edge TPU over other embedded devices

Figure 2.9 visualizes the results from the initial benchmarking we performed on bibliography neural networks and applications. Essentialy, it depicts the acceleration factors between the embedded deviced we benchmarked. We have sorted the table by model size in order to evaluate its effect on performance among embedded devices.

For low- and mid-sized CNNs, the Edge TPU accelerator achieves 3-7x lower inference time than the Nvidia GPU, for half the power consumption. In comparison to the Intel Myriad X VPU, the acceleration factors are ranging between 2-5x for the same low and mid sized CNNs, while for the same networks performs equally to the Zynq ZCU 104.

For bigger size and higher complexity networks, such as ResNet, Inception V4 and Vgg, the Edge TPU either under-performs or achieves equal results to the other embedded devices.

2.4.2 In-house Testing

Table 2.6 reports the results that were obtained from the in-house benchmarking we performed. The aim of that procedure was to verify the validity of the results presented by bibliography, get hands-on experience with both development boards and build our own methodology for upcoming benchmarking with custom network architectures. During in-

house testing we also had the opportunity to compare the 'large' development board with the 'mini', for both the accelerators and the ARM processors.

Table 2.6: In-house benchmarking Results (latency)

Neural Network	**Input size**	**Model size (MB)**	**ARM A-53 (ms)**	**TPU DevB (ms)**	**ARM A-35 (ms)**	**TPU DevB Mini (ms)**
Unet Mv2	128x128	7.2	192	26	389	275
DeepLab V3	513x513	2.9	1107	206	2170	472
MobileNet V1	224x224	4.5	167	3.6	350	17.7
MobileNet V2	224x224	4.1	134	3.1	261	14.3
Inception V1	224x224	7.0	377	5.8	813	34.3
Inception V2	224x224	12.0	601	18	1214	221
Inception V3	299x299	23.9	1433	54	3045	721
Inception V4	299x299	42.9	2951	103	6200	1400
EfficientNet-S	224x224	6.8	691	6.2	1424	21
EfficientNet-M	240x240	8.7	1069	9.9	2212	57
EfficientNet-L	300x300	12.8	2696	27	5641	233
PoseNet ResNet-50	288x416	24.4	3747	93	7764	929
PoseNet ResNet-50	480x640	26.4	9244	387	19144	6009
SSD MobileNet V1	224x224	7.0	352	12	710	53
SSD MobileNet V2	300x300	6.7	284	15	570	58
SSD MobileNet V2	320x320	6.7	286	7	568	33
SSD Lite MobileDet	320x320	5.1	492	16	1000	62
EfficientDet Lite 0	320x320	5.7	517	141	958	490
EfficientDet Lite 1	384x384	7.6	988	206	1859	650
EfficientDet Lite 2	448x448	10.2	1565	297	2960	1142
EfficientDet Lite 3	512x512	14.4	2920	282	5673	1168
EfficientDet Lite 3x	640x640	20.6	5476	716	10618	*crash*
Tiny Yolo V3	416x416	9.0	737	24	1490	300

Table 2.7: Acceleration factor between Edge TPU devices and ARM CPUs

Neural Network	**Model size (MB)**	**TPU DevB/ ARM A-53**	**TPU DevB/ TPU Mini**	**ARM A-53/ ARM A-35**
Unet Mv2	7.2	7	11	2
DeepLab V3	2.9	5	2	2
MobileNet V1	4.5	46	5	2.1
MobileNet V2	4.1	32	5	1.9
Inception V1	7.0	65	6	2.2
Inception V2	12.0	33	12	2
Inception V3	23.9	27	13	2.1
Inception V4	42.9	29	14	2.1
EfficientNet-S	6.8	112	3	2.1
EfficientNet-M	8.7	108	6	2.1
EfficientNet-L	12.8	100	9	2.1
PoseNet ResNet-50	24.4	40	10	2.1
PoseNet ResNet-50	26.4	24	16	2.1
SSD MobileNet V1	7.0	29	4	2
SSD MobileNet V2	6.7	19	4	2
SSD MobileNet V2	6.7	39	5	2
SSD Lite MobileDet	5.1	30	4	2
EfficientDet Lite 0	5.7	4	4	1.9
EfficientDet Lite 1	7.6	5	3	1.9
EfficientDet Lite 2	10.2	5	4	1.9
EfficientDet Lite 3	14.4	10	4	1.9
EfficientDet Lite 3x	20.6	8		1.9
Tiny Yolo V3	9.0	31	13	2

Table 2.7 reports the acceleration factors between the Edge TPUs and the ARM CPUs. At first, we notice that the in-house testing comparative results between the Edge TPU and the ARM A-53 processor are similar or closely related to those obtained by bibliography. Furthermore, we can draw conclusions about the 'mini' development board. Although, both devices accommodate the same TPU chip, the TPU Mini seems to be 5-20x slower than the large TPU. This suggests that performance is hugely affected by selected ARM processor and memory architecture of the development board. Taking into account that the ARM A-53 CPU is twice as fast as the ARM A-35, we can assume that the LPDDR3 RAM on the TPU Mini contributes to the added latency.

Chapter 3

Development Methodology and Inference on TPU

3.1 Methodology for TPU Inferencing

In this chapter we will discuss the stages of building an Edge TPU compatible model and running an inference on the Edge TPU. These are two different processes executed on different machines. At first, we will follow the basic process presented in Figure 2.5 to create our TensorFlow Lite model on our host machine. The next step will involves using the TensorFlow Lite API to execute the model on the Edge TPU development board.

3.1.1 Build Edge TPU Compatible Model

Building a TensorFlow model compatible with the Edge TPU accelerator available in Coral devices, requires only two additional steps beyond normal procedures:

- Conversion to TensorFlow Lite using full integer post-training quantization
- Compiling with the Edge TPU Compiler.

.

The following code snippet presents the conversion to TensorFlow Lite model by defining the appropriate representative dataset in order to calibrate the range of all floating-point tensors in the model. This dataset is formatted the same as the original training dataset, using the same data range, and is of a similar style. It usually contains a small subset (around 100-500 samples) of the training data. Additionally, to ensure compatibility with the Coral Edge TPU, you we enforce full integer quantization for all ops including the inputs and outputs. The TensorFlow Lite Converter allows both int8/uint8 and float32 type inputs and outputs; however, full integer quantization results in further latency improvements and reductions in peak memory usage.

Listing 3.1: TFLite Conversion using full integer quantization for all ops and I/Os

```
def representative_data_gen():
  dataset_list = tf.data.Dataset.list_files(flowers_dir + '/*/*'
      )
  for i in range(100):
    image = next(iter(dataset_list))
    image = tf.io.read_file(image)
    image = tf.io.decode_jpeg(image, channels=3)
    image = tf.image.resize(image, [IMAGE_SIZE, IMAGE_SIZE])
    image = tf.cast(image / 255., tf.float32)
    image = tf.expand_dims(image, 0)
    yield [image]

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.
   TFLITE_BUILTINS_INT8]
converter.target_spec.supported_types = [tf.int8]
converter.inference_input_type = tf.uint8  # or tf.int8
converter.inference_output_type = tf.uint8  # or tf.uint8
tflite_model = converter.convert()
```

3.1.2 Run Inference on TPU

Running an inference on the Edge TPU accelerator [9] is based on the **TensorFlow Lite Interpreter API**. The TensorFlow Lite interpreter is designed to be lean and fast. The interpreter uses a static graph ordering and a custom, less-dynamic memory allocator to ensure minimal load, initialization, and execution latency. However, to make development even easier, Coral created a wrapper library, the **PyCoral API**, to handle a lot of boilerplate code that's required when running an inference with TensorFlow Lite. The PyCoral API is built atop the TensorFlow Lite Python API to simplify the code when running an inference on the Edge TPU, and to provide advanced features for the Edge TPU such as model pipelining across multiple Edge TPUs, and on-device transfer learning.

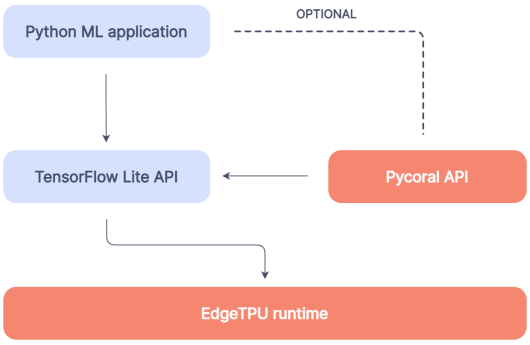


Figure 3.1: Inferencing options

Edge TPU inferencing with Pycoral API and TensorFlow Lite Interpreter includes the following steps, as shown on Figure 3.2. These steps are constant whichever model we want to execute on the TPU accelerator. Only differentiation would occur depending on the input data.

1. Load the .tflite model, which contains the model's execution graph, into memory

2. Transform input data. Raw input data for the model does not match the input data format expected by the model. Usually, it is necessary to resize an image or change the image format to be compatible with the model.

3. Run inference, using the TensorFlow Lite API to execute the model. It involves a few steps such as:

    - Build an Interpreter based on the existing model.
    - Allocate tensors.
    - Set input tensor values based on quantized model's format.
    - Invoke inference.
    - Read output tensor values.

4. Interpret output. When we receive results from the model inference, we need to interpret the tensors in a meaningful way that's useful in our application.

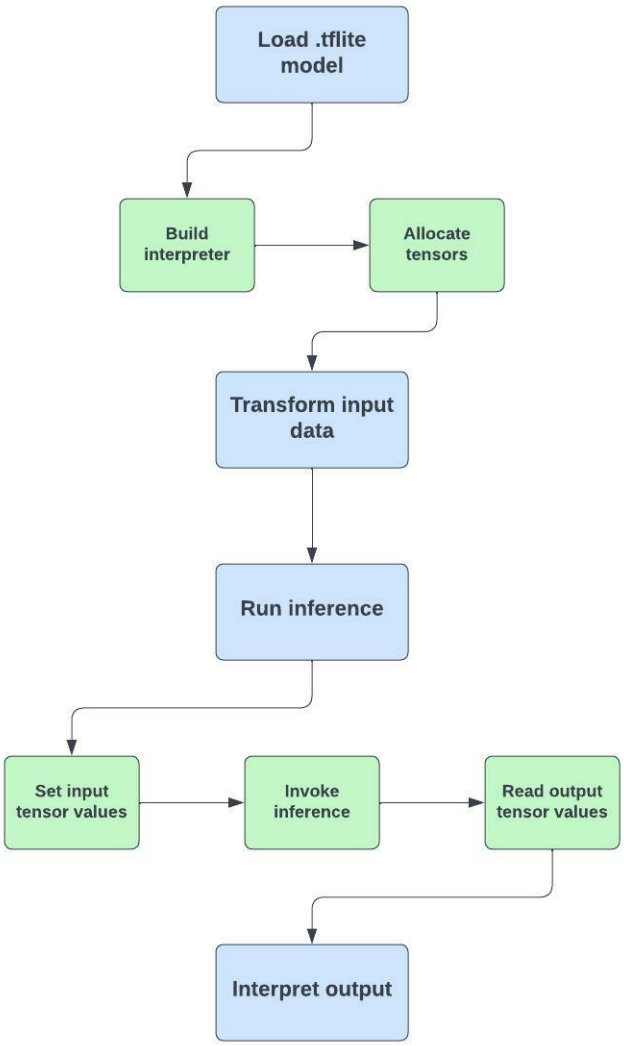


Figure 3.2: Inferencing procedure

The following code snippet presents all the steps mentioned previously about running an inferece on the Edge TPU using the TensorFlow Lite Interpreter and the Pycoral API.

Listing 3.2: Run inference on TPU

```
## Step 1: Build interpreter on .tflite model
interpreter = make_interpreter(model_quant_edgetpu.tflite)
## Step 2: Allocate tensors
interpreter.allocate_tensors()
## Step 3: Transform input data using Pycoral API
size = common.input_size(interpreter)
image = Image.open(input).convert('RGB').resize(size, Image.
    ANTIALIAS)
## Step 4: Set input tensor values
```

```
def set_input_tensor(interpreter, input):
  input_details = interpreter.get_input_details()[0]
  tensor_index = input_details['index']
  input_tensor = interpreter.tensor(tensor_index)()[0]
  # Quantize input data, if necessary:
  scale, zero_point = input_details['quantization']
  input_tensor[:, :] = np.uint8(input / scale + zero_point)

set_input_tensor(interpreter, input)
## Step 5: Invoke inference
interpreter.invoke()
## Step 6: Read output tensor values
def run_inference(interpreter, input):
  output_details = interpreter.get_output_details()[0]
  output = interpreter.get_tensor(output_details['index'])
  # Dequantize output results, if necessary:
  scale, zero_point = output_details['quantization']
  output = scale * (output - zero_point)
  return output

prediction = run_inference(interpreter, input)
## Step 7: Interpret output
out = np.uint64(np.round(prediction,0)) # example
```

3.2 Test Case: Inference of CNN for Ship Detection

3.2.1 CNN Architecture and Dataset

In this section, we will present the implementation of an image detection model on the Edge TPU. We will use a convolutional neural network (CNN) that performs ship detection on satellite imagery in order to evaluate the performance of the TPU accelerator on a real-world application. Also, in the same context, we will present a demo application of ship detection in large scale images of bay areas in California.

The neural network we will use is built in-house and is already presented in previous projects of NTUA's Microlab [10–12]. It is a 6 layers deep CNN, consisting of 4 2-D convolutional + 4 pooling layers and 2 fully connected layers followed by a softmax layer for output, as shown in Figure 3.3.

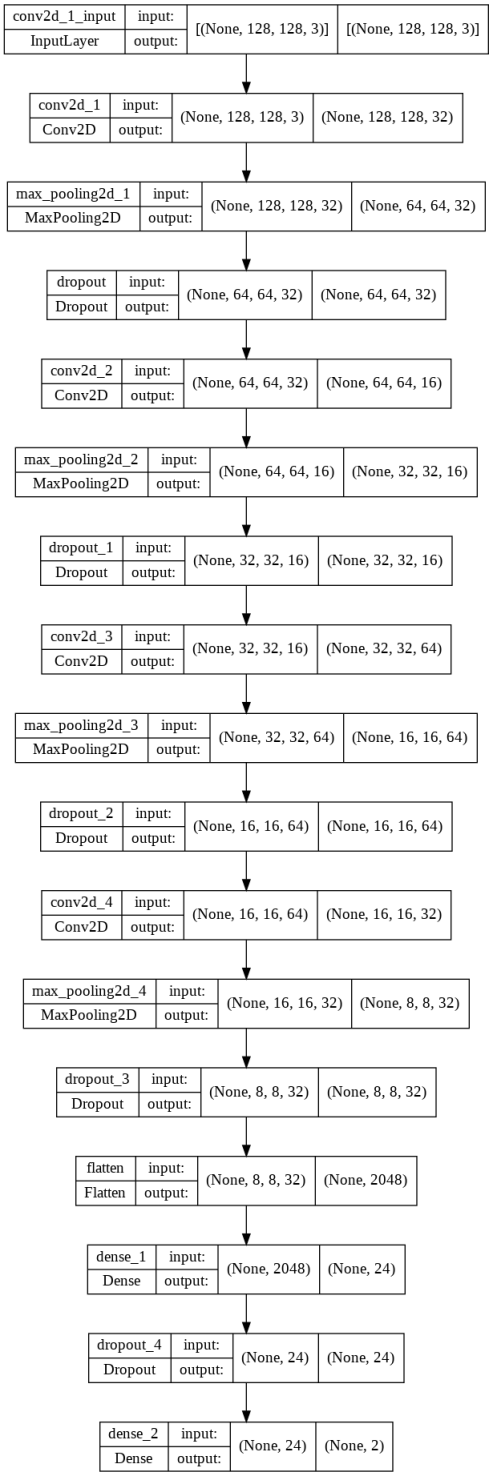
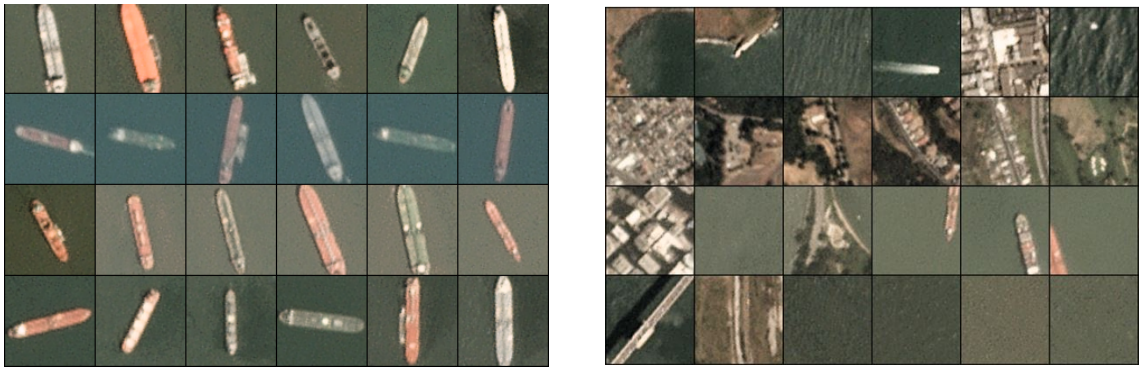


Figure 3.3: Ship Detection CNN Visualization

Our convolutional neural network will be trained on a kaggle-derived dataset [13] that consists of image chips extracted from Planet satellite imagery collected over the San Francisco Bay and San Pedro Bay areas of California. The dataset includes 4000 80x80 RGB images labeled with either a "ship" or "no-ship" classification.

The "ship" class includes 1000 images. Images in this class are near-centered on the body of a single ship. Ships of different sizes, orientations, and atmospheric collection conditions are included. Example images from this class are shown in Figure 3.4a.

The "no-ship" class includes 3000 images. A third of these are a random sampling of different landcover features - water, vegetation, bare earth, buildings, etc. - that do not include any portion of an ship. The next third are "partial ships" that contain only a portion of an ship, but not enough to meet the full definition of the "ship" class. The last third are images that have previously been mislabeled by machine learning models, typically caused by bright pixels or strong linear features. Example images from this class are shown in Figure 3.4b.



(a) "Ship" class

(b) "No-ship" class

Figure 3.4: Dataset classes

3.2.2 Implementation on Edge TPU

First, we begin by pre-processing the image data; we rescale the images into float values so the tensor values are between 0 and 1 and then we split the dataset into training and validation. Then, we continue by creating the base keras model on our host machine and fitting it on the pre-processed training dataset. We save the trained model's weights in .hdf5 format. From now, our model can be loaded straight from the weights file at any moment.

At this point, we need to make our trained model compatible for the Edge TPU. Following the steps we presented as methodology in chapter 3, we will convert our float TensorFlow model into TensorFlow using full integer post-training quantization as mentioned on Listing 3.1. To fully quantize the model with the TFLiteConverter, we define a representative dataset that contains 100 samples of the training image data. The only step left, is to compile the model for the Edge TPU. Figure 3.5a notes that our quantized model has been compiled successfully and is totally mapped on the Edge TPU accelerator, resulting in maximum performance and lowest latency. The log file in Figure 3.5b displays all 14 operations to be executed on the Edge TPU.

```
Edge TPU Compiler version 16.0.384591198
Started a compilation timeout timer of 180 seconds.

Model compiled successfully in 121 ms.

Input model: ship_tf_quant.tflite
Input size: 89.67KiB
Output model: ship_tf_quant_edgetpu.tflite
Output size: 160.62KiB
On-chip memory used for caching model parameters: 199.00KiB
On-chip memory remaining for caching model parameters: 6.71MiB
Off-chip memory used for streaming uncached model parameters: 0.00B
Number of Edge TPU subgraphs: 1
Total number of operations: 14
Operation log: ship_tf_quant_edgetpu.log
See the operation log file for individual operation details.
Compilation child process completed within timeout period.
Compilation succeeded!
```

(a) Compiler output message

```
Edge TPU Compiler version 16.0.384591198
Input: ship_tf_quant.tflite
Output: ship_tf_quant_edgetpu.tflite

Operator                       Count      Status

SOFTMAX                        1          Mapped to Edge TPU
RESHAPE                        1          Mapped to Edge TPU
CONV_2D                        4          Mapped to Edge TPU
FULLY_CONNECTED                2          Mapped to Edge TPU
QUANTIZE                       2          Mapped to Edge TPU
MAX_POOL_2D                    4          Mapped to Edge TPU
```

(b) Operation log file

Figure 3.5: Edge TPU Compiler output

Following the workflow presented in Figure 3.2 and the script in Listing 3.2, we will run inferences of ship detection on the Edge TPU. The model takes 80x80 images as inputs and returns a prediction of their classes, as shown in Figure 3.6 and table 3.1.

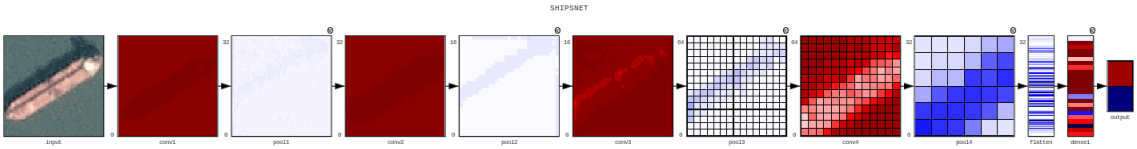
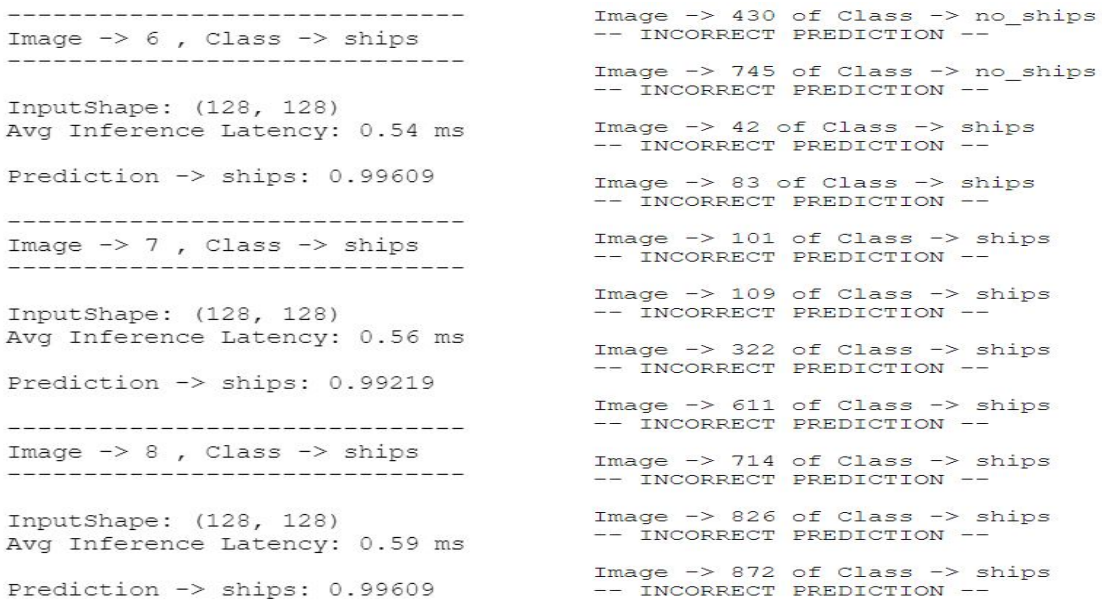


Figure 3.6: Ship Detection Network with Image input

Table 3.1: Image prediction

Label	No ship	Ship
Prediction	0.006	0.994

In order to examine the accuracy of the model, we will inference half of the given dataset on the TPU. In particular, we load 1000 'ships' and 1000 'no-ships' images and expect prediction for each one of them. The output log files we receive after inferencing, are presented in Figure 3.7. On the left side, we print the average inference time and the prediction for each specific input image, while on the right hand side we save the input labels that where predicted incorrectly.



(a) Outputs file

(b) Incorrect predictions log file

Figure 3.7: Ship detection test on 2000 images

The accuracy and the performance of our model, either executed on CPU or on the TPU, quantized or not, are presented on Table 3.2. Related work on the same dataset developed on a Xilinx Virtex 7 XC7VX485T FPGA accelerator has shown similar performance (inference time @ 0.687 ms) for higher power consumption (5 watts) [14], while the same CNN for ship detection has achieved 725 FPS on a Xilinx Zynq Z-7020 FPGA [15] at 4 watts [12].

Table 3.2: Accuracy & Performance

Model	quant type	inference	Accuracy	Latency (ms)	Throughput (FPS)
TFLite	no	CPU	87.8 %	36	28
TFLite	uint8	CPU	99.4 %	22	45.5
TFLite Compiled	uint8	Edge TPU	99.4 %	0.5-1	1000-2000

The directory of the implementation for the complete ship detection model we have discussed so far can be found it the following github repository: Applications on Google Edge TPU.

Ship Detection

Using the same model, we will present a demo application for ship detection. Our target is to identify regions of interest in large scale images of bay areas. The workflow we will follow is presented below in figure 3.8. We have built a custom function that splits an image both horizontally and vertically in selected width and height boxes with pre-defined overlap, so that we do not lose any vital information during this procedure. For example, large images like Figure 3.9, with input resolution of 2844x1828, can be split into **12460** 80x80 images with overlap equal to 60 pixels, like Figure 3.10.

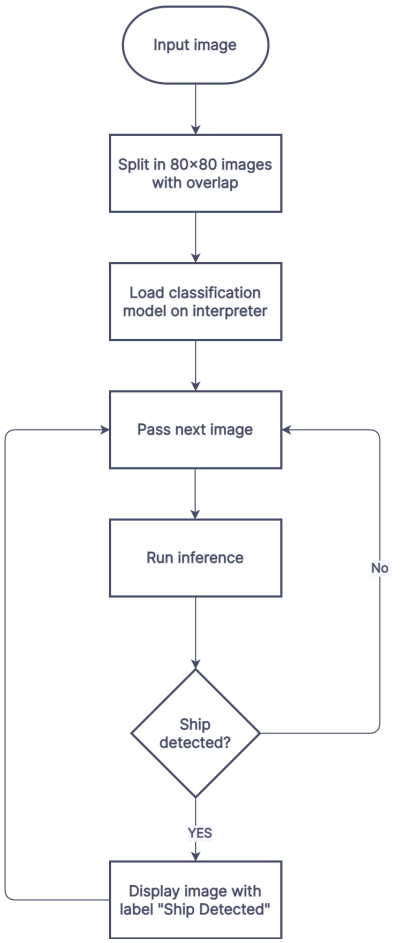


Figure 3.8: Ship Detection Workflow

Figure 3.9: Long Beach bay area (2844x1828)

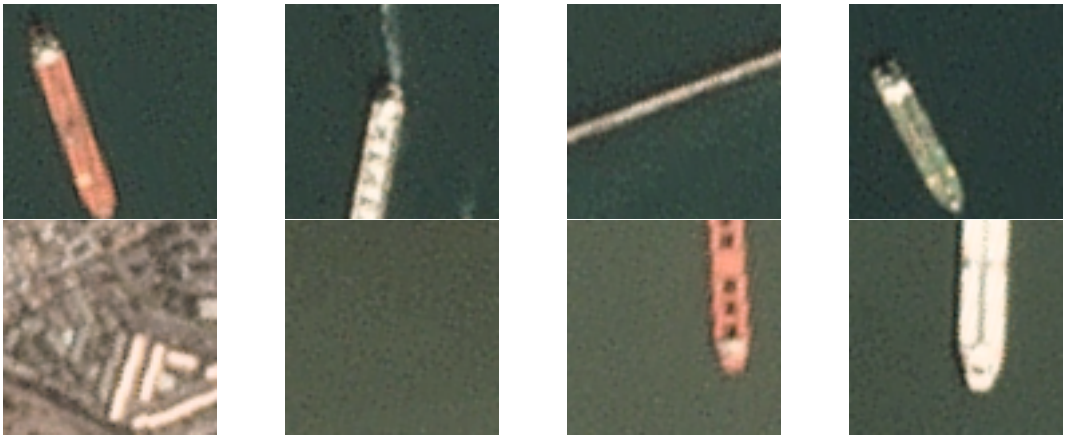


Figure 3.10: 80x80 cropped images

Figure 3.11: Ship Detection Demo output

Our model accelerated with Edge TPU can run inference for any 80x80 input image at 0.5-1 milliseconds (ms). So with throughput of 1000 FPS, it can return accurate ship predictions for 12460 inputs in less than 12.5 seconds.

3.3 Development of Custom General-Purpose Operations

In this section, we will experiment with the Edge TPU developing custom designs. We need to examine whether this neural network accelerator, optimized specifically to handle neural network workloads, can perform low-level operations. So, our target is to understand more about TPU's architecture and also propose some solutions for accelerated general-purpose computing.

We have already discussed that building a TPU compatible model can be done in two ways, either with high-level or with low-level APIs. In the first case, we can build any model in TensorFlow using the Keras library, which offers access to numerous implementations of commonly used neural-network building blocks such as layers, objectives, activation functions and optimizers. On the other hand, we can use tf.function in order to compile any given function or operation into a callable TensorFlow graph. TF Functions are polymorphic, thus supporting inputs of different types ans shapes. Every time we call a TF Function with a new combination of input types or shapes, it generates a new concrete function, with its own graph specialized for this particular combination. Listings 3.3 and 3.4 illustrate the implementation of the same operation with both options we mentioned:

Listing 3.3: Keras model

```
input_shape = (4, 1)

input_1 = tf.keras.layers.Input(shape=(input_shape))
input_2 = tf.keras.layers.Input(shape=(input_shape))
add = tf.keras.layers.Add()([input_1, input_2])
model = tf.keras.models.Model(inputs=[input_1,input_2], outputs
    =[add])
```

Listing 3.4: Concrete function

```
@tf.function
def add(x1, x2):
  return tf.add(x1,x2)

tensor = add.get_concrete_function(tf.ones([4, 1]), tf.ones([4,
    1]))
```

In the following sections, we will explore thoroughly both implementations and will propose any necessary modification in order to perform inference on general-purpose operations.

3.3.1 8-bit Add & Multiply operations

First, we will propose an implementation for general algebraic operations, like element-wise additions and multiplications. As we have already discussed, the TPU accelerator achieves maximum performance for fully quantized uint8 models. Consequently, all input and output tensors are in the range of 0 and 255. This requires that the representative dataset is defined properly so that inputs and outputs are quantized with the correct scale factor.

- **Addition:** adding two 8-bit integers, results in one 9-bit integer. The output result needs to be stored in a 8-bit register. The minimum possible addition is 0+0, which returns 0, while the maximum possible addition is 255+255 and returns 510. So we have 510 possible results, that need to be stored only in 256 positions inside the Edge TPU.

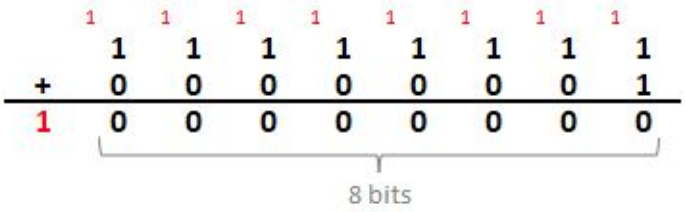


Figure 3.12: Adding 8-bit unsigned integers

In Table 3.3 we can see the mapping of every possible algebraic result, to the 256 memory locations. After de-quantization from the Edge TPU, every value will be scaled by 2, in order to return the expected output, hence the maximum error will be 1. In terms of representative dataset, we create a sample list of 1000 numbers that contains each and every integer number in the range of 0-255. During post-training quantization, input tensors will be scaled by 1, while output tensors will be scaled by 2, based on the function format.

Table 3.3: Addition

Possible result	Output
0	0
1-2	1
3-4	2
⋮	⋮
2·N + 1 - 2·(N+1)	N+1
⋮	⋮
509-510	255

- **Multiplication:** multiplying two 8-bit integers, results in one 16-bit integer. The minimum possible multiplication is 0·0, which returns 0, while the maximum possible multiplication is 255·255 and returns 65025. So we have 65025 possible results, that need to be stored only in 256 positions inside the Edge TPU.

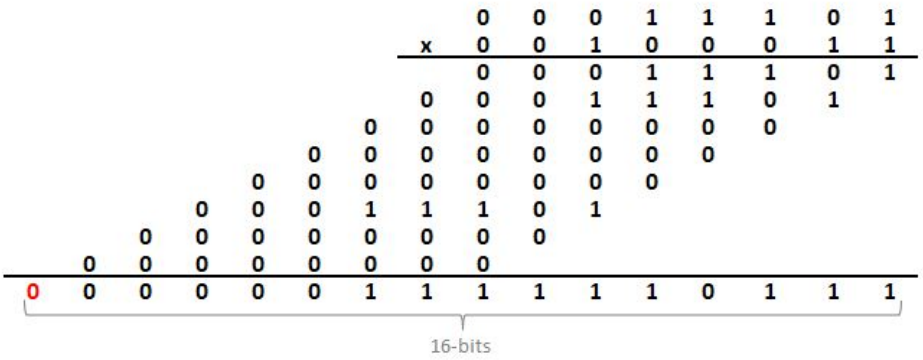


Figure 3.13: Multiplying 8-bit unsigned integers

In Table 3.4 we can see the mapping of every possible algebraic result, to the 8-bit output. After de-quantization from the Edge TPU, every value will be scaled by 255, in order to return the expected output; maximum error will be 128. In terms of representative dataset, we create a sample list of 1000 numbers that contains each and every integer number in the range of 0-255. During post-training quantization, input tensors will be scaled by 1, while output tensors will be scaled by 255, based on the function format.

Table 3.4: Multiplication

Possible result	Output
0 - 126	0
127 - 381	1
382 - 636	2
⋮	⋮
255·N+127 - 255·(N+1)+126	N+1
⋮	⋮
255·254+127 - 255·255	255

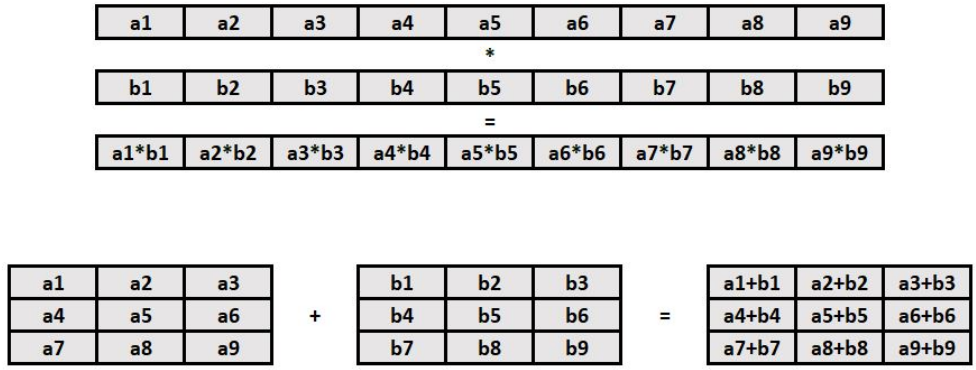


Figure 3.14: Element-wise operations Nx1 or NxN

Listing 3.5: Add & Multiply Examples

```
Example -> Add

TPU Input 1 :       [[[198 190 231  82  15  99  36   7]]]
TPU Input 2 :       [[[ 72 115  28 124 176  16  44 221]]]
Scale:              2.0

Result:             [135 152 130 103 96 58 40 114]
Scaled Output :     [270 304 260 206 192 116  80 228]
Expected Result:    [270. 305. 259. 206. 191. 115.  80. 228.]
Error:              [ 0.  1. -1.  0. -1. -1.  0.  0.]


Example -> Multiply

TPU Input 1 :       [[[126 127 127 127 127 255]]]
TPU Input 2 :       [[[  1   1   3   4   5 255]]]
Scale:              255.0

Result:             [0 1 1 2 2 255]
Scaled Output :     [0  255 255 510 510 65025]
Expected Result:    [126. 127. 381. 508. 635. 65025.]
Error:              [126. -128. 126. -2. 125. 0.]
```

N-bit Operations

We have already seen that performing algebraic operations on the TPU requires compromises in terms of accuracy even if the inputs are 8-bit numbers. In the case of larger bitwidth numbers, we expect that the compromise will be even bigger, so we need to

examine whether this accuracy loss is admissible or we need find other solutions.

The only modification we will make to our model is the representative dataset, which will designate the scale factors for the input and output tensors. In particular, we will study 16 and 32 bit operations. 16bits unsigned integers are ranging between 0 and 65535, while 32-bit integers between 0 and 4294967295. This makes it even more difficult to represent in the range of 8-bit numbers.

- Adding 16-bit unsigned integers results in 17-bit integers, while 32-bit integers lead to 33-bit integers. In the first case, input tensors need to be scaled down 255:1, so that they are mapped into 8-bit numbers, while output sensors will be scaled up 2·255 times, in order to represent 17-bit integers. Accordingly, 32-bit input tensors will need to be scaled down 65535:1 for 16-bit mapping and 255:1 more for 8-bit mapping, while 8-bit output tensors will be scaled 2·255·65535 times to represent 33-bit integers.
- In multiplication, the expected output is 32-bit and 64-bit integers respectively. Input and output tensor scaling will follow the same procedure as above. Table 3.5 summarizes the cases presented above, concerning bitwidth and tensor scaling.

Table 3.5: Bitwidth & Scaling

	8-bits	16-bits	32-bits
Addition			
Input scale	1	255	255·65535
Output scale	2	2·255	2·255·65535
Multiplication			
Input scale	1	255	255·65535
Output scale	255	255·65535	255·65535·4294967295

3.3.2 Matrix Multiplications

Matrix multiplication is a binary operation that produces a matrix from two matrices. It is an algebraic operation that combines multiplication and addition. It works as follows:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in} + b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj} \tag{3.1}$$

Our reasoning is that matrix multiplication requires two operation to be executed in series before giving a result. This indicates that the systolic array of our accelerator can inference these operations faster than a common ARM processor. Therefore, our target is to evaluate whether the Edge TPU can provide satisfactory acceleration on a small and low-level operation like matrix multiplication.

Matrix multiplication function is similar to single multiplications with the distinction that the outputs of multiplication stage are added together. For example, for 3x3 matrices, equation 3.1 for cell (1,1) gives:

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} \tag{3.2}$$

In general, multiplication of NxN matrices require N **parallel** element-wise multiplications and N-1 element-wise additions **in series**. The representative dataset we define consists only of 8-bit number ranging from 0 to 255; input tensors scale is equal to 1, while output tensor scale depends on the matrices size. So for the multiplication we scale by 255, and then for (N-1) additions we scale up by (N-1)+1=N.

Example

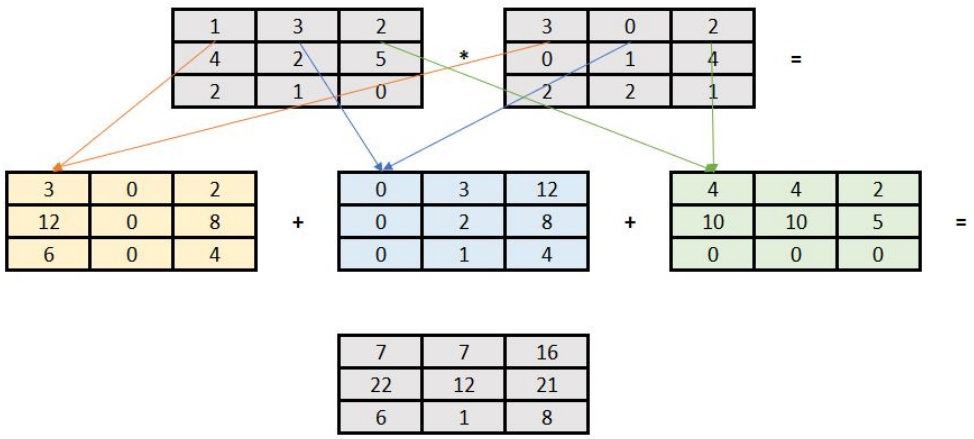


Figure 3.15: 3x3 Matrix multiplication example

Listing 3.6: MatMul Example

```
Example -> MatMul

TPU Input 1:        [[[[156  100]
                    [ 88  133]]]]
TPU Input 2:        [[[[15  190]
                    [165   22]]]]
Scale:              510.0
```

```
Result:             [37 62 46 39]
Scaled Output:      [18870. 31620. 23460. 19890.]
Expected Result:    [18840. 31840. 23265. 19646.]
Error:              [ -30.  220. -195. -244.]
```

3.3.3 N-bit Custom Operations

In both implementation we presented before, we noticed that performing algebraic operations with bitwidth larger than 8, adds intolerable error into the calculations. Extensive results about the mean-error of any operation we have already discussed or we will mention in this section, are presented in Chapter 4. Our goal, in this section, is to experiment with custom designs of element-wise or matrix multiplications in order to solve the problem of limited bitwidth operations. So, **our proposal will have to do with breaking N-bit integers into multiple 8-bit integers.**

3.3.3.1 N-bit element-wise operations

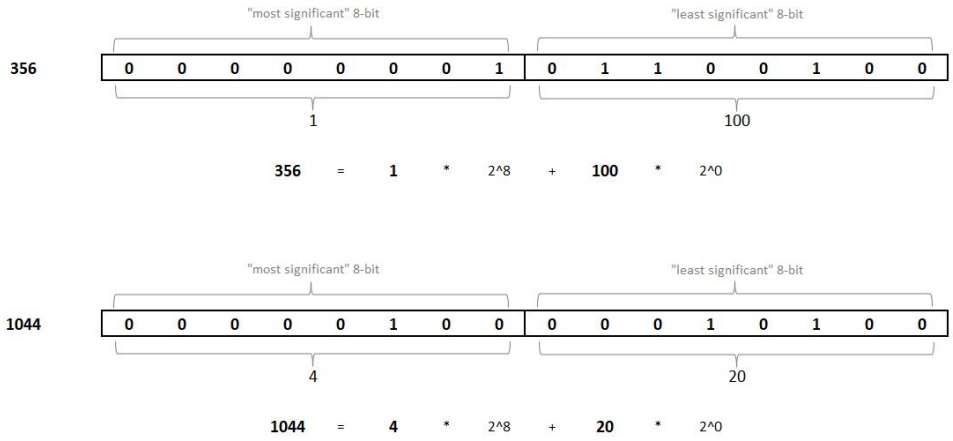


Figure 3.16: Example: Break 16-bit integers into 8-bit parts

In general, any N-bit number (16,32,64,... bits) can be broken into N/2 8-bit parts following the same procedure. Implementing this procedure on the Edge TPU, required building a function that converts any given input of defined bitwidth, into 8-bit numbers. In particular, this function takes two integers as inputs and brakes each one of them into two parts, until width is equal to 8. For example, 32-bit inputs are firstly broken into 16 numbers; first 16 bits are distinguished by right shifting and the last 16 bits by modulus division with 2^{16}. Then, each part is again broken into 2 8-bit numbers. So, at the end, both 32-bit inputs are broken into 4 8-bit numbers. This procedure is presented in the following code snippet.

Listing 3.7: Conversion function

```
def convert(input1,input2,value):
  x = [None] * 4
  a = input1
  b = input2
  x[0] = (a>>int(value/4)>>int(value/4)>>int(value/4)>>int(value
     /4))
  x[1] = (a%2**(value))
  x[2] = (b>>int(value/4)>>int(value/4)>>int(value/4)>>int(value
     /4))
  x[3] = (b%2**(value))

  if value>8:
    y1 = convert(x[0], x[1], int(value/2))
    y2 = convert(x[2], x[3], int(value/2))
    return [y1,y2]
  else:
    return x[:4]

## input1 = ...
## input2 = ...
## bit = ...

value = int(bits/2)
x = convert(input1,input2,value)
```

Listing 3.8: Concrete function

```
@tf.function
def anybit(*arg):
  dim1 = int(bits/8)
  print("1st input num of 8bits: ",dim1)
  dim2 = int(bits/8)
  print("2nd input num of 8bits: ",dim2)

  return [[tf.multiply(arg[i], arg[dim1+k]) for k in range(dim2)
     ] for i in range(dim1)]

tensor = anybit.get_concrete_function(input1,input2)
```

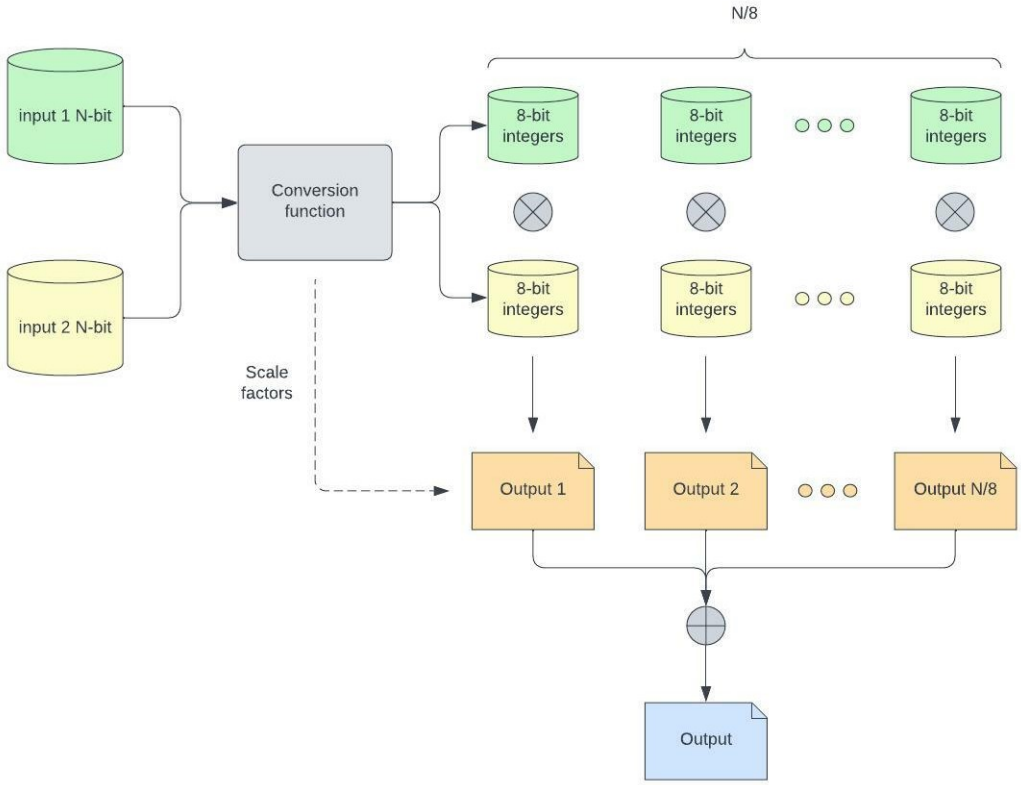


Figure 3.17: N-bit inference workflow

Implementing this concept on the Edge TPU, requires building TF Function and setting input and output tensors with variable values. For any desired bitwidth, we consider each input's broken parts as a list and perform the calculation on each specific part, either addition or multiplication. The workflow we follow is presented in Figure 3.17. Also, on Table 3.6 we display the number of operations and I/Os for the implementation of any bitwidth calculation.

Table 3.6: Bitwidth & Operations

Bitwidth	operations	inputs	outputs
16-bits	2	4	2
32-bits	4	8	4
64-bits	8	16	8
⋮	⋮	⋮	⋮
N-bits	N/8	N/4	N/8

Listing 3.9: 16-bit Multiply Example

```
Example -> Multiply
```

```
Input 1:              [16771 9409]
Input 2:              [6730 32248]
--------------------------------------

TPU Input 1:          [[[65 36]]]
TPU Input 2:          [[[131 193]]]
TPU Input 3:          [[[26 125]]]
TPU Input 4:          [[[74 248]]]

Scale:                255.0

----------RESULTS----------

Result 1:             [7 18]
Scaled Output 1:      [1785 4590]
Factor 1:             16
Total 1:              [[[[116981760 300810240]]]]

Result 2:             [19 35]
Scaled Output 2:      [4845 8925]
Factor 2:             8
Total 2:              [[[[1240320 2284800]]]]

Result 3:             [13 95]
Scaled Output 3:      [3315 24225]
Factor 3:             8
Total 3:              [[[[848640 6201600]]]]

Result 4:             [38 188]
Scaled Output 4:      [9690 47940]
Factor 4:             0
Total 4:              [[[[9690 47940]]]]

--------------------------------------

Sum:                  [119080410 309344580]
Expected Result:      [112868830 303421432]
Error:                [-6211580  -5923148]
```

3.3.3.2 N-bit Matrix Multiplications

Except for element-wise operations, we will also explore an approach for more complex operations, like matrix multiplication. We will follow the same workflow we have already discussed, however it will need to be adapted to NxN matrices. Each N-bit element of the matrices will be broken into 8-bit parts, like we presented in Figure 3.16 and Listing 3.7. Let's say we have two KxK matrices, A and B, that contain N-bit numbers.

$$\textbf{(1)}\ \Big[A\Big] = \Big[A_1\Big] \times 2^{N-8} + \Big[A_2\Big] \times 2^{N-16} + \cdots + \Big[A_N\Big] \times 2^0$$

$$\textbf{(2)}\ \Big[B\Big] = \Big[B_1\Big] \times 2^{N-8} + \Big[B_2\Big] \times 2^{N-16} + \cdots + \Big[B_N\Big] \times 2^0$$

Then, sub-matrices A_i, that now contain only 8-bit numbers, will be multiplied individually with every one of the B_i sub-matrices. This indicates that $(N/8)^2$ matrix multiplications will be performed.

$$\Big[A\Big]\cdot\Big[B\Big] = \Big[A_1\Big]\times\Big[B_1\Big]\times 2^{N-8}\times 2^{N-8} + \Big[A_1\Big]\times\Big[B_2\Big]\times 2^{N-8}\times 2^{N-16} + \cdots + \Big[A_2\Big]\times\Big[B_1\Big]\times 2^{N-16}\times 2^{N-8}$$
$$+ \Big[A_2\Big] \times \Big[B_2\Big] \times 2^{N-16} \times 2^{N-16} + \cdots + \Big[A_N\Big] \times \Big[B_N\Big] \times 2^0 \times 2^0$$

In terms of implementing this concept on the Edge TPU, we will mostly follow the approach we discussed for N-bit element-wise operations. The only distinction between two approaches will have to do with the number of operations based on the model's bitwidth. On Table 3.7 we display the number of operations and I/Os for the implementation.

Table 3.7: Bitwidth & Operations

Bitwidth	operations	inputs	outputs
16-bits	4	4	4
32-bits	16	8	16
⋮	⋮	⋮	⋮
N-bits	$(N/4)^2$	N/4	$(N/4)^2$

Listing 3.10: 16-bit Matrix Multiplication Example

```
Example -> MatMul

Input 1 :       [[10246 11498]
                 [34470 47539]]
Input 2 :       [[17979 40390]
                 [62514 25629]]
-------------------------------
```

```
TPU Input 1 :    [[[ 40  44]
                  [134 185]]]
TPU Input 2 :    [[[  6 234]
                  [166 179]]]
TPU Input 3 :    [[[ 70 157]
                  [244 100]]]
TPU Input 4 :    [[[ 59 198]
                  [ 50  29]]]

Scale:           510.0

----------RESULTS----------

Result 1:            [[27 21]
                      [107 78]]
Scaled Output 1 :    [[13770 10710]
                      [54570 39780]]
Factor 1 :           16
Total 1 :            [[902430720 701890560]
                      [3576299520 2607022080]]

Result 2:            [[57 77]
                      [12 15]]
Scaled Output 2 :    [[29070 39270]
                      [ 6120  7650]]
Factor 2 :           8
Total 2 :            [[7441920 10053120]
                      [1566720 1958400]]

Result 3:            [[24 16]
                      [37 75]]
Scaled Output 3 :    [[12240  8160]
                      [18870 38250]]
Factor 3 :           8
Total 3 :            [[3133440 2088960]
                      [4830720 9792000]]

Result 4:            [[52 87]
                      [35 147]]
Scaled Output 4 :    [[26520 44370]
                      [17850 74970]]
```

```
Factor 4 :          0
Total 4 :           [[26520 44370]
                    [17850 74970]]

_____________________________________

Sum :               [[913032600 714077010]
                    [3582714810 2618847450]]
Expected output:    [[902998806 708518182]
                    [3591589176 2610620331]]
Error:              [[[[-10033794 -5558828]
                    [8874366 -8227119]]]]
```

3.3.4 More designs for evaluation purposes

In this section, we will present some additional model designs we made, that can be enlightening about the evaluation of the TPU architecture.

3.3.4.1 MultiLayer Perceptron

A multilayer perceptron (MLP) is a fully connected class of feedforward artificial neural network (ANN). An MLP is composed of one input layer, one or more hidden layers and one final layer called the output layer (Figure 3.18 [50]). Every layer except the output layer includes a bias neuron and is fully connected to the next layer. Fully Connected layers in neural networks are those layers where all the inputs from one layer are connected to every activation unit of the next layer. In most popular machine learning models, the last few layers are full connected layers which compiles the data extracted by previous layers to form the final output. Fully Connected layers are very commonly used on Classification model; after feature extraction we need to classify the data into various classes and this can be done using a fully connected (FC) neural network to make the model end-to-end trainable. As we notice in Figure 3.18, the neurons of a fully connected layer represent a matrix-vector multiplication between weights and the input/output vectors, known as activation values.

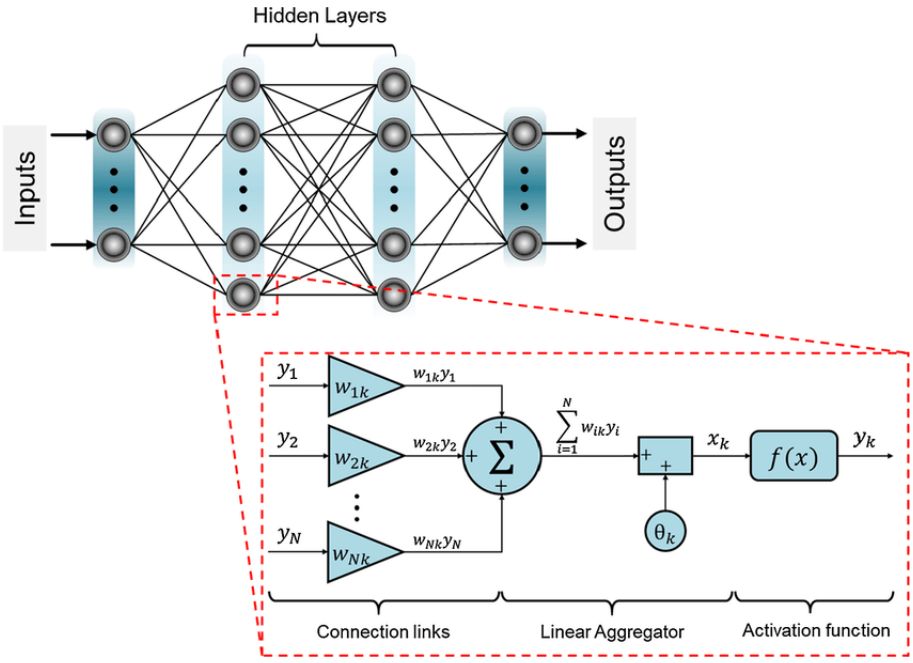


Figure 3.18: Schematic of a fully connected neural network

The implementation of a fully connected layer in TensorFlow is similar to matrix multiplication, however we will create keras model instead of TF Function, as shown below:

Listing 3.11: Concrete function

```
model = Sequential()
model.add(Dense(2000, input_dim=100, activation='relu'))
model.add(Dense(1000, activation='relu'))
model.add(Dense(500, activation='relu'))
model.add(Dense(200, activation='relu'))
model.add(Dense(5, activation='relu'))
model.add(Dense(1, activation='softmax'))
```

3.3.4.2 Custom designs with multiple Operations and I/Os

In this section, we will experiment with multiple operation and input/output models, that may not have any real application, however will help us to identify patterns that will reveal information about the architecture of the systolic array of the TPU accelerator. Specifically, we will examine three key aspects of inferencing on TPU:

1. How different amount of inputs and outputs affect inference time.
2. How many operations can be executed in parallel.
3. How quantization type affects inference time. Float32 models are quantized on ARM processor, uint8 models on Edge TPU and int8 are not quantized at all.

Listing 3.12: Multiple muls in parallel

```
@tf.function
def parallel(x1,x2,x3,x4,...)
    :
  y1 = tf.mul(x1,x2)
  y2 = tf.mul(x3,x4)
  ...
  return [y1,y2,...]
```

Listing 3.13: Multiple muls in single output

```
@tf.function
def single_output(x1,x2,x3,x4
    ):
  y1 = tf.mul(x1,x2)
  y2 = tf.mul(x3,x4)
  y3 = tf.add(y1,y2)
  return y3
```

Listing 3.14: Multiple connected muls

```
@tf.function
def connected_muls(x1, x2):
  y1 = tf.mul(x1,x2)
  y2 = tf.mul(y1,x1)
  y3 = tf.mul(y2,x1)
  ...
  yN = tf.mul(y(N-1),x1)
return yN
```

Chapter 4

Experimental Evaluation

4.1 Experimental Setup

Following the implementations we presented in chapter 3 about the development of custom general-purpose operations, we will now display all the results we received from experimenting with the Edge TPU, based on the designs mentioned in section 3.3. The experimental evaluation includes building the TensorFlow models, as presented before, exporting the quantized TensorFlow Lite model mapped for both the ARM processor and the TPU, and inferencing both models on the development board. We ran multiple inferences and took into account any possible scenario in order to ensure the validity of our research regarding accuracy and performance. For the experimental evaluation, we connected with development board remotely using the ssh protocol, while the mini board was connected to the host machine by USB. (Figure 2.2). All the models were built in TensorFlow 2.1, converted to TFLite and compiled for the Edge TPU on the host machine using Ubuntu 20.04.

4.2 Evaluation of Custom Operations on Edge TPU

4.2.1 Element-wise Operations

First, we will display the results of the proposed implementation for general algebraic operations, like element-wise additions and multiplications. We evaluated both Nx1 and NxN input sizes; results are presented in Tables 4.1 and 4.2 for each case.

Table 4.1: Average inference time (ms) for Nx1 arrays

	Multiplication		Addition	
inference	**ARM A-53**	**Edge TPU**	**ARM A-53**	**Edge TPU**
quant type	uint8	uint8	uint8	uint8
inputs	2	2	2	2
outputs	1	1	1	1
operations	1	1	1	1
quantizations	3	3	3	3
Array size				
1	0.012	0.342	0.012	0.348
2	0.012	0.278	0.016	0.333
4	0.012	0.344	0.012	0.327
8	0.013	0.394	0.013	0.454
16	0.012	0.363	0.014	0.357
32	0.013	0.389	0.012	0.38
64	0.013	0.396	0.014	0.399
128	0.013	0.37	0.014	0.523
256	0.015	0.606	0.015	0.685
512	0.018	0.584	0.02	0.684
1024	0.024	0.585	0.028	0.513
2048	0.036	0.583	0.043	0.763
4096	0.06	0.762	0.068	0.751
8192	0.112	0.765	0.124	0.576
16384	0.209	0.805	0.235	0.82
32768	0.406	0.859	0.463	1.003
65536	0.801	1.217	0.909	1.07
82944	0.998	1.49	1.134	1.534
131072	1.601	1.909	1.829	2.043
147456	1.813	2.313	2.065	2.383
163840	2.024	2.435	2.302	2.613
172032	2.124	2.618	2.423	2.574

Table 4.2: Average inference time (ms) for NxN arrays

	Multiplication		Addition	
inference	**ARM A-53**	**Edge TPU**	**ARM A-53**	**Edge TPU**
quant type	uint8	uint8	uint8	uint8
inputs	2	2	2	2
outputs	1	1	1	1
operations	1	1	1	1
quantizations	3	3	3	3
Array size				
1x1	0,012	0,273	0,012	0,296
2x2	0,012	0,281	0,013	0,292
4x4	0,012	0,651	0,012	0,295
8x8	0,013	0,273	0,013	0,301
16x16	0,015	0,445	0,015	0,274
32x32	0,024	0,302	0,026	0,258
64x64	0,06	0,299	0,067	0,693
128x128	0,207	0,965	0,245	0,952
256x256	0,799	2,22	0,915	1,108

Figures 4.1 and 4.2 compare the average inference time per array size between ARM A-53 CPU and the Edge TPU for both multiplications and additions. The results lead to five observations on the Edge TPU.

1. Inferencing on TPU has an overhead of 0.25-0.3 ms, that involves data transportation and communication with the memory.

2. Linear algebra implementations that frequent branching or are dominated element-wise by algebra do not perform well on TPUs, because they do not exploit the advantages of TPU's systolic array.

3. Performing operations with NxN input tensor dimensions is far more efficient and faster than Nx1 dimensions.

4. The ASIC of Edge TPU is based on a systolic array of multipliers and accumulators, that can perform a series of operations in parallel. Our estimation of a 64x64 array seems to be confirmed based on our results, as the inference time for NxN arrays rises exponentially for arrays larger than 64x64.

5. The Edge TPU cannot serve matrix operations of dimensions larger than 2^8x2^8.

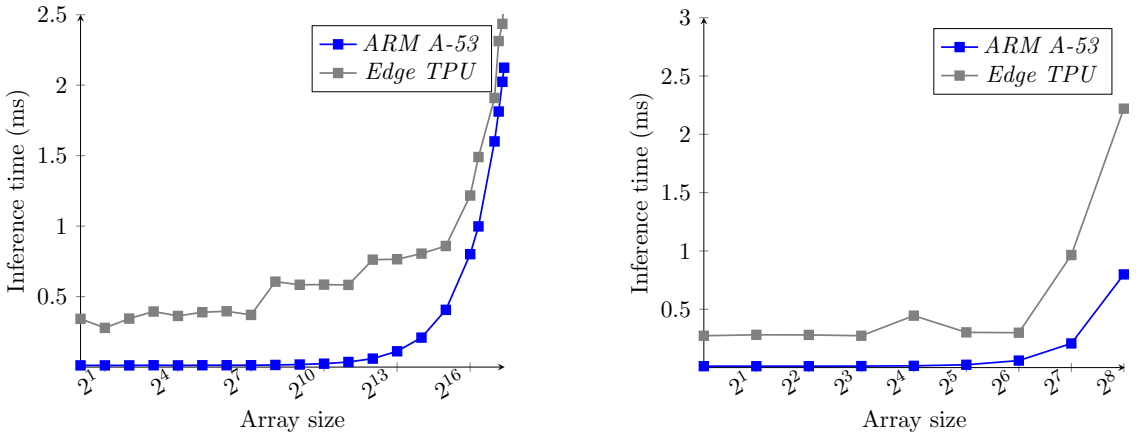


Figure 4.1: Nx1 vs NxN multiplications

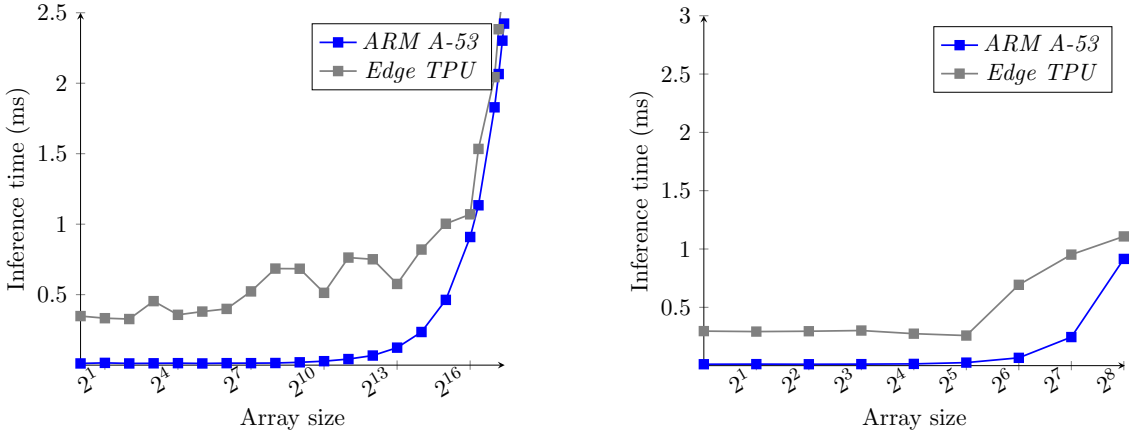


Figure 4.2: Nx1 vs NxN additions

As we have already mentioned in previous chapters, performing algebraic operations on the TPU requires compromises in terms of accuracy even if the inputs are 8-bit numbers. In the case of larger bitwidth numbers, the compromise is even bigger. Therefore, we have examined and present you (Tables 4.3 and 4.4) the percentage error of both element-wise operations for 8, 16 and 32-bit inputs. The size of the sampling pool that lead to these results is 170000 samples.

Table 4.3: Percentage Error of *multiplications*

	8-bit integers	**16-bit integers**	**32-bit integers**
Mean	2.9 %	4.6 %	4.6 %
Max	100.0 %	100.0 %	100.0 %
Min	0.0 %	0.0 %	0.0 %

Table 4.4: Percentage Error of *additions*

	8-bit integers	**16-bit integers**	**32-bit integers**
Mean	0.28 %	1.1%	1.1 %
Max	100.0 %	100.0%	100.0 %
Min	0.0 %	0.0%	0.0 %

4.2.2 Matrix Multiplication

Tensor Processing Units are very fast at performing dense vector and matrix computations. So, in this section we evaluate the performance of the Edge TPU on matrix multiplications. Results are presented on Table 4.5 and visualized in Figure 4.3. In this case, we decided to include measurements on the ARM processor with the original TensorFlow Lite before it was quantized to the 8-bit equivalent, so that we can have an understanding of the performance gains of the smaller quantized models.

Table 4.5: Average inference time (ms)

	NxN			N^2x1	
inference	**ARM A-53**	**ARM A-53**	**Edge TPU**	**ARM A-53**	**Edge TPU**
quant type	no	uint8	uint8	uint8	uint8
inputs	2	2	2	2	2
outputs	1	1	1	1	1
operations	3	3	3	2	2
quantizations	0	3	3	3	3
Array size					
N=1	0,014	0,015	0,27	0,015	0,283
N=2	0,015	0,016	0,277	0,015	0,321
N=4	0,015	0,016	0,287	0,015	0,288
N=8	0,015	0,017	0,296	0,016	0,389
N=16	0,018	0,022	0,297	0,018	0,441
N=32	0,033	0,043	0,322	0,027	0,462
N=64	0,162	0,146	0,322	0,056	0,843
N=128	0,833	0,769	**0,488**	0,187	0,938
N=256	6,807	4,806	**1,288**	-	-
N=288	9,729	6,646	**1,55**	-	-

The conclusion we reach concerning matrix multiplication on the TPU development board, is that for input tensor dimensions smaller than 64x64, the inference time on the Edge TPU is one (1) order of magnitude larger than the one on the ARM processor,

while for bigger dimensions, the systolic array achieves great parallelization leading to acceleration over the ARM A-53; 4x times faster than the 8-bit quantized model executed and up to 7x times faster than the 32-bit floating point model on the same ARM processor.

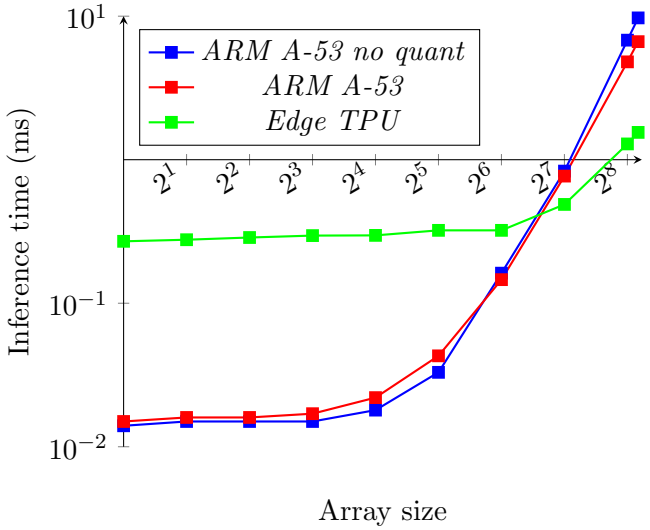


Figure 4.3: NxN matrix multiplications

Similar to the element-wise operation, execution on the TPU accelerator trades off performance against accuracy. Table 4.6 displays the percentage error of matrix multiplications comparatively to the input tensor dimensions. In matrix multiplications, the bigger the array size, the more additions are executed, hence bigger accuracy loss. However, the bigger the array size, the lesser contribution has each addition factor in the output, hence smaller accuracy loss. Needless to say, multiplication's accuracy loss is higher than addition's, so in the end, bigger array size is preferred.

Table 4.6: Percentage Error of *matrix multiplications*

Array size	Mean error	Max error
1x1	2.27 %	100.0 %
2x2	0.95 %	100.0 %
4x4	0.52 %	21.1 %
8x8	0.45 %	3.5 %
16x16	0.41 %	1.6 %
32x32	0.41 %	1.4 %
64x64	0.40 %	1.14 %
128x128	0.39 %	0.99 %
256x256	0.40 %	0.95 %
288x288	0.40 %	0.93 %

4.2.3 Custom N-bit Operations

In this section, we will present the results of experimenting with custom designs of element-wise or matrix multiplications in order to solve the problem of limited bitwidth operations, by breaking N-bit integers into multiple 8-bit integers.

4.2.3.1 N-bit Element-wise Operations

As we have already discussed, in chapter 3, any N-bit number (16,32,64,... bits) can be broken into N/2 8-bit parts and then multiplied or added to any other number of the same bitwidth. Figure 4.4 illustrates the operations and I/Os interconnections of a 16-bit multiplication model on the Edge TPU.

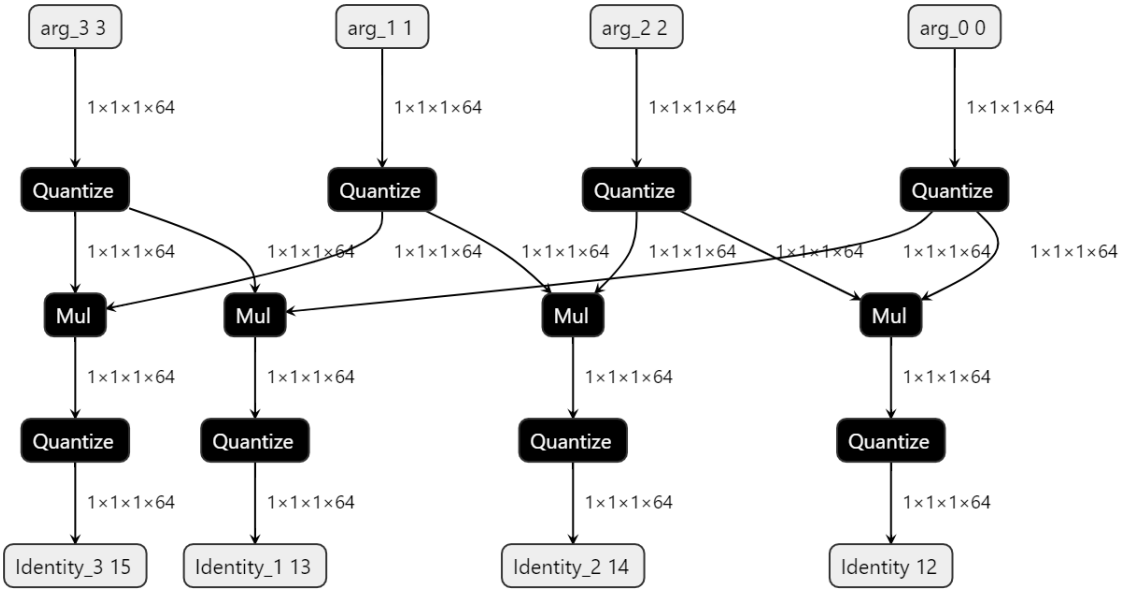


Figure 4.4: 16-bit multiplications

Then, we will display the results of the proposed implementation for N-bit element-wise operations for both Nx1 and NxN input sizes; results are presented in Tables 4.7 and 4.8 for each case.

Table 4.7: Average inference time (ms) for Nx1 arrays

	16-bit		32-bit	
inference	**ARM A-53**	**Edge TPU**	**ARM A-53**	**Edge TPU**
quant type	uint8	uint8	uint8	uint8
inputs	4	4	8	8
outputs	4	4	16	16
operations	4	4	16	16
quantizations	8	8	24	24
Array size				
1	0.016	0.5	0.027	1.49
2	0.016	0.531	0.028	1.39
4	0.016	0.597	0.03	1.41
8	0.017	0.573	0.033	1.39
16	0.016	0.577	0.029	1.49
32	0.016	0.589	0.03	1.49
64	0.018	0.712	0.034	1.35
128	0.02	0.547	0.042	1.48
256	0.025	0.755	0.06	1.50
512	0.034	0.754	0.093	1.72
1024	0.055	0.909	0.155	1.64
2048	0.9	0.733	0.284	1.77
4096	0.165	0.952	0.55	1.83
8192	0.32	1.05	1.08	2.3
16384	0.622	1.29	2.1	3.4
32768	1.24	2	4.3	5.6
65536	2.5	3.6	8.9	10.2
82944	3.2	4.2	11	12.5
131072	5.1	6.3	18.1	20.3
147456	5.8	7	20.4	21.8
163840	6.4	7.7	22.6	

Table 4.8: Average inference time (ms) for NxN arrays

	16-bit		32-bit	
inference	**ARM A-53**	**Edge TPU**	**ARM A-53**	**Edge TPU**
quant type	uint8	uint8	uint8	uint8
inputs	4	4	8	8
outputs	4	4	16	16
operations	4	4	16	16
quantizations	8	8	24	24
Array size				
1x1	0.015	0.584	0.027	1.3
2x2	0.018	0.470	0.029	1.6
4x4	0.016	0.700	0.028	1.2
8x8	0.017	0.709	0.034	1.5
16x16	0.025	0.680	0.058	1.7
32x32	0.056	0.678	0.155	1.5
64x64	0.165	0.929	0.549	1.8
128x128	0.629	1.31	2.2	3.9
256x256	2.5	3.9	8.9	33.5

The results lead to some worth-mentioning observations on the Edge TPU. Specifically, the most import observation is that, the increase in inference time is linear to the number of operations executed on the accelerator. However, this is not applicable for smaller input dimensions, and cannot be noticed in our tables because of the added overhead by the TPU. Moreover, our results confirm the estimation of a 64x64 systolic array as the inference time for NxN arrays rises exponentially for arrays larger than 64x64, as shown in Figure 4.5.

Table 4.9: Effect of number of operations in inference time

	8-bit integers	**16-bit integers**	**32-bit integers**
Operations No.	4	12	40
Latency Nx1	2.3	7	21.8
Latency NxN	0.965	1.31	3.9

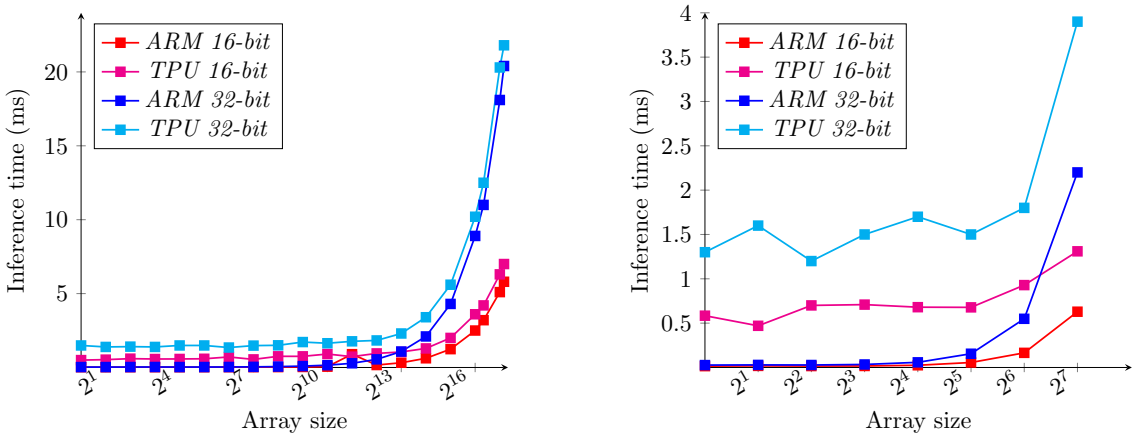


Figure 4.5: Nx1 vs NxN multiplications

In terms of accuracy improvement, our proposed design reaches its goal in the case of 16-bit inputs integers, decreasing the calculations' accuracy from 4.6 % to 2.5 % with a small trade-off in latency; the custom design is up to 100 % slower than the 8-bit implementation.

Table 4.10: Percentage Error of N-bit *multiplications*

	16-bit integers	**32-bit integers**	**64-bit integers**
Mean	2.5 %	7.3 %	33.2 %
Max	100.0 %	99.7 %	99.7 %
Min	0.0 %	0.0 %	0.0 %

4.2.3.2 N-bit Matrix Multiplication

Following the custom design for element-wise operations, we will present the results of our approach for more complex operations, like matrix multiplication. Figure 4.6 illustrates the operations and I/Os interconnections of a 16-bit matrix multiplication model on the Edge TPU. We notice that, matrix multiplications are mapped by the compiler as Fully Connected operations, one input of which is transposed in order to be eligible for the calculation. For this reason, it is important to always reconsider how to build an algebraic model with multiple operations, so that the compiler can map correctly every operation.

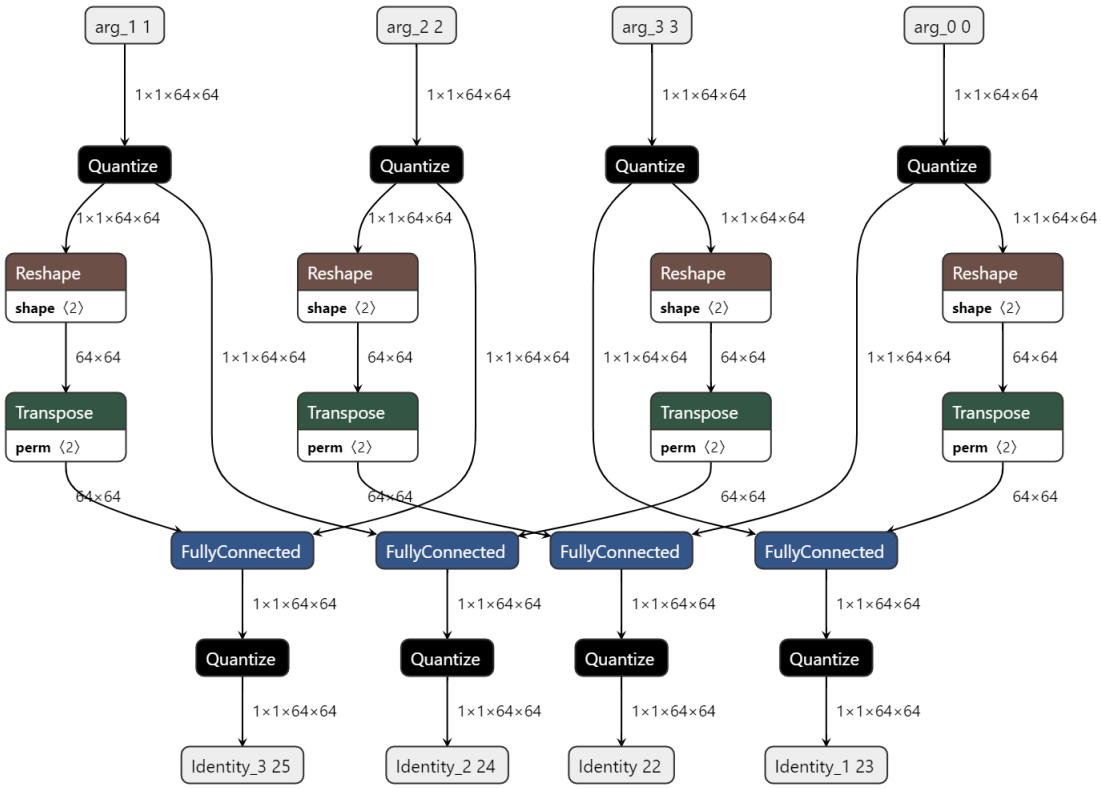


Figure 4.6: 16-bit matrix multiplications

The results displayed on Table 4.11 help us confirm some previously discussed conclusions. Similar to element-wise operations, the increase in inference time is linear to the number of operations executed on the accelerator. Also, Figure 4.7 makes it clear that in matrix multiplication of large dimension input tensors offer significant acceleration. In terms of accuracy (Table 4.12), our proposed design reaches its goal in the case of 16-bit inputs integers, managing the same percentage error for 128x128 input tensors as the 8-bit implementation.

Table 4.11: Average inference time (ms) for NxN arrays

	16-bit		32-bit	
inference	**ARM A-53**	**Edge TPU**	**ARM A-53**	**Edge TPU**
quant type	uint8	uint8	uint8	uint8
inputs	4	4	8	8
outputs	4	4	16	16
operations	12	12	32	32
quantizations	8	8	24	24
Array size				
2x2	0.029	0.630	0.056	1.53
4x4	0.026	0.465	0.055	1.70
8x8	0.03	0.600	0.076	1.73
16x16	0.046	0.982	0.133	1.91
32x32	0.121	0.912	0.392	1.82
64x64	0.506	1.03	1.84	2.4
128x128	2.8	1.41	10.7	4.7

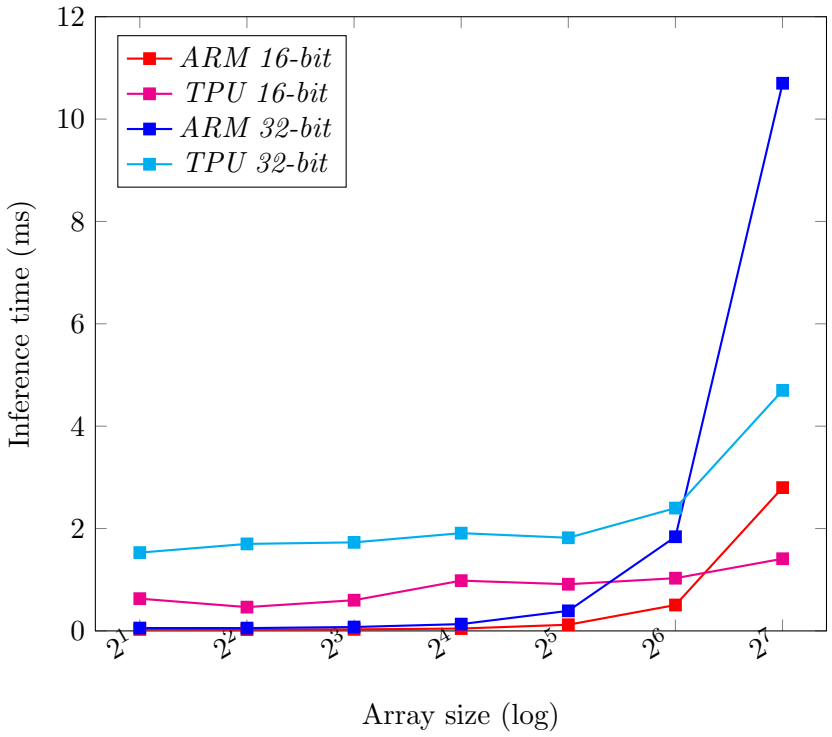


Figure 4.7: N-bit matrix multiplications

Table 4.12: Percentage Error of 16-bit *matrix multiplications*

Array size	Mean error	Max error
128x128	0.39 %	1.04 %

4.2.3.3 MultiLayer Perceptron

Fully connected neural networks are another form of interconnected matrix multiplications. However, in this section we will evaluate different designs, either applicable to real-world applications or testing designs, in terms of number of neurons, number of layers and input size.
In the first part, we will explore real-world applications, that we divide in three categories based on their size and the number of neurons they consist of. The distinction is made based on the average inference time on the ARM A-53 processor; small MLPs are considered those with inference time lower than 1 ms, medium MLPs those between 1 and 10 ms and large MLPs for higher than 10 ms. The results displayed on Table 4.13 lead to some observations on the Edge TPU:

- Single Fully Connected Layer

    1. Small and very big workloads-MLPs **cannot** be accelerated on the TPU. Minimum latency is already around 0.3 ms, due to TPU overhead, so small MLPs are faster when executed on the CPU. On the other hand, large or very large MLPs (model size $>$ 7.6 MB use off-chip memory, something that leads to extensive latency.

    2. Medium sized networks can be **sufficiently** accelerated, with up to 20 times faster inference time over the CPU.

- Multiple Fully Connected Layers

    1. Smaller MLPs with multiple hidden layers have the same-minimum inference time on the TPU, but their complexity increases the latency on the ARM processor.

    2. Medium sized MLPs with multiple hidden layers can be **sufficiently** accelerated, with up to 20 times faster inference time over the CPU.

Table 4.13: Comparison of Real-world MLPs

MLP Layer	Network	Model size (MB)	ARM-A53 (ms)	Edge TPU (ms)	Accel factor
MLP Small	Dense(1000,10) Dense(1)	72.6	0.053	0.3	0.18
MLP Small	Dense(1000,100) Dense(1)	160.6	0.13	0.3	0.43
MLP Mid	Dense(1000,1000) Dense(1)	1040	1.4	0.3	4.7
MLP Mid	Dense(1000,2000) Dense(1)	2020	2.6	0.3	8.7
MLP Mid	Dense(1000,5000) Dense(1)	4950	6.0	0.3	20
MLP Large	Dense(10000,2000) Dense(1)	19390	26	51	0.5
MLP Small	Dense(800,100) Dense(400) Dense(200) Dense(1)	589	0.53	0.3	1.77
MLP Mid	Dense(800,1000) Dense(400) Dense(200) Dense(1)	1290	1.7	0.3	5.7
MLP Mid	Dense(2000,100) Dense(1000) Dense(500) Dense(200) Dense(5) Dense(1)	283	3.6	0.3	12
MLP Mid	Dense(2000,100) Dense(1000) Dense(500) Dense(200) Dense(5) Dense(1)	4600	6.2	0.33	18.8

In the second part, we will custom-built networks with fully connected layers, that serve no real application. Our target is to draw conclusions concerning the effect of model size, number of neurons and number of hidden layers on the Edge TPU.

Table 4.14: Comparison of Custom MLP Networks

Network	Model size (MB)	Off-chip used (Kb)	ARM-A53 (ms)	Edge TPU (ms)	ARM-A35 (ms)	TPU Mini (ms)
Dense(1000,8) Dense(1)	68.6	0	0.04	0.3	0.1	1.1
Dense(7924,8) Dense(1)	256.6	0	0.22	0.3	0.42	1.2
Dense(7925,8) Dense(1)	256.6	124	0.22	0.62	0.42	5.9
Dense(7925,8) Dense(500) Dense(1)	4020	3870	5.5	10.9	7.8	151
Dense(500,8) Dense(10000) Dense(1)	5030	156	6.9	0.7	11	7.3
Dense(500,8) Dense(16000) Dense(1)	7980	7930	10.9	22	17.6	308
Dense(1000,8) Dense(1000)x5 Dense(1)	5010	0	7.3	0.38	11	2.4
Dense(500,8) Dense(500)x20 Dense(1)	5060	0	6.7	0.61	11	4.2
Dense(250,8) Dense(250)x80 Dense(1)	5240	0	6.5	1.14	10.9	8.8

The results displayed on Table 4.14, including inference on the Mini development kit, lead to some important findings about the TPU accelerator:

1. Maximum amount of neurons per FC layer is 7924 neurons. For any number of neurons larger than that, the compiler assigns the model to off-chip memory.

2. The size of off-chip memory used affects linearly the added TPU inference time, as shown in Figure 4.8. Every KB used in the model adds 2.6-2.7 µsec of latency.

3. For the majority of MLPs compiled without the use of off-chip memory, the model size hardly affects the inference time on the TPU. However, for equal model sizes, larger number of layers for less neurons increase the average latency.

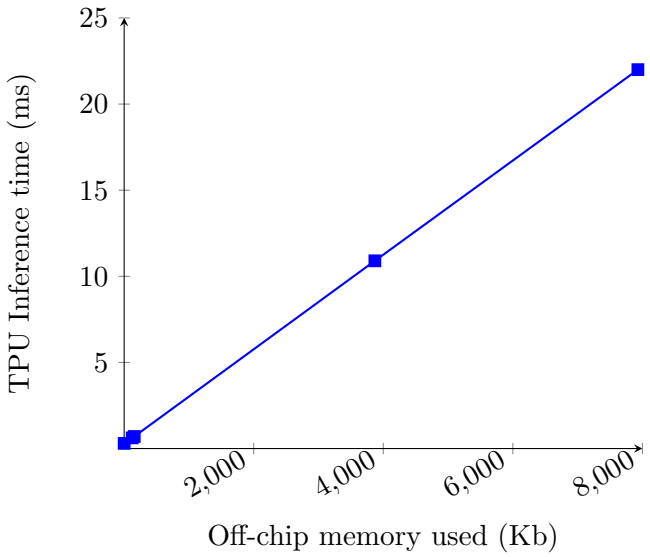
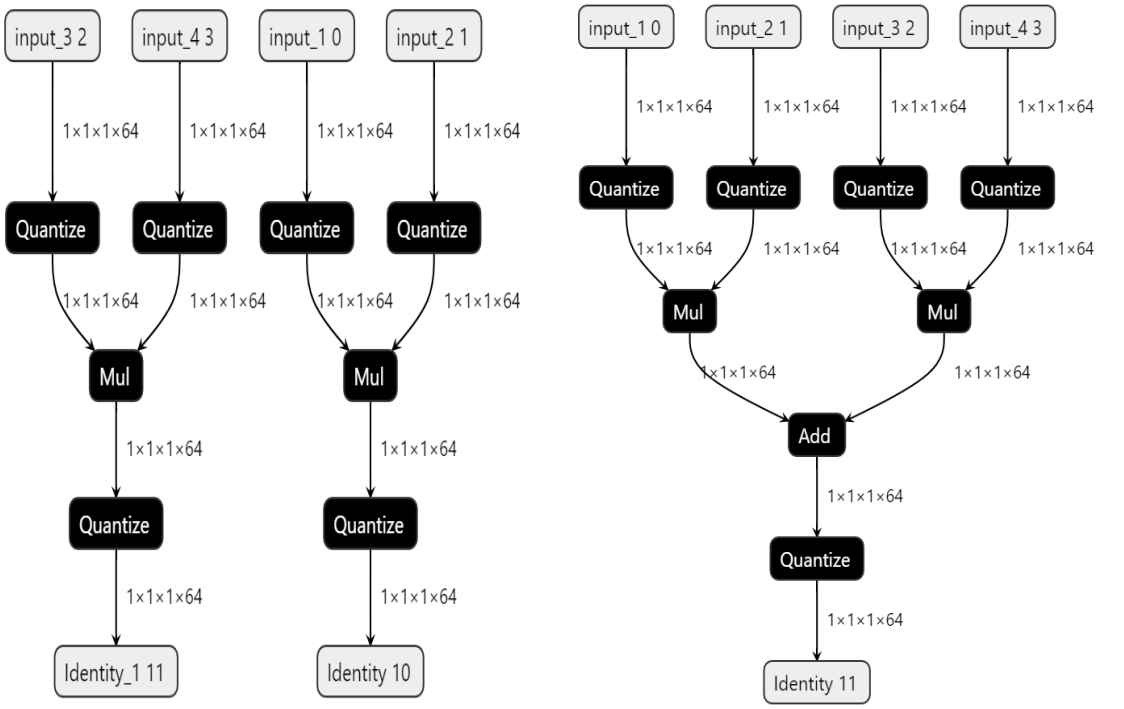


Figure 4.8: Effect of different parameters on inference time

4.2.3.4 Custom Designs with multiple Operations and I/Os

In this final section of evaluating custom designs on the Edge TPU, we will explore designs and models with multiple operations and inputs/outputs, that may not have any real application, however will help us to identify patterns that will reveal information about the architecture of the systolic array of the TPU accelerator. In the first part, we will present comparative results of two similar designs about executing multiplications in parallel. Both designs are displayed side by side in Figure 4.9.



(a) Muls in parallel

(b) Parallel muls added together to give single output

Figure 4.9: Comparison of similar implementations with different no. of outputs

The results of the two different models are presented on Table 4.15. On the left side we display the inference time of parallel multiplications both on ARM A-53 and the Edge TPU, while on the right side we can see the results of parallel multiplications that end up in single output through addition. At first, we can notice that for the same number of inputs (either 4 or 8), the second implementation with one output results in **50 %** less inference time that the first approach. Moreover, it is worth mentioning that the execution of parallel multiplications with additions in order to get one output, can be offer acceleration on the Edge TPU over the ARM processor. Even though, our second design require the same amount of total operations (including quantizations) to be executed, inference on ARM A-53 for larger arrays is noticeably slower. Extensive results on the acceleration factors are included on Table 4.16. Figures 4.10 and 4.11 clearly illustrate the observations and the conclusions we drew for both of our approaches.

Table 4.15: Average inference time (ms) for Nx1 arrays

	Parallel				Parallel w/ single output					
inference	**ARM A-53**		**Edge TPU**		**ARM A-53**			**Edge TPU**		
quant type	uint8	uint8	uint8	uint8	uint8	uint8	uint8	uint8	uint8	uint8
inputs	4	8	4	8	4	8	16	4	8	16
outputs	2	4	2	4	1	1	1	1	1	1
operations	2	4	2	4	3	7	15	3	7	15
quantizations	6	12	6	12	5	9	17	5	9	17
Array size										
256	0.022	0.031	0.369	0.546	0.023	0.033	0.053	0.394	0.427	0.550
512	0.026	0.043	0.382	0.646	0.028	0.048	0.097	0.360	0.405	0.606
1024	0.041	0.067	0.406	0.606	0.042	0.081	0.146	0.396	0.468	0.600
4096	0.114	0.217	0.432	0.765	0.129	0.255	0.506	0.391	0.510	0.783
8192	0.212	0.415	0.543	0.972	0.263	0.484	0.995	0.482	0.637	1.04
16384	0.414	0.815	0.804	1.43	0.459	0.957	1.95	0.570	0.930	1.56
32768	0.815	1.64	1.26	2.5	0.900	1.90	4	0.927	1.52	2.7
65536	1.6	3.3	2.3	4.6	1.81	3.9	8.2	1.5	2.8	4.8
131072	3.4	6.8	4.5	7.2	3.7	8	16.6	3	5	8.9
147456	3.8	7.7	5	7.9	4.2	9	18.6	3.4	5.6	9.9

Table 4.16: Edge TPU acceleration vs ARM A-53

inputs	4	8	16
outputs	1	1	1
operations	3	7	15
Array size			
256	0.06	0.08	0.1
512	0.08	0.12	0.16
1024	0.11	0.17	0.24
4096	0.33	0.50	0.65
8192	0.49	0.76	0.96
16384	0.80	1.03	1.25
32768	0.97	1.25	1.47
65536	1.20	1.39	1.69
131072	1.26	1.59	1.86
147456	1.25	1.60	1.88

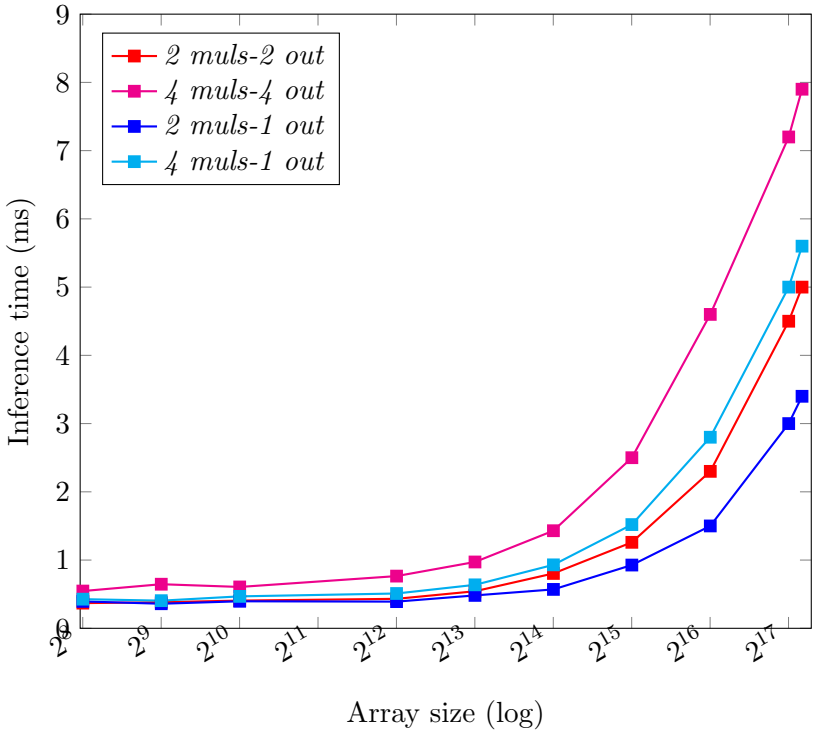


Figure 4.10: Parallel multiplications on Edge TPU

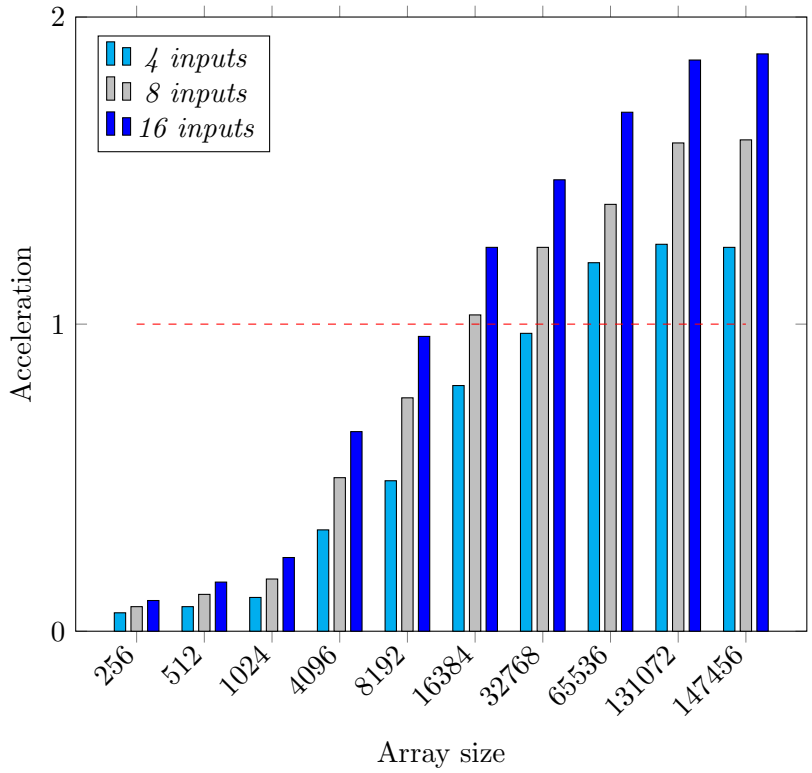


Figure 4.11: Edge TPU Acceleration vs ARM

In the second part of our custom designs' evaluation and as a logical consequence of the first part's findings, we will discuss more extensively the influence of executing connected operations in series all in one inference. Our implementation is displayed in Figure 4.12.

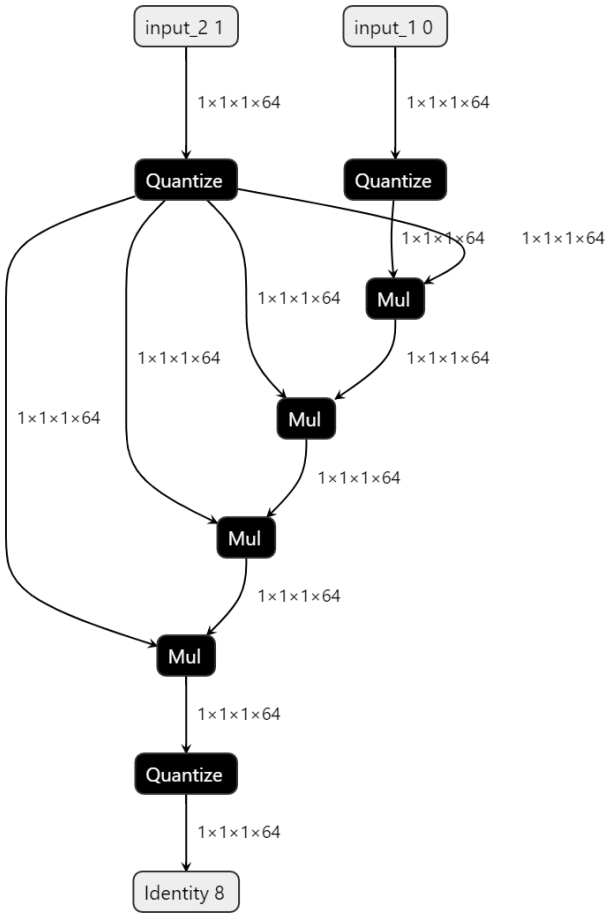


Figure 4.12: Schematic of connected operations with constant I/O

The results of our implementation on the development board are presented on Table 4.17. First of all, we need to take into consideration that the only variable parameter in our evaluation is the number of operations executed in series; number of inputs and outputs is constant.

The conclusion we reach after examining the results is that the Edge TPU accelerator performs better on huge workloads that require extensive computations to be made before accessing the memory to return the results. So in our case, the bigger the number operations and the larger the input dimensions, the bigger the acceleration achieved by the Edge TPU over the ARM A-53 processor. Extensive results on the acceleration factors are included on Table 4.18. Figures 4.13 and 4.14 clearly illustrate the observations and the conclusions we drew for our approach.

Table 4.17: Average inference time (ms) for Nx1 arrays

inference	ARM A-53						Edge TPU					
quant type	uint8						uint8					
inputs	2						2					
outputs	1						1					
quant.	3						3					
operations	1	2	4	8	16	64	1	2	4	8	16	64
Array size												
512	0.018	0.020	0.026	0.035	0.056	0.173	0.382	0.388	0.340	0.339	0.367	0.484
1024	0.027	0.028	0.037	0.056	0.090	0.304	0.346	0.326	0.299	0.363	0.357	0.466
4096	0.061	0.076	0.111	0.172	0.307	1.08	0.319	0.327	0.321	0.358	0.421	0.604
8192	0.116	0.142	0.207	0.340	0.590	2.12	0.348	0.419	0.397	0.434	0.463	0.738
16384	0.217	0.279	0.409	0.661	1.178	4.24	0.459	0.522	0.594	0.533	0.627	1.09
32768	0.414	0.543	0.804	1.32	2.35	8.7	0.676	0.703	0.730	0.813	0.928	1.78
65536	0.812	1.1	1.58	2.61	4.65	16.9	1.25	1.26	1.28	1.47	1.65	3.23
131072	1.64	2.14	3.16	5.22	9.3	33.9	2.32	2.32	2.47	2.73	3.24	6.20
147456	1.83	2.42	3.58	5.88	10.5	38.1	2.52	2.69	2.73	3.02	3.77	7.09

Table 4.18: Edge TPU acceleration vs ARM A-53

operations	1	2	4	8	16	64
Array size						
512	0.05	0.05	0.08	0.1	0.15	0.36
1024	0.08	0.09	0.12	0.15	0.25	0.65
4096	0.19	0.23	0.35	0.48	0.73	1.78
8192	0.33	0.34	0.52	0.78	1.28	2.87
16384	0.47	0.53	0.69	1.24	1.88	3.9
32768	0.61	0.77	1.1	1.62	2.53	4.9
65536	0.65	0.87	1.24	1.77	2.83	5.25
131072	0.71	0.93	1.28	1.91	2.87	5.46
147456	0.72	0.9	1.31	1.95	2.78	5.38

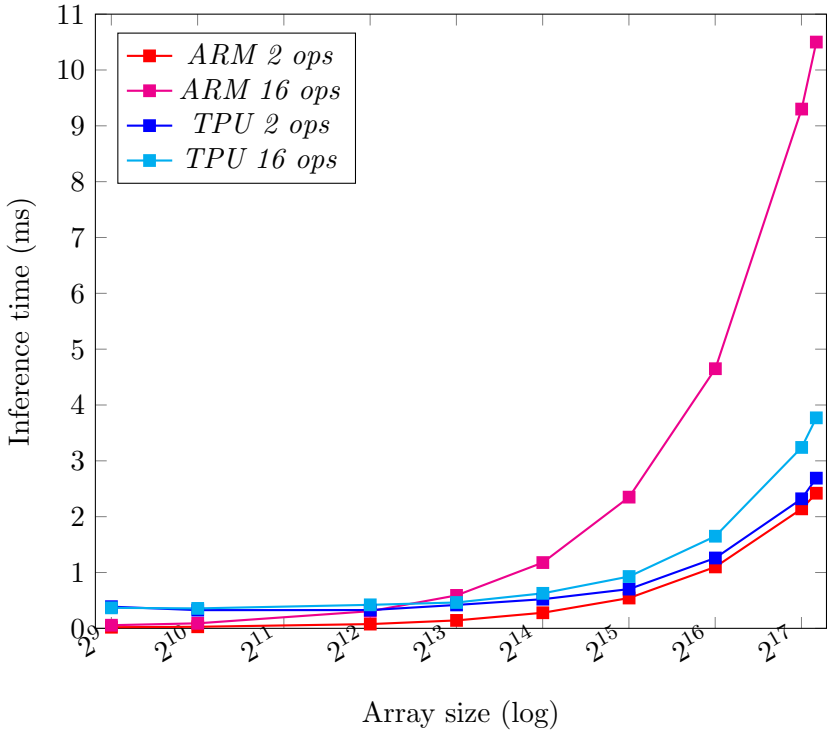


Figure 4.13: Connected multiplications

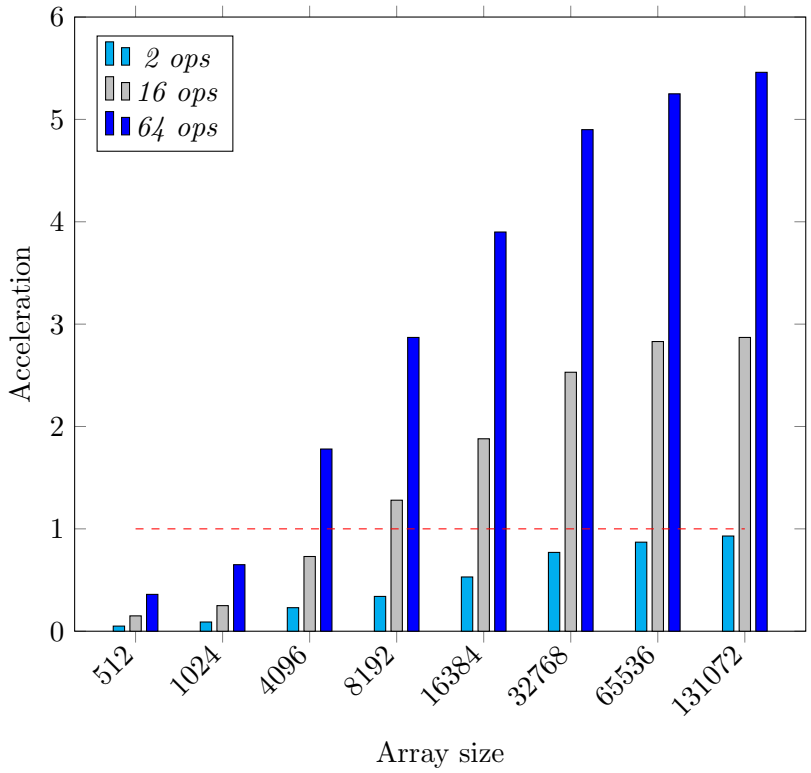


Figure 4.14: Edge TPU Acceleration vs ARM

4.3 General-Purpose Workloads on TPU: Acceleration of Image Processing Kernels

Image processing [51,52] is a method to perform some operations on an image, in order to get an enhanced image or to extract some useful information from it. It is a type of signal processing in which input is an image and output may be image or characteristics/features associated with that image. Image processing basically includes the following three steps:

- Importing the image via image acquisition tools
- Pre-processing and feature extraction
- Output in which result can be an altered image or a report that is based on image analysis

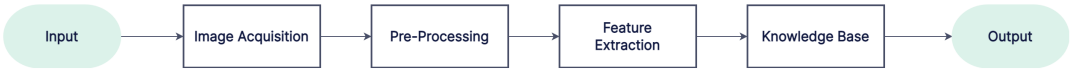


Figure 4.15: Digital Image processing steps

Image processing algorithms, nowadays, pose the need for increasing computation capabilities at a limited power budget. Modern applications, involving machine learning and AI models, require image acquisition, analysis and information extraction to be executed on embedded low-power, low-latency, portable and autonomous devices, requiring novel architecture solutions to be studied and implemented. In this work we present two design cases targeting edge-cutting applications, exploiting Edge TPU development board.

4.3.1 Sobel Edge Detector

4.3.1.1 Edge Detection

Edge Detection [53] is a case of trying to find the regions in an image where we have a sharp change in intensity or a sharp change in color; a high value indicates a steep change and a low value indicates a minor change.

The purpose of detecting sharp changes in image brightness is to capture important events and changes in properties of the world. It can be shown that under rather general assumptions about the image formation process, a discontinuity in image brightness can be assumed to correspond to a discontinuity in either depth, surface orientation, reflection, or illumination.

The result of applying an edge detector to an image may lead to a set of connected curves that indicate the boundaries of objects, the boundaries of surface markings as well as curves that correspond to discontinuities in surface orientation. Thus, applying an edge detection algorithm to an image may significantly reduce the amount of data to be

processed and may therefore filter out information that may be regarded as less relevant, while preserving the important structural properties of an image.

Edge detection is one of the fundamental steps in image processing, image analysis, image pattern recognition, and computer vision techniques.

A very common operator for edge detection is Sobel Operator, which is an approximation to a derivative of an image. It is separate in the y and x directions.

4.3.1.2 Theoretical background

The most usual digital image processing operations are implemented with filtering. Digital filters are used for smoothing, sharpening, and edge enhancement in digital images. Filtering can be performed by:

- convolution with specifically designed kernels (filter array) in the spatial domain
- masking specific frequency regions in the frequency (Fourier) domain

In this work, we will deal with convolution kernels. Convolution is an operation that accomplishes linear filtering of an image; it is a neighborhood operation in which each output pixel is the weighted sum of neighboring input pixels. The matrix of weights is called the convolution kernel, also known as the filter.

In an image processing context, one of the input arrays is normally just a grayscale image. The second array is usually two-dimensional and is known as the kernel. Figure 4.16 shows an example image and a kernel that we will use to illustrate convolution.

The convolution is performed by sliding the kernel over the image, generally starting at the top left corner, so as to move the kernel through all the positions (stride=1) where the kernel fits entirely within the boundaries of the image. Each kernel position corresponds to a single output pixel, the value of which is calculated by multiplying together the kernel value and the underlying image pixel value for each of the cells in the kernel, and then adding all these numbers together.

The **stride** controls how the filter convolves around the input volume. The amount by which the filter shifts is the stride. Stride is normally set in a way so that the output volume is an integer and not a fraction. In the example below we have a 5x5 input image, a 3x3 kernel filter and a stride of 1.

So, the value of the top left pixel in the output image will be given by:

$$O_{1,1} = 3 \cdot 1 + 1 \cdot 0 + 2 \cdot 1 + 2 \cdot 0 + 4 \cdot 1 + 5 \cdot 1 + 0 \cdot 0 + 3 \cdot 1 + 3 \cdot 0 = 17$$

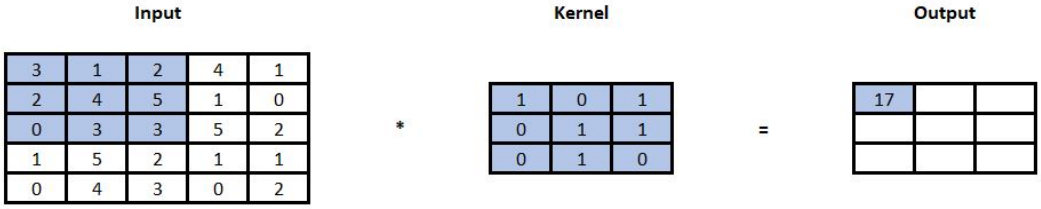


Figure 4.16: 3x3 kernel image convolution

Convolution can be used to implement many different operators, particularly spatial filters and feature detectors. Examples include the Sobel edge detector.

4.3.1.3 Sobel Operator

The **Sobel Operator** [54] performs a 2-dimensional spatial gradient measurement on an image and so emphasizes regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in a grayscale image.

The operator uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives – one for horizontal changes, and one for vertical. The gradient for x-direction has negative numbers on the left hand side and positive numbers on the right hand side and we are preserving a little of the center pixels. Similarly, the gradient for y-direction has negative numbers on top and positive numbers on the bottom and here we are preserving the middle row pixels.

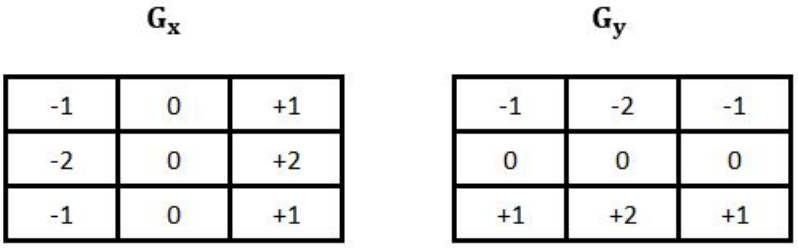


Figure 4.17: Sobel convolution kernels

These kernels (Figure 4.17) are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid. The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation. These can then be combined together to find the absolute magnitude of the gradient at each point and the orientation of that gradient. The gradient magnitude is given by:

$$G = \sqrt{G_x^2 + G_y^2} \tag{4.1}$$

The angle of orientation of the edge (relative to the pixel grid) giving rise to the spatial gradient is given by:

$$\theta = \arctan \frac{G_y}{G_x} \tag{4.2}$$

In this case, orientation equal 0 means that the direction of maximum contrast from black to white runs from left to right on the image, and other angles are measured anti-clockwise from this.

Example

We are given a small (custom) grayscale image 6x5 consisting of 30 pixels. For grayscale images, the pixel value is a single number stored as an 8-bit integer giving a range of possible values from 0 to 255. Typically zero is taken to be black, and 255 is taken to be white.

Essentially what we are trying to do here with the **Sobel Operator** is trying to find out the amount of the difference by placing the gradient matrix over each pixel of our image.

Figure 4.18 shows the result of doing convolution by placing the gradient matrix X over the yellow region of the input image. The calculation is shown on the right which sums up to 200, which is non-zero, hence there is an edge. If all the pixels of images were of the same value, then the convolution would result in a resultant sum of zero. So the gradient matrix will provide a big response when finding a significant divergence in brightness between neighboring pixels.

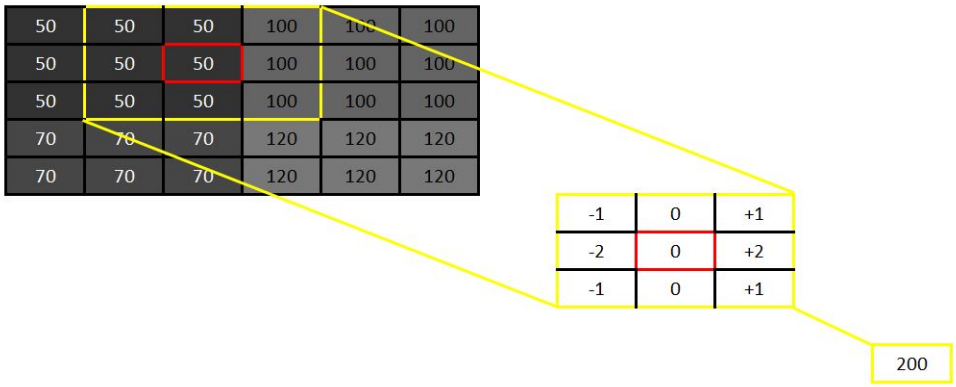


Figure 4.18: 3x3 convolution in the x-direction using G_x.

After repeating the same procedure for the whole image, we get two images as output, one for x-direction and another one for y-direction. So, by using kernel convolution, we can see that, in the example image below, there is an edge both in the horizontal and the vertical direction.

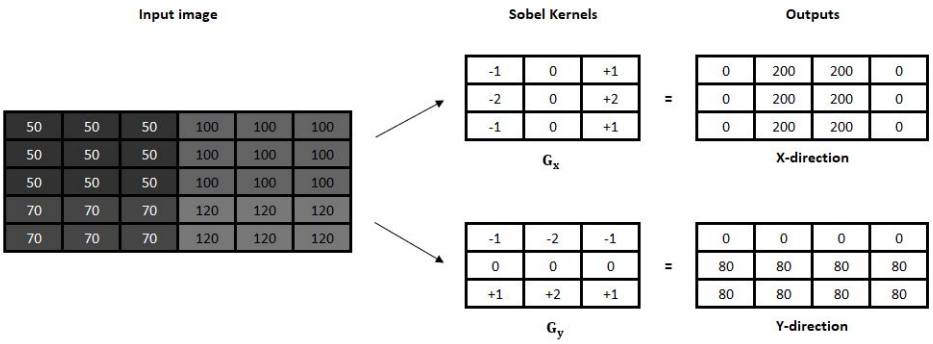


Figure 4.19: Application of sobel operator on image

0	200	200	0
80	215	215	80
80	215	215	80

Figure 4.20: Image gradient magnitude

4.3.1.4 Implementation on Edge TPU board

According to the methodology we have already developed, in relation to running inference on the Edge TPU development board, we will present the implementation of the sobel operator, using Python libraries and TensorFlow framework. We will assess the performance of the operator running on the TPU, comparatively to running exclusively on the ARM processor. At the end, we will present a demo application of the Sobel edge detector on the Edge TPU development board, by loading an input image and receiving the output one on the host machine.

The main stages of the implementation can be found in Figure 4.21. The directory of the Python implementation for the complete sobel operator we have discussed so far can be found it the following github repository: Applications on Google Edge TPU.

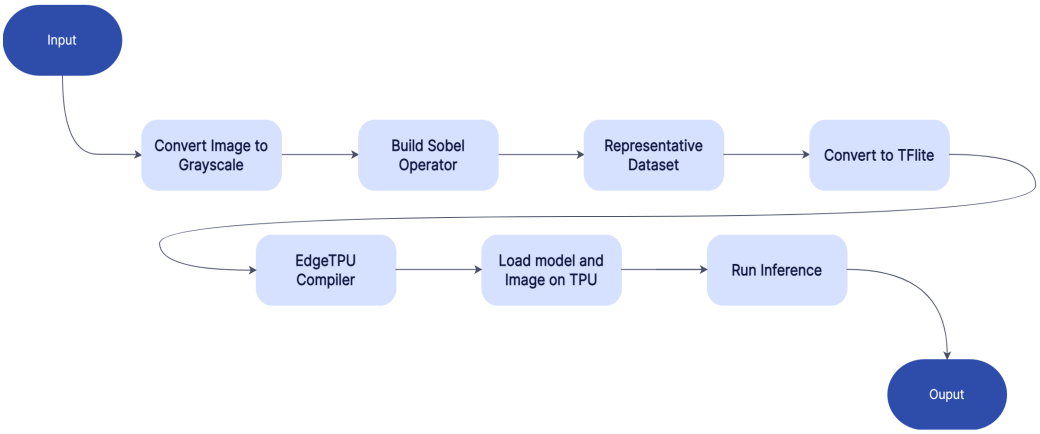


Figure 4.21: Sobel operator process

Figure 4.22 illustrates all the stages of processing that a test image undergoes in order to export the edge map, a new image that describes each original pixel's edge classification. Specifically, we start by converting the input image to grayscale using the python's PIL imaging library. Then, we build the sobel operator as keras model, which consists of a 2-D convolutional layer for each gradient, x and y and 3 additional operations in order to produce the gradient magnitude of the output image. Based on equation 4.1 we use 2x multiplications, 1x addition and a final operation for the square root. We continue by creating the TF Lite model using the TFLiteConverter. However, this TFlite file uses floating-point values for the parameter data, and we need to fully quantize the model to int8 format. To do so, we need to perform post-training quantization with a representative dataset, indicative of single-channel images. So now that we have a fully quantized TensorFlow Lite model, we compile the model for the Edge TPU. At this point, we notice our model was successfully compiled but not all operations are supported by the Edge TPU, as a result a percentage of the model will instead run on the CPU. This happens because the Edge TPU Compiler does not still support operations like square root.

After completing this procedure on the host machine, we load the image and the TFlite models to the Coral device and proceed with inferencing. Following the methodology we discussed on Chapter 3, we receive the final image.

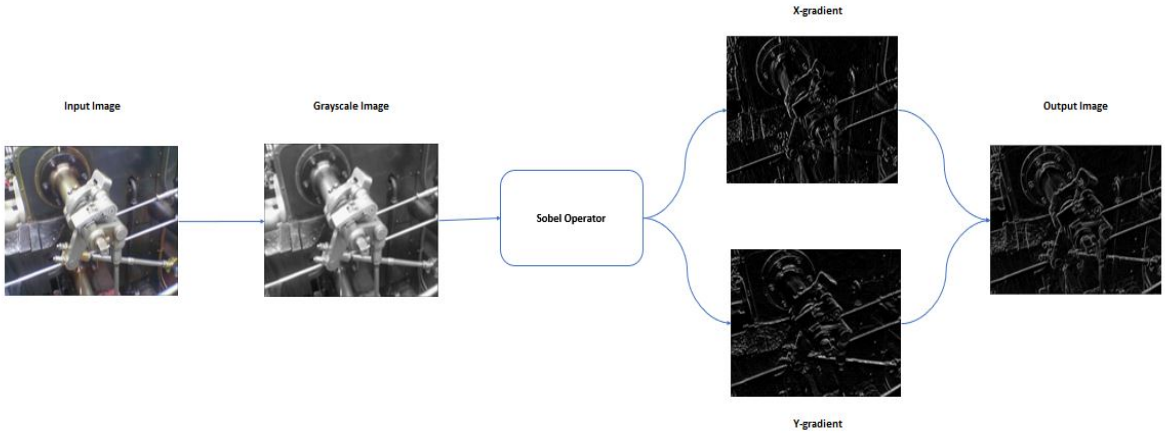


Figure 4.22: Sobel operator on test image

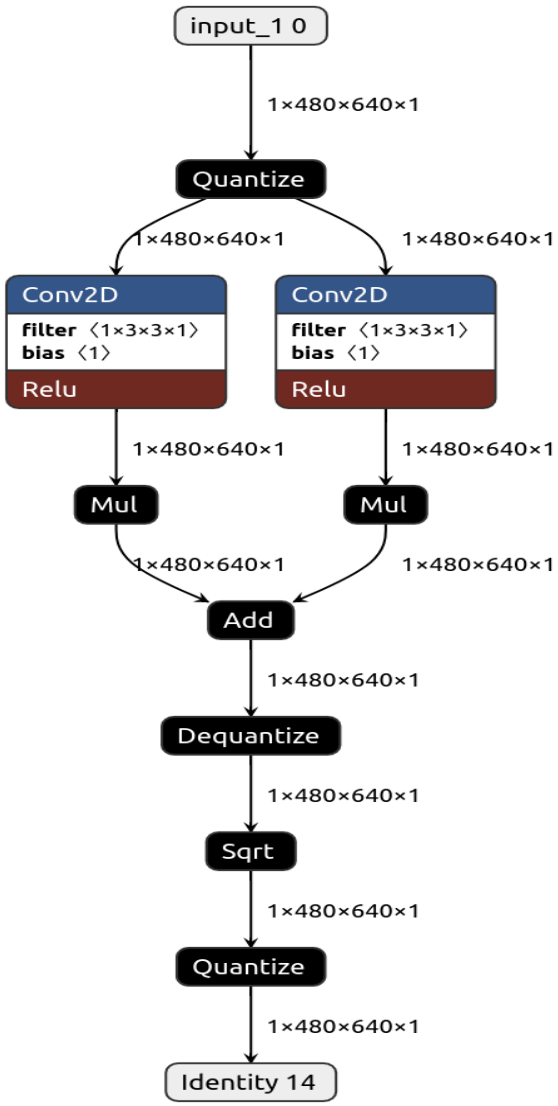


Figure 4.23: Sobel operator TFlite network

The structure of the Sobel operator as TFlite model is depicted in Figure 4.23. As we can notice, the network consists of two 2-dimensional convolutional layers, both of which take the same image as input and give two distinct output, one for each direction, x and y, and 4 algebraic operations. Therefore, it is obvious that the sobel operator is a 1 input & 1 output model consisting of 6 operations and 3 quantizations. Out of all these 9 operations, 3 are executed on the ARM processor; the de-quantization, the calculation of the square root and the final quantization.

After having presented the implementation of the Sobel Edge Detector on the Coral development board, we proceed on assessing whether the on-board Edge TPU co-processor is able to achieve adequate and satisfactory acceleration comparatively to the ARM processor, even though part of the model is executed on the CPU. Table 4.19 presents all the necessary data we collected while conducting measurements on the development board. Our goal is to examine whether and how the input tensor size affects the overall inference time. Our test includes array sizes both small and large, other theoretical and other indicative of real image sizes.

Table 4.19: Average inference time (ms)

inference	**ARM A-53**	**Edge TPU**
quant type	uint8	uint8
inputs	1	1
outputs	1	1
operations	2	2
quantizations	3	3
Array size		
4x4	0.028	0.295
8x8	0.046	0.292
16x16	0.118	0.633
32x32	0.399	0.748
64x64	1.55	**0.612**
128x128	6.03	**2.26**
256x256	25.3	**8.99**
512x512	101.2	**31.5**
1024x1204	405.9	**107.8**
2048x2048	1616	**404.5**

The results displayed on Table 4.19 help us confirm some previously discussed conclusions from the custom general purpose operations we evaluated. First of all, the visualization of the results in Figure 4.24 makes it clear that inferencing the Sobel Edge Detector, a classic Convolutional Neural Network, on the Edge TPU achieves significant acceleration

in comparison to the ARM A-53 processor. For larger input dimensions, indicative of high resolution images, the TPU can be up to 4x times faster than the CPU. Moreover, Figures 4.25 are in support of our estimation that the Edge TPU's systolic array is a 64x64 array, because we can notice the exponential increase in latency for arrays larger than 64x64. However, we also need to take into consideration that part of the Sobel operator is executed on the ARM processor, adding significant latency to execution because of data transportation and communication between the TPU and the CPU, something that gets worse the bigger the input tensor dimensions.

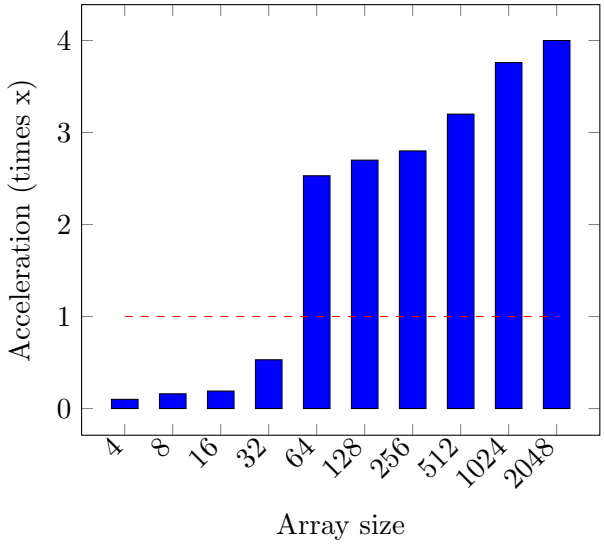
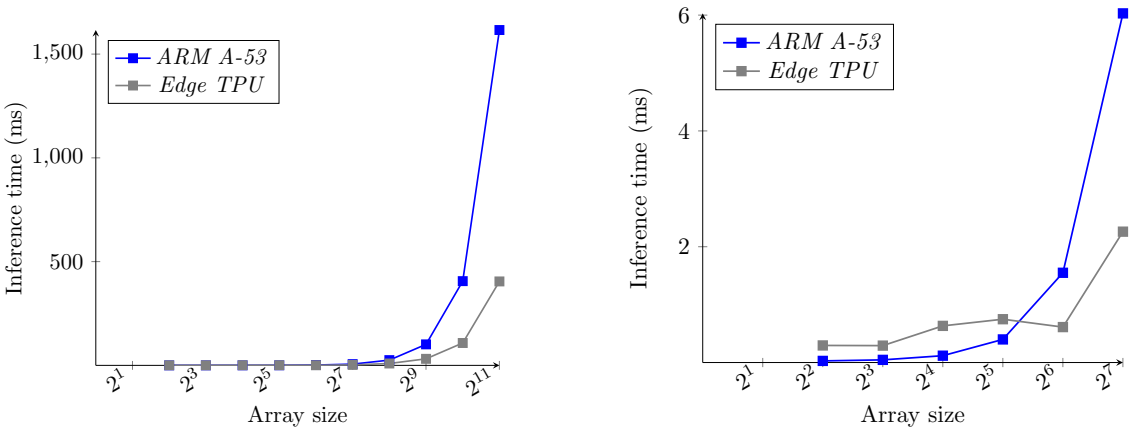


Figure 4.24: Acceleration of Sobel Operator on Edge TPU vs ARM A-53



(a) Total results

(b) Zoom in smaller arrays

Figure 4.25: Sobel operator inference time

4.3.2 Binning in Digital Image Processing

Binning is another operation that accomplishes linear filtering of an image. It is a type of image pre-processing, which combines blocks of adjacent pixels throughout an image,

by summing or averaging their values. In our work, we will examine image binning by averaging the values of 2×2 configurations in which 4 pixels are combined to produce 1 single pixel at a time.

The biggest downside of this technique is your resolution is effectively divided by four when binning an image. That means a binned shot from a 48MP camera is actually 12MP. However, this downside can be turned into an advantage for many real-world applications that prioritize performance and power efficiency over accuracy. For instance, applications related to autonomous driving and space exploration, that require ultra low-latency processing and big-data managing and transferring for artificial intelligence algorithms on the edge, usually sacrifice high image resolution from advanced image sensors, for real-time performance.

In order to apply averaging binning into images, we will use 2×2 kernels, consisting of ones, which are convolved with the original input and stride equal to 2 and then divided by 4. Therefore, for any given image with dimensions NxN, the output will be $\frac{N}{2}$x$\frac{N}{2}$ and have pixel ratio 4:1.

1	1
1	1

Figure 4.26: Binning kernel

In Figure 4.27 we can see an example of image binning, in which we convolve the binning kernel over each 2x2 region of the input image and then dividing by 4. The calculation is presented on the right hand side, as a 3x3 array.

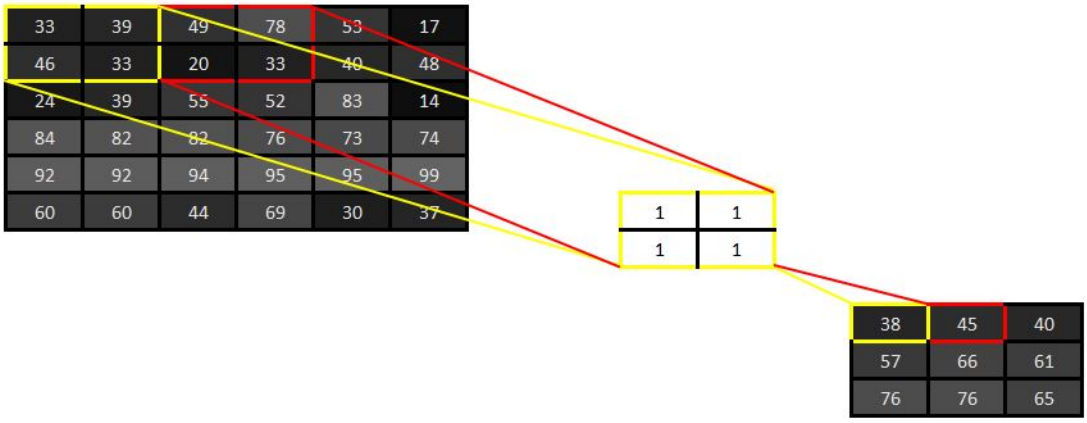


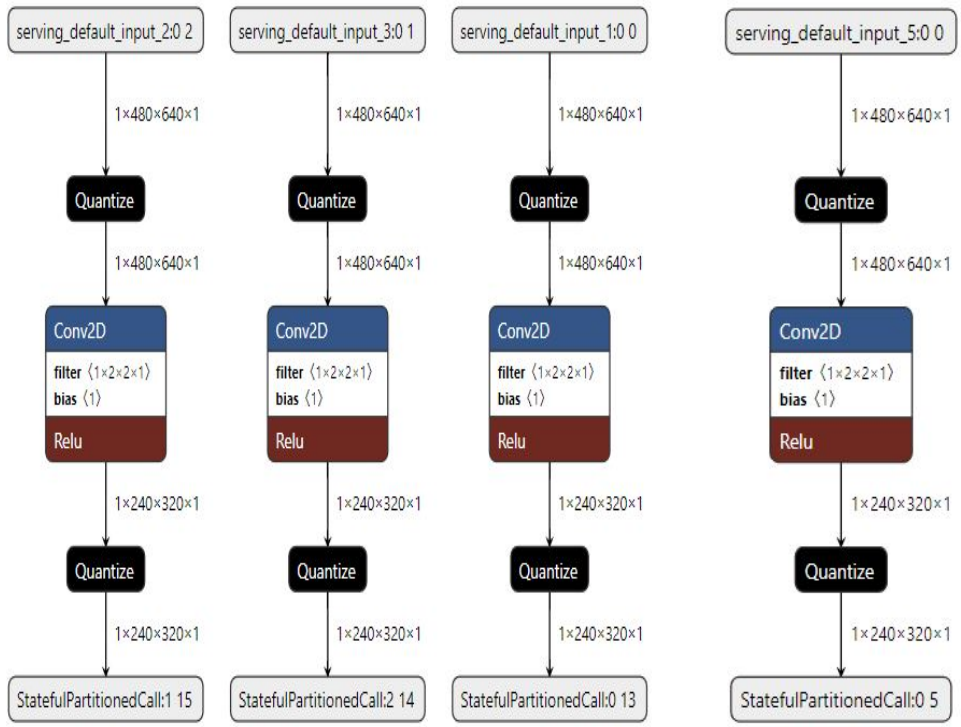
Figure 4.27: Example of 2x2 averaging binning on image

4.3.2.1 Implementation on Edge TPU board

We will follow the same steps as we did with the sobel operator, making only minor modifications, as follows:

1. Maintain input image as RGB instead of grayscale.
2. Binning kernel is a 2x2 array of ones.
3. RGB images have 3 channels so we need to convolve each one of them with the kernel. There are two alternatives:
   - Build a model with 3 convolutional layers and process the 3 inputs at once.
   - Build a model with 1 convolutional layer, like the sobel operator, and run 3 inferences, one for every input.
4. In any case, all 3 outputs combined will form the binned image.

The structure of the Binning network as TFlite model is depicted in Figure 4.28. As we can notice, in the first case the network consists of three 2-dimensional convolutional layers, executed on parallel, taking one input and giving one output. On the other hand, we have a single 2-dimensional convolutional layer, which however is inferenced three times, one for each channel of the image.



(a) 3 Conv2d layers - 1 inference **(b)** 1 Conv2d layer - 3 inferences

Figure 4.28: Binning TFlite network

Figure 4.29 illustrates all the stages of processing that an input image undergoes in order to be binned. At first, the test image is split into its respective Red, Green, and Blue channels and each one of the channels is convolved with the 2x2 binning kernel. So at the end we get 3 channels at 4:1 pixel ratio over the input image. All 3 channels combined, form the final binned image.

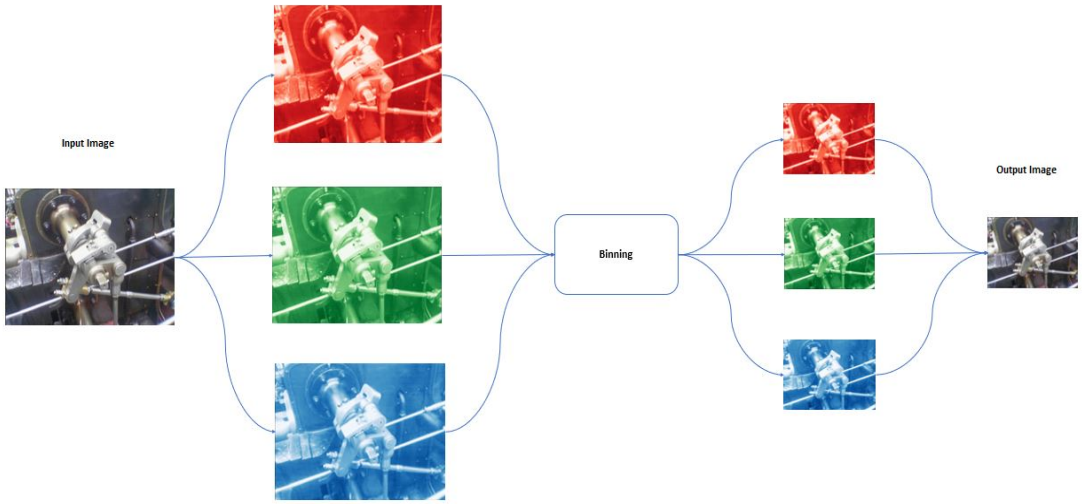


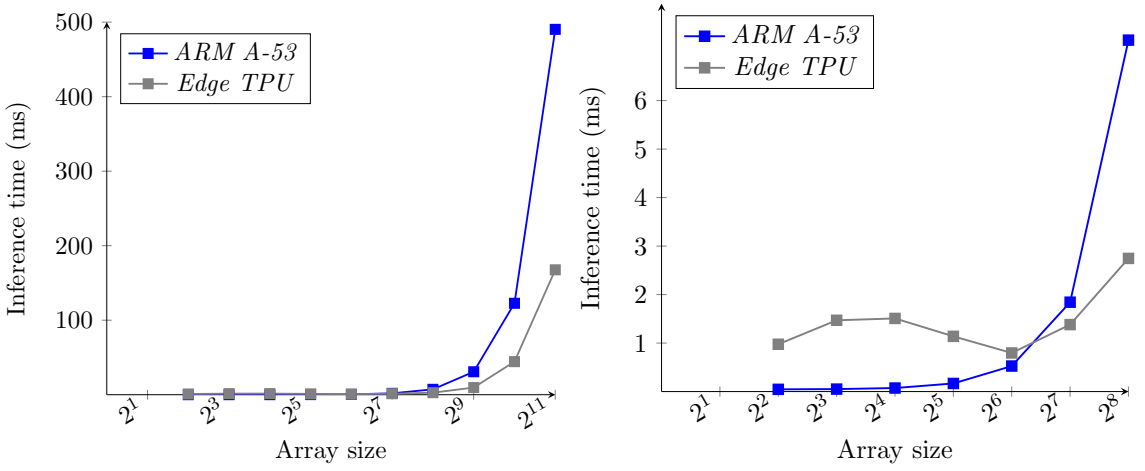
Figure 4.29: Binning on test image

After having presented two implementation of Image Binning on the Coral development board, we proceed on evaluate both alternatives in terms of performance acceleration.Table 4.20 presents all the necessary data we collected while conducting measurements on the development board for both cases. Our goal here is to examine how the input tensor size affects the overall inference time and whether executing 3 operations on parallel is beneficial in comparison to inferencing single operations three times. Our test includes array sizes both small and large, other theoretical and other indicative of real image sizes.

Table 4.20 displays side by side the experimental results of both approaches on the development board. At first, we can reach similar conclusions with those mentioned previously on the Sobel operator implementation, concerning the 64x64 systolic array and the significant acceleration achieved by the Edge TPU (Figure 4.30). Besides this, in this case we can also draw some conclusions about the influence of parallelizing multiple layers in one only inference. More specifically, Figure 4.31 help us realize that, even though for input dimensions larger than 128x128, the inference time is similar for both cases, for smaller arrays, parallelization inside the TPU results in up to 2 times faster inferencing. This is a logical consequence of the fact that running separate inferences for each one the 3 inputs requires 3 times more data transportation and communication between cores and memory. However, it is safe to argue that both approaches are satisfactory for accelerating high resolution image binning.

Table 4.20: Average inference time (ms)

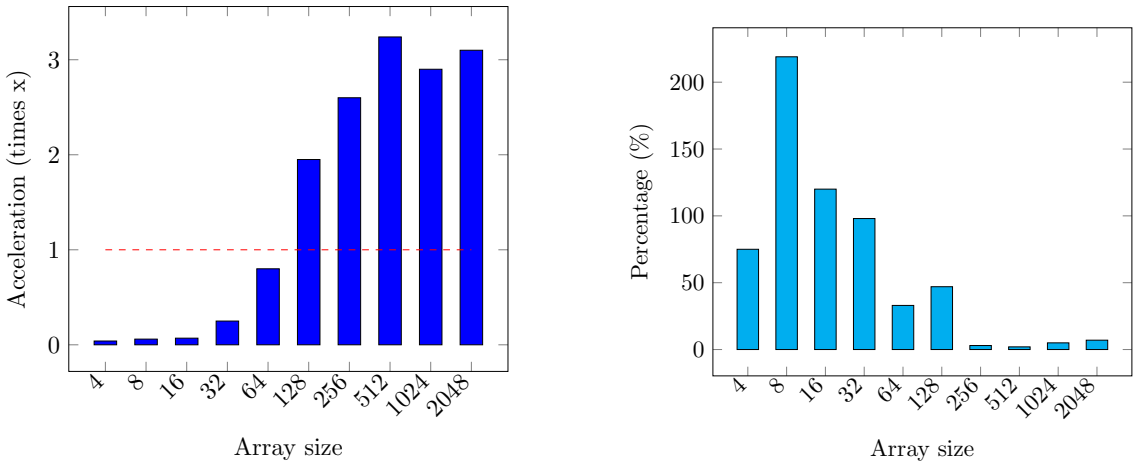
	1 layer - 3 inferences		3 layers - 1 inference	
inference	**ARM A-53**	**Edge TPU**	**ARM A-53**	**Edge TPU**
quant type	uint8	uint8	uint8	uint8
inputs	1	1	3	3
outputs	1	1	3	3
operations	1	1	3	3
quantizations	2	2	6	6
Array size				
4x4	0.046	0.976	0.023	0.557
8x8	0.051	1.47	0.028	0.461
16x16	0.073	1.51	0.05	0.685
32x32	0.167	1.14	0.141	0.574
64x64	0.526	0.8	0.481	0.597
128x128	1.84	**1.38**	1.83	**0.939**
256x256	7.25	**2.75**	7.3	**2.82**
512x512	30.8	**9.61**	30.5	**9.4**
1024x1204	122.7	**44.4**	122.4	**42.5**
2048x2048	490.3	**167.6**	488.1	**156.8**



(a) Total results

(b) Zoom in smaller arrays

Figure 4.30: Binning inference time



(a) Edge TPU speedup over ARM A-53

(b) 1 layer vs 3 layers

Figure 4.31: Binning comparisons

Chapter 5

Conclusion and Future Work

The purpose of this thesis was to explore the ability of neural network accelerators, like the Google Edge TPU, in order to perform general-purpose computing and to develop methodologies for this purpose. Neural network accelerators offer power efficiency orders-of-magnitude better than that of conventional vector processors, however they are not optimised for general tensor algebra. Therefore, with reverse engineering we developed a methodology that allows implementation of general purpose computing on the TPU accelerator. Using that methodology, we managed to both reveal important architectural characteristics of the Edge TPU and develop demo applications for novel image processing.

The experimental results revealed significant performance acceleration for the Google Edge TPU in comparison to the ARM A53 processor and other embedded devices. Overall, the Edge TPU achieves significant acceleration for computationally demanding workloads, like medium and large sized CNNs and MLPs or custom models dominated by matrix multiplications. Comparatively to the Jetson Nano GPU and the Intel Myriad X VPU, the Edge TPU achieves up to 5x better performance, while it can perform just as well as the Zynq ZCU104 FPGA. The matrix multiplication operations are improved up to 4x compared to the 8-bit quantized ARM execution and up to 7x for 32-bit floating point. Our results also show that, beyond small model sizes, the Edge TPU platform is always superior to the ARM A-53 achieving up to 100x for medium and up to 30x better performance for large CNNs. Moreover, for classic Digital Signal Processing (DSP) operations, such as the Sobel Edge Detector and Image Binning, the Edge TPU provides up to 6x better performance than ARM A-53. We further observed that the advantage of Edge TPU over ARM processor declines once the model size reaches the size of the on-chip memory of the Edge TPU platform. The final finding of our experiments, concerning general-purpose computing, is that the TPU does not perform well on element-wise operations or models that require accessing memory in a sparse manner.

Our future work will focus on creating frameworks for accelerating the software development of computationally intensive applications on Tensor Processing Units (TPUs) with the aim of integrating them to heterogeneous systems for future space and avionics applications. In this context, we are working in collaboration with the European Space

Agency (ESA) for deploying the Edge TPU accelerator for future space missions. We will also explore and verify the implementation of demanding applications on multiple Edge TPUs with pipelining.

Bibliography

[1] Coral, "Edge TPU performance benchmarks." [Online]. Available: https://coral.ai/docs/edgetpu/benchmarks/

[2] Nvidia, "Deep Learning Inference Benchmarks." [Online]. Available: https://developer.nvidia.com/blog/jetson-nano-ai-computing/

[3] Intel, "Performance for Deep Learning Inference." [Online]. Available: https://docs.openvino.ai/2020.2/_docs_performance_benchmarks.html#intel-ncs2

[4] Xilinx, "Model Zoo deployed on Xilinx hardware with Vitis AI." [Online]. Available: https://github.com/Xilinx/Vitis-AI/tree/master/model_zoo

[5] Qengineering, "Google coral edge tpu explained in depth." [Online]. Available: https://qengineering.eu/google-corals-tpu-explained.html

[6] Nvidia, "Jetson nano developer kit technical specifications." [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[7] Intel, "Intel Neural Compute Stick 2 with Movidius Myriad X VPU." [Online]. Available: https://www.intel.com/content/dam/develop/public/us/en/documents/ncs2-data-sheet.pdf

[8] Xilinx, "Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit." [Online]. Available: https://www.xilinx.com/products/boards-and-kits/zcu104.html

[9] "Run inference on the Edge TPU with Python." [Online]. Available: https://coral.ai/docs/edgetpu/inference/

[10] V. Leon, C. Bezaitis, G. Lentaris, D. Soudris, D. Reisis, E.-A. Papatheofanous, A. Kyriakos, A. Dunne, A. Samuelsson, and D. Steenari, "FPGA & VPU Co-Processing in Space Applications: Development and Testing with DSP/AI Benchmarks," in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2021, pp. 1–5.

[11] E. Petrongonas, V. Leon, G. Lentaris, and D. Soudris, "ParalOS: A Scheduling & Memory Management Framework for Heterogeneous VPUs," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 221–228.

[12] G. Lentaris, G. Chatzitsompanis, V. Leon, K. Pekmestzi, and D. Soudris, "Combining Arithmetic Approximation Techniques for Improved CNN Circuit Design," in *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2020, pp. 1–4.

[13] Kaggle, "Ships in satellite imagery," 2018 (acces. Sep 2021. [Online]. Available: https://www.kaggle.com/datasets/rhammell/ships-in-satellite-imagery

[14] A. Kyriakos, E.-A. Papatheofanous, C. Bezaitis, and D. Reisis, "Resources and power efficient fpga accelerators for real-time image classification," *Journal of Imaging*, vol. 8, no. 4, 2022.

[15] Xilinx, "Zynq-7000 SoC Data Sheet." [Online]. Available: https://www.mouser.com/datasheet/2/903/ds190-Zynq-7000-Overview-1595492.pdf

[16] Y. Bengio, "Learning deep architectures for ai," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[17] S.S.Haykin, in *Neural networks and learning machines*. Pearson, 2016.

[18] W. J. Song, "Chapter two - hardware accelerator systems for embedded systems," in *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*, ser. Advances in Computers, S. Kim and G. C. Deka, Eds. Elsevier, 2021, vol. 122, pp. 23–49.

[19] M. Lofqvist and J. Cano, "Accelerating deep learning applications in space," in *AIAA/USU Conference on Small Satellites*, 2020, pp. 1–19.

[20] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, 2015.

[21] C. Marantos, N. Karavalakis, V. Leon, V. Tsoutsouras, K. Pekmestzi, and D. Soudris, "Efficient support vector machines implementation on Intel/Movidius Myriad 2," in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2018, pp. 1–4.

[22] S. Mouselinos, V. Leon, S. Xydis, D. Soudris, and K. Pekmestzi, "Tf2fpga: A framework for projecting and accelerating tensorflow cnns on fpga platforms," in *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2019, pp. 1–4.

[23] V. Leon, I. Stamoulias, G. Lentaris, D. Soudris, R. Domingo, M. Verdugo, D. Gonzalez-Arjona, D. Merodio Codinachs, and I. Conway, "Systematic Evaluation of the European NG-LARGE FPGA & EDA Tools for On-Board Processing," 06 2021.

[24] J. Goodwill *et al.*, "NASA SpaceCube Edge TPU SmallSat Card for Autonomous Operations and Onboard Science-Data Analysis," in *AIAA/USU Conf. on Small Satellites*, 2021, pp. 1–13.

[25] S. Hosseininoorbin, S. Layeghy, M. Sarhan, R. Jurdak, and M. Portmann, "Exploring Edge TPU for network intrusion detection in iot," *CoRR*, vol. abs/2103.16295, 2021.

[26] Y. Hui, J. Lien, and X. Lu, "Early experience in benchmarking edge ai processors with object detection workloads," in *Benchmarking, Measuring, and Optimizing*, W. Gao, J. Zhan, G. Fox, X. Lu, and D. Stanzione, Eds. Springer International Publishing, 2020, pp. 32–48.

[27] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019, pp. 1–8.

[28] V. Leon, G. Lentaris, E. Petrongonas, D. Soudris, G. Furano, A. Tavoularis, and D. Moloney, "Improving performance-power-programmability in space avionics with edge devices: VBN on Myriad2 SoC," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 3, pp. 1–23, 2021.

[29] G. Giuffrida, L. Fanucci, G. Meoni, M. Batič, L. Buckley, A. Dunne, C. Van Dijk, M. Esposito, J. Hefele, N. Vercruyssen, G. Furano, M. Pastena, and J. Aschbacher, "The Φ-Sat-1 mission: the first on-board deep neural network demonstrator for satellite earth observation," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 60, pp. 1–14, 2022.

[30] V. Leon, K. Pekmestzi, and D. Soudris, "Systematic Embedded Development and Implementation Techniques on Intel Myriad VPUs," in *30th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022, pp. 1–2.

[31] J. Hochstetler, R. Padidela, Q. Chen, Q. Yang, and S. Fu, "Embedded deep learning for vehicular edge computing," in *IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 341–343.

[32] V. Leon, G. Lentaris, D. Soudris, S. Vellas, and M. Bernou, "Towards Employing FPGA and ASIP Acceleration to Enable Onboard AI/ML in Space Applications," in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022, pp. 1–4.

[33] G. Lentaris, I. Stratakos, I. Stamoulias, D. Soudris, M. Lourakis, and X. Zabulis, "High-performance vision-based navigation on soc fpga for spacecraft proximity operations," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 4, pp. 1188–1202, 2020.

[34] A. Yazdanbakhsh, K. Seshadri, B. Akin, J. Laudon, and R. Narayanaswami, "An evaluation of edge tpu accelerators for convolutional neural networks," 2021.

[35] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, sep 2020.

[36] Google, "Coral dev board datasheet," 2020. [Online]. Available: https://coral.ai/static/files/Coral-Dev-Board-datasheet.pdf

[37] "In-datacenter performance analysis of a tensor processing unit," vol. 45, no. 2. Association for Computing Machinery, jun 2017, p. 1–12.

[38] A. Shahid and M. Mushtaq, "A survey comparing specialized hardware and evolution in tpus for neural networks," in *2020 IEEE 23rd International Multitopic Conference (INMIC)*, 2020, pp. 1–6.

[39] Z. Pan and P. Mishra, "Hardware acceleration of explainable machine learning using tensor processing units," 2021.

[40] D. Friedman, "Sc: Hardware approaches to machine learning and inference," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, 2018, pp. 533–534.

[41] "Google coral." [Online]. Available: https://coral.ai/products

[42] Google, "Coral dev board mini datasheet," 2020. [Online]. Available: https://coral.ai/static/files/Coral-Dev-Board-Mini-datasheet.pdf

[43] G. Aurelien, in *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems.* OReilly, 2019.

[44] "About keras." [Online]. Available: https://keras.io/about/

[45] "Tensorflow graph execution." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/Graph

[46] "TensorFlow Lite." [Online]. Available: https://www.tensorflow.org/lite/guide

[47] "Post-training quantization." [Online]. Available: https://www.tensorflow.org/lite/performance/post_training_quantization

[48] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," 2017.

[49] "Edge TPU Compiler." [Online]. Available: https://coral.ai/docs/edgetpu/compiler/

[50] T. Habite, O. Abdeljaber, and A. Olsson, "Automatic detection of annual rings and pith location along norway spruce timber boards using conditional adversarial networks," *Wood Science and Technology*, vol. 55, pp. 1–28, 03 2021.

[51] A. HajiRassouliha, A. J. Taberner, M. P. Nash, and P. M. Nielsen, "Suitability of recent hardware accelerators (dsps, fpgas, and gpus) for computer vision and image processing algorithms," *Signal Processing: Image Communication*, vol. 68, pp. 101–119, 2018.

[52] D. Demirović, E. Skejić, and A. Šerifović–Trbalić, "Performance of some image processing algorithms in tensorflow," in *2018 25th International Conference on Systems, Signals and Image Processing (IWSSIP)*, 2018, pp. 1–4.

[53] M. Ansari, D. Kurchaniya, and M. Dixit, "A comprehensive analysis of image edge detection techniques," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 12, pp. 1–12, 11 2017.

[54] W. Gao, X. Zhang, L. Yang, and H. Liu, "An improved sobel edge detection," in *2010 3rd International Conference on Computer Science and Information Technology*, vol. 5, 2010, pp. 67–71.