Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

τομεας τεχνολογιας πληροφορικης και υπολογιστων

Εργαστηριο Μικροϋπολογιστων και Ψηφιακων Συστηματων

LUMAX: A LUT-Based Mixed-Precision Accelerator for LLM Inference on the Edge

A LUT-Based Mixed-Precision GeMM Accelerator with RISC-V RoCC for Efficient Low-Bit LLM Inference

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Κωνσταντίνου Ιωάννου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2025

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

LUMAX: A LUT-Based Mixed-Precision Accelerator for LLM Inference on the Edge

A LUT-Based Mixed-Precision GeMM Accelerator with RISC-V RoCC for Efficient Low-Bit LLM Inference

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Κωνσταντίνου Ιωάννου

Επιβλέπων: Δημήτριος Σούντρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 29η Σεπτεμβρίου, 2025.

.....	.....	.....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξύδης
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2025

.....

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

στην οικογένεια μου

Περίληψη

Τα τελευταία χρόνια, η ραγδαία ανάπτυξη των μεγάλων γλωσσικών μοντέλων (LLMs) έχει αυξήσει τη ζήτηση για αποδοτική εκτέλεση τόσο σε υποδομές κέντρων δεδομένων όσο και σε πλατφόρμες περιορισμένων πόρων. Παρόλο που η κβαντοποίηση (quantization) μειώνει το υπολογιστικό κόστος και την απαιτούμενη μνήμη, οι μικτής ακρίβειας πράξεις, όπου οι ενεργοποιήσεις διατηρούνται σε υψηλότερη ακρίβεια ενώ τα βάρη ποσοτικοποιούνται σε μικρότερο εύρος bit, παραμένουν αναποτελεσματικές στο γενικού σκοπού υλικό. Οι μέθοδοι που βασίζονται σε Πίνακες Αναζήτησης (LUT) προσφέρουν μια πολλά υποσχόμενη εναλλακτική λύση· ωστόσο, η επίτευξη της βέλτιστης ισορροπίας μεταξύ χρήσης μνήμης, ευελιξίας και προσαρμοστικότητας στο φόρτο εργασίας παραμένει πρόκληση. Προτείνουμε το **LUMAX**, έναν πλήρως ενσωματωμένο επιταχυντή μικτής ακρίβειας πολλαπλασιασμού πινάκων με βάση LUT, για ενεργειακά αποδοτική εκτέλεση μεγάλων γλωσσικών μοντέλων. Το LUMAX διαθέτει επαναδιαμορφώσιμο υλικό, επιτρέποντας την αποδοτική υποστήριξη διαφορετικών εύρων bit για ενεργοποιήσεις και βάρη. Για τη μείωση του κόστους των LUT, χρησιμοποιούμε LUT τεταρτημορίου μεγέθους ($\frac{1}{4}$-LUT) με αποδοτική δεικτοδότηση και συμπίεση δεδομένων, ελαχιστοποιώντας την αποθήκευση και τη μεταφορά δεδομένων. Το LUMAX έχει υλοποιηθεί ως συνεπεξεργαστής RocketChip (RoCC), επιτρέποντας έτσι απρόσκοπτη ενσωμάτωση με πυρήνες RISC-V. Επεκτείνοντας βασικές ιδέες από πρόσφατα σχέδια βασισμένα σε LUT και συνδυάζοντάς τες με πλήρη ενσωμάτωση στον επεξεργαστή και επαναδιαμορφώσιμο υλικό, το LUMAX προσφέρει έναν ευέλικτο, ενεργειακά αποδοτικό επιταχυντή για ποσοτικοποιημένη εκτέλεση LLM, συνδυάζοντας προσαρμοστικότητα υλικού, χρηστικότητα λογισμικού και αρχιτεκτονική αποδοτικότητα. **Αποτελέσματα αξιολόγησης** δείχνουν ότι το LUMAX, πρωτοτυποποιημένο σε FPGA ZCU106, μειώνει τη χρήση LUT και DSP έως και 33% και 96% αντίστοιχα, επιτυγχάνει με 79% λιγότερους κύκλους και προσφέρει έως και 4.7× επιτάχυνση στο LLaMA2, με έως και 70% βελτίωση στην ενεργειακή αποδοτικότητα σε σύγκριση με προηγούμενους επιταχυντές πολλαπλασιασμού πινάκων όπως το Gemmini.

Λέξεις Κλειδιά — Πίνακας Αναζήτησης (LUT), Μικτής Ακρίβειας Πράξεις, Επιταχυντής, Πολλαπλασιαμός Πινάκων, Χαμηλής Ακρίβειας Μεγάλα Γλωσσικά Μοντέλα.

Abstract

In recent years, the rapid growth of large language models (LLMs) has increased demand for efficient inference on both datacenter and edge platforms. While quantization reduces computation and memory costs, mixed-precision operations, where activations remain in higher precision while weights are quantized to lower bitwidths, remain inefficient on general-purpose hardware. Lookup Table (LUT)-based methods offer a promising alternative, yet achieving an optimal balance of memory usage, flexibility, and workload adaptability remains challenging. We propose **LUMAX**, a fully integrated LUT-based mixed-precision GeMM accelerator for energy-efficient LLM inference. LUMAX features a reconfigurable hardware design, allowing for efficient support of different activation and weight bitwidths. To reduce LUT overhead, we employ a quarter-size LUT ($\frac{1}{4}$-LUT) with efficient indexing and data packaging, minimizing storage and data transfer. LUMAX has been implemented as a tightly coupled RocketChip Co-processor (RoCC), thus enabling seamless processor integration with RISC-V cores. By extending key ideas from recent LUT-based designs and combining them with full processor integration and reconfigurable hardware, LUMAX provides a flexible, power-efficient accelerator for quantized LLM inference, blending hardware adaptability, software usability, and architectural efficiency. **Evaluation results** show that LUMAX, prototyped on a ZCU106 FPGA, reduces LUT and DSP usage by up to 33% and 96%, achieves 79% fewer cycles, and delivers up to 4.7× speedup on LLaMA2, with up to 70% improved energy efficiency over prior GeMM accelerators such as Gemmini.

Key words — LUT, Mixed-Precision, Accelerator, GeMM, Low-bit LLM.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον κύριο Σωτήριο Ξύδη για την καθοδήγησή του κατά την εκπόνηση της διπλωματικής μου εργασίας, καθώς και τους υποψήφιους διδάκτορες Γιώργο Αλεξάνδρη και Γιώργο Αναγνωστόπουλο για τη συνεχή υποστήριξη και συνεργασία.

Ιδιαίτερες ευχαριστίες απευθύνω στον κύριο Δημήτρη Σούντρη για το δημιουργικό περιβάλλον του εργαστηρίου, που συνέβαλε καθοριστικά στην ολοκλήρωση της μελέτης.

Πάνω απ' όλα, θα ήθελα να ευχαριστήσω τους γονείς μου και την αδερφή μου για τη στήριξη και την υπομονή τους όλα αυτά τα χρόνια.

Κωνσταντίνος Ιωάννου
Σεπτέμβριος 2025

Contents

Figure List

Table List

Chapter 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Στη σημερινή εποχή, τα Βαθιά Νευρωνικά Δίκτυα (DNNs) έχουν γίνει αναπόσπαστο μέρος πολλών αναδυόμενων ερευνητικών δραστηριοτήτων, προκαλώντας τους σύγχρονους αρχιτέκτονες συστημάτων να προσαρμόζονται συνεχώς και να βελτιστοποιούν το σχεδιασμό για ταχύτερες και πιο ακριβείς λύσεις [1, 2, 3, 4, 5, 6]. Μετά αποτέλεσμα, οι σύγχρονες ερευνητικές προσεγγίσεις διερευνούν λύσεις που διαφέρουν από το απλό μοντέλο εκτέλεσης σε CPU [7], επικεντρώνοντας είτε σε λύσεις βασισμένες σε GPU είτε σε ακόμη πιο εξειδικευμένους επιταχυντές, π.χ. Tensor Processing Units (TPUs) ή Neural Processing Units (NPUs) [8, 9, 10, 7].

Οι σύγχρονες edge αναπτύξεις εφαρμογών τεχνητής νοημοσύνης βασισμένων σε LLMs [11, 12, 13, 14] ωθούν ακόμη περισσότερο τα όρια της ενεργειακής αποδοτικότητας. Μέθοδοι όπως η κβαντοποίηση νευρωνικών δικτύων (NN quantization) [15, 16, 17, 10, 18] έχουν εμφανιστεί για να αναπαριστούν τα βάρη και/ή τις εισόδους ορισμένων επιπέδων με λιγότερα bits από την πλήρη αναπαράσταση κινητής υποδιαστολής, εξοικονομώντας τόσο ενέργεια, όσο και χώρο και υπολογιστικούς πόρους.

Η εξειδίκευση υλικού μέσω επιθετικών τεχνικών σχεδίασης όπως ο πολλαπλασιασμός βασισμένος σε LUT [19, 20, 21] χρησιμοποιείται για περαιτέρω βελτιστοποίηση των επιταχυντών όσον αφορά τα χαρακτηριστικά Απόδοση-Ενέργεια-Χώρος (Performance-Power-Area, PPA). Έτσι, παρατηρείται αυξανόμενο ενδιαφέρον για επιταχυντές ειδικού τομέα που εκμεταλλεύονται μειωμένη αριθμητική ακρίβεια στις αναπαραστάσεις δεδομένων. Τέτοιες σχεδιάσεις επιτυγχάνουν σημαντικές βελτιώσεις στην απόδοση, την ενεργειακή αποδοτικότητα και την καλύτερη αξιοποίηση των πόρων υλικού [1, 15, 22, 23, 24, 25, 26, 27, 28]. Αντίθετα, οι γενικού σκοπού επιταχυντές που υποστηρίζουν κυρίως υπολογισμούς υψηλής ακρίβειας δεν είναι κατάλληλοι για μεγάλα γλωσσικά μοντέλα (LLMs) που βασίζονται σε inferencing με χαμηλή ακρίβεια bit [21, 29].

Τα σύγχρονα LLMs, που συνήθως συνδυάζουν ενεργοποιήσεις υψηλής ακρίβειας με βάρη χαμηλής ακρίβειας, έχουν οδηγήσει στο σχεδιασμό νέων αρχιτεκτονικών επιταχυντών. Αυτοί οι επιταχυντές στοχεύουν στην αποτελεσματική υποστήριξη λειτουργιών **mixed-precision** ενώ μεγιστοποιούν την επαναχρησιμοποίηση δεδομένων [1, 15, 22].

Μεταξύ των πιο υποσχόμενων τεχνικών σχεδίασης είναι οι **LUT-based μέθοδοι**. Αυτό συμβαίνει κυρίως επειδή, όταν κάποια μοντέλα προσπαθούν να λειτουργήσουν σε χαμηλότερα bitwidths, τα κέρδη από τη χρήση αναζήτησης μέσω LUT είναι σημαντικά σε σύγκριση με την εκτέλεση του πολλαπλασιασμού. Σε αυτή την προσέγγιση, οι επιταχυντές είτε αποθηκεύουν στους πίνακες όλο τον χώρο προϊόντων για ενεργοποιήσεις και βάρη χαμηλής ακρίβειας [16], είτε διατηρούν ένα μειωμένο σύνολο τιμών για ένα συγκεκριμένο παράθυρο ενεργοποίησης εκτελώντας προ-υπολογισμό πίνακα on-the-fly για κάθε μονάδα LUT. Τα αντίστοιχα βάρη ανακτώνται και συνδυάζονται για να παραχθούν τα τελικά αποτελέσματα. Αυτή η μεθοδολογία είναι ιδιαίτερα αποτελεσματική μόνο όταν τα βάρη είναι κβαντοποιημένα σε χαμηλότερη

ακρίβεια, ενώ οι ενεργοποιήσεις παραμένουν υψηλής ακρίβειας π.χ. 8–16 bits [19, 21].

Το κύριο μειονέκτημα προηγούμενων υλοποιήσεων [19, 21] είναι η περιορισμένη κλιμάκωση των κερδών απόδοσης καθώς αυξάνεται η ακρίβεια των bit των βαρών λόγω της χρήσης bit-serial αρχιτεκτονικής. Σε αυτή την εργασία, παρουσιάζουμε το LUMAX, έναν LUT-based GeMM επιταχυντή που υποστηρίζει mixed-precision εισόδους και βάρη, στοχεύοντας στον τομέα των LLM και παρέχοντας μια χαμηλής κατανάλωσης ενέργειας και ευέλικτη λύση για inferencing σύγχρονων νευρωνικών δικτύων στην edge.

Σε αντίθεση με προηγούμενους LUT-based επιταχυντές, οι οποίοι προϋπολογίζουν και αποθηκεύουν συνδυασμούς ενεργοποιήσεων και στη συνέχεια βασίζονται σε bit-serial λογική και shift–accumulate βήματα για την ανακατασκευή προϊόντων, το LUMAX υιοθετεί μια θεμελιωδώς διαφορετική στρατηγική αποθήκευσης. Για κάθε τιμή ενεργοποίησης, το LUMAX αποθηκεύει απευθείας όλα τα πιθανά προϊόντα σύμφωνα με την ακρίβεια του βάρους, οργανώνοντάς τα και ομαδοποιώντας τα σε MEMs. Κατά την εκτέλεση, το βάρος λειτουργεί απλώς ως δείκτης για την επιλογή του αντίστοιχου προϊόντος σε έναν μόνο κύκλο, επιτρέποντας ταυτόχρονη συσσώρευση χωρίς την ανάγκη επαναλαμβανόμενων bit-serial λειτουργιών. Αυτός ο σχεδιασμός μειώνει όχι μόνο την καθυστέρηση αλλά και βελτιώνει την κλιμάκωση για mixed-precision workloads.

Επιπλέον, αυτό το πλαίσιο επιταχυντή είναι ικανό να αναδιαμορφώνει δυναμικά την ακρίβεια εισόδων και βαρών, παρέχοντας ένα ευέλικτο περιβάλλον, που μπορεί να καλύψει πολλές από τις σύγχρονες καινοτόμες κβαντοποιημένες εφαρμογές LLM. Ο επιταχυντής υλοποιείται ως RocketChip Co-processor (RoCC), χρησιμοποιώντας έναν RISC-V πυρήνα ως host processor. Επιπλέον, η μέθοδος πολλαπλασιασμού βασισμένη σε LUT βελτιστοποιεί περαιτέρω την ενεργειακή αποδοτικότητα του επιταχυντή εξερευνώντας την αναζήτηση ορισμένων χαρακτηριστικών βαρών/εισόδων αντί για τον υπολογισμό τους. Ειδικές βελτιστοποιήσεις αρχιτεκτονικής είναι επίσης ενσωματωμένες στο πλαίσιο επιτάχυνσης, όπως προανάκτηση βαρών και συμπίεση δεδομένων.

Ένα SoC βασισμένο στο LUMAX πρωτοτυπήθηκε χρησιμοποιώντας το Chipyard [30], χαρτογραφημένο στο FPGA ZCU106 [31]. Εξετάστηκαν διαφορετικές μικρο-αρχιτεκτονικές διαμορφώσεις του LUMAX, δείχνοντας μειώσεις στη χρήση LUT και DSP έως 33% και 96% αντίστοιχα, σε σύγκριση με τον ισοπόρο επιταχυντή TPU-based SoC βασισμένο στην αρχιτεκτονική Gemini [2]. Χρησιμοποιώντας το δίκτυο LLaMA2 [32] ως εφαρμογή οδήγησης, επιτεύχθηκαν έως και 4.7× μείωση κύκλων και 70% βελτίωση στην ενεργειακή αποδοτικότητα σε σύγκριση με προηγούμενους GeMM επιταχυντές.

1.2 Σχετική Βιβλιογραφία

1.2.1 Αφιερωμένοι Επιταχυντές για Κβαντισμένα Νευρωνικά Δίκτυα

Gemmini: Αρχιτεκτονική Συστολικού Πίνακα για GEMM

Οι συστολικοί πίνακες είναι μια δημοφιλής αρχιτεκτονική για την επιτάχυνση του πολλαπλασιασμού πινάκων, βασικής πράξης στα βαθιά νευρωνικά δίκτυα (DNNs). Αποτελούνται από πλέγμα επεξεργαστικών στοιχείων που λειτουργούν παράλληλα και συγχρονισμένα, μεταφέροντας δεδομένα σωληνωτά. Η προσέγγιση αυτή βελτιστοποιεί τη χρήση πόρων και μειώνει την ανάγκη πρόσβασης σε μνήμη εκτός τσιπ, αυξάνοντας την αποδοτικότητα και απόδοση.

Η χρήση συστολικών πινάκων είναι ιδανική για τις επαναλαμβανόμενες και παραλληλοποιήσιμες πράξεις πολλαπλασιασμού πινάκων στα DNNs, με χαμηλή καθυστέρηση κατά τις λειτουργίες πολλαπλασιασμούαθροίσματος.

Το Gemmini [2] είναι ένας επιταχυντής ανοικτού κώδικα που βασίζεται σε σιστολικό πίνακα, μέσα στο οικοσύστημα Chipyard. Υποστηρίζει λειτουργίες Output Stationary (OS) και Weight Stationary (WS), που καθορίζουν πώς αποθηκεύονται τα δεδομένα στα στοιχεία κατά τις πράξεις. Στην OS λειτουργία, οι τιμές εξόδου παραμένουν σταθερές, ενώ στη WS τα βάρη είναι σταθερά, μειώνοντας την κίνηση δεδομένων.

Το Gemmini υποστηρίζει κβαντισμένα τύπους int και uint, καθώς και κινητής υποδιαστολής fp32, καλύπτοντας ευρύ φάσμα εφαρμογών από κβαντισμένα DNN έως πλήρους ακρίβειας μοντέλα. Ιδιαίτερα, η υποστήριξη int8 το καθιστά αποδοτικό για γρήγορη και ενεργειακά αποδοτική εκτέλεση.

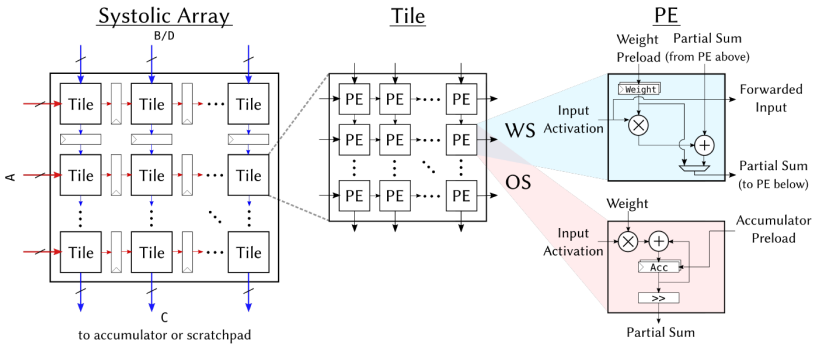


Figure 1.2.1: Σιστολικός Πίνακας Gemmini.

Είναι ευέλικτο, επιτρέποντας εναλλαγή μεταξύ OS και WS dataflows σε πραγματικό χρόνο και παραμετροποιήσιμη ακρίβεια αριθμών (`int8`, `int16`, `int32` και τα unsigned αντίστοιχα, καθώς και προαιρετικά float). Επιπλέον, υποστηρίζει βασικές λειτουργίες βαθιάς μάθησης όπως ReLU, ReLU6, max pooling, average pooling, και εγγενή υποστήριξη για μετατροπή πινάκων/διανυσμάτων μέσω υλικού. Η στενή ενσωμάτωση με το σύνολο εντολών RISC-V μέσω προσαρμοσμένων εντολών διευκολύνει τη συνεργασία λογισμικού-υλικού.

Carat: Αρχιτεκτονική Επιταχυντή με Πολλαπλασιασμούς χωρίς Χρήση Πολλαπλασιαστών

Το Carat [22] είναι μια καινοτόμος αρχιτεκτονική επιταχυντή σχεδιασμένη ειδικά για λειτουργίες γενικού πολλαπλασιασμού πινάκων (GEMM). Εισάγει μια προσέγγιση χωρίς πολλαπλασιαστές που μειώνει σημαντικά την κατανάλωση ενέργειας και την πολυπλοκότητα υλικού, εξαλείφοντας τις παραδοσιακές μονάδες πολλαπλασιασμού. Αντίθετα, το Carat μετατρέπει τους πολλαπλασιασμούς σε προσθετικές λειτουργίες χρησιμοποιώντας μια νέα τεχνική γνωστή ως *parallelism επιπέδου τιμών* (VLP).

Η βασική ιδέα πίσω από το VLP είναι η αξιοποίηση της πλεονασματικότητας στις τιμές εισόδου — ιδιαίτερα συχνή σε μορφές χαμηλής ακρίβειας όπως `INT4`, `FP8` ή `BF16`. Αντί να εκτελείται ένας ξεχωριστός πολλαπλασιασμός για κάθε ζεύγος τελεστέων, το Carat υπολογίζει κάθε μοναδικό γινόμενο μόνο μία φορά. Τα επαναλαμβανόμενα τελεστέα συνδέονται στη συνέχεια με αυτά τα γινόμενα μέσω μιας μορφής *χρονικής κωδικοποίησης*, όπου σήματα χρονισμού υποδεικνύουν πότε ένα γινόμενο που έχει ήδη υπολογιστεί πρέπει να επαναχρησιμοποιηθεί. Για παράδειγμα, σε έναν πίνακα εισόδων `INT4`, μπορεί να υπάρχουν μόνο 16 μοναδικά γινόμενα ανάμεσα σε χιλιάδες πολλαπλασιασμούς, οδηγώντας σε σημαντική εξοικονόμηση υπολογισμού.

Αρχιτεκτονικά, το Carat υιοθετεί μια δομή εμπνευσμένη από τους σιστολικούς πίνακες, όπου τα Επεξεργαστικά Στοιχεία (Processing Elements - PEs) είναι διατεταγμένα σε ένα δισδιάστατο πλέγμα πλακιδίων. Αυτά τα πλακίδια λειτουργούν πάνω σε εισερχόμενους τανυστές και επικοινωνούν μέσω μηχανισμών μετάδοσης και συγχρονισμού που εξασφαλίζουν τη συνεπή επαναχρησιμοποίηση των μερικών αποτελεσμάτων. Η αρχιτεκτονική μπορεί να κλιμακωθεί σε πολλαπλούς κόμβους μέσω ενός πλέγματος Δικτύου επί Τσιπ (Network-on-Chip - NoC), καθιστώντας την κατάλληλη για εργασίες υψηλής απόδοσης και κατανεμημένα φορτία.

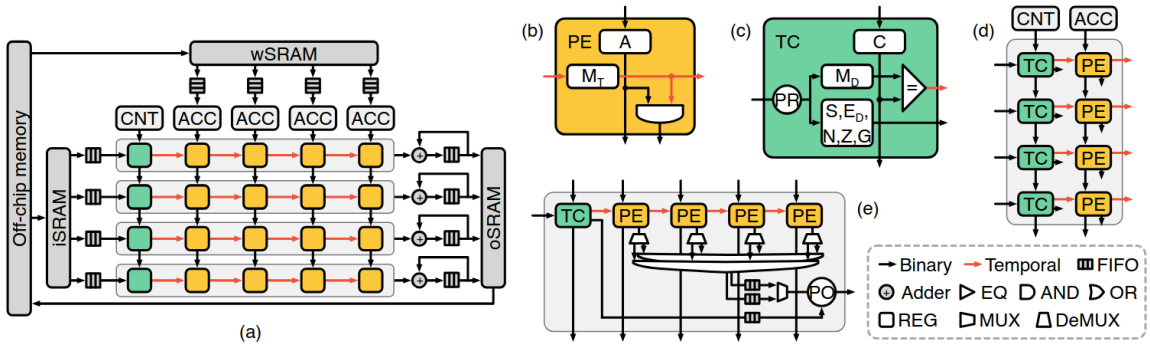


Figure 1.2.2: Επισκόπηση αρχιτεκτονικής επιταχυντή Carat.

Η αρχιτεκτονική είναι ιδιαίτερα βελτιστοποιημένη για εργασίες inference χαμηλής ακρίβειας και κβαντισμένες σε βαθιά νευρωνικά δίκτυα, όπου η ενεργειακή αποδοτικότητα και η απόδοση ανά Watt είναι κρίσιμες. Οι εμπειρικές αξιολογήσεις αναφέρουν βελτίωση της διαμέσου απόδοσης από 1.02 έως 3.2 φορές και κέρδη ενεργειακής αποδοτικότητας από 1.06 έως 4.3 φορές σε σύγκριση με συμβατικούς επιταχυντές σιστολικού πίνακα.

TATAA – Μεταβαλλόμενος Επιταχυντής για Transformers

Τα μοντέλα transformer, που αποτελούν τη βασική υποδομή των σύγχρονων Μεγάλων Γλωσσικών Μοντέλων (LLMs), απαιτούν τόσο γραμμικές όσο και μη γραμμικές λειτουργίες. Ενώ οι γραμμικές λειτουργίες κυριαρχούν στο συνολικό φόρτο υπολογισμού, οι μη γραμμικές λειτουργίες — όπως οι Softmax και κανονικοποίηση — απαιτούν υψηλότερη ακρίβεια. Ωστόσο, οι περισσότεροι επιταχυντές εστιάζουν σχεδόν αποκλειστικά σε λειτουργίες GEMM, παραμελώντας συχνά τη μη γραμμική επεξεργασία.

Το TATAA [1] αντιμετωπίζει αυτή τη διάσταση εισάγοντας μια αρχιτεκτονική επιταχυντή προγραμματιζόμενη σε πραγματικό χρόνο, ικανή να εκτελεί αποδοτικά και γραμμικές και μη γραμμικές λειτουργίες μέσα σε μοντέλα transformer. Οι βασικές του ικανότητες περιλαμβάνουν μια ευέλικτη, ενιαία αρχιτεκτονική υλικού σχεδιασμένη για αποδοτική υποστήριξη των υπολογισμών μοντέλων transformer, συμπεριλαμβανομένων τόσο των γραμμικών πολλαπλασιασμών πινάκων όσο και των σύνθετων μη γραμμικών λειτουργιών. Κεντρική καινοτομία αποτελεί η λειτουργία *Dual Mode Processing Unit* (DMPU), η οποία μπορεί να εναλλάσσεται ομαλά μεταξύ της λειτουργίας πολλαπλασιασμού πινάκων int8 (MatMul) και της λειτουργίας μη γραμμικής επεξεργασίας bfloat16. Αυτή η ευελιξία επιτυγχάνεται μέσω προσαρμοσμένου συνόλου εντολών (ISA) και πλαισίου μεταγλώττισης που μεταφράζει τις υψηλού επιπέδου λειτουργίες transformers σε βασικές υποστηριζόμενες λειτουργίες, επιτρέποντας ολοκληρωμένη εκτέλεση χωρίς ανάγκη εκ νέου εκπαίδευσης ή σημαντικών τροποποιήσεων υλικού.

Το σύστημα διαθέτει μια **προγραμματιζόμενη αρχιτεκτονική επεξεργασίας** που επιτρέπει ευέλικτη επαναχρησιμοποίηση των μονάδων υπολογισμού. Οι μονάδες επεξεργασίας είναι διασυνδεδεμένες ώστε να υποστηρίζουν τόσο ένα *σιστολικό πίνακα* υψηλής απόδοσης για GEMM όσο και μια *αρχιτεκτονική SIMD* για μη γραμμικές λειτουργίες. Υποστηρίζονται τα πρότυπα `int8` και `bfloat16`, μαζί με ενσωματωμένη **ποσοτικοποίηση επί του τσιπ**, εξαλείφοντας την ανάγκη για εξωτερική προεπεξεργασία.

Η μετάβαση μεταξύ των δύο λειτουργιών γίνεται δυναμικά σε πραγματικό χρόνο μέσω του *Mode MUX* και μιας ελαφριάς μονάδας ελέγχου. Αυτό επιτρέπει στον επιταχυντή να μετασχηματίζεται μεταξύ βελτιστοποιημένης για γραμμικές λειτουργίες σιστολικής εκτέλεσης και ευέλικτης SIMD εκτέλεσης, ανάλογα με τις απαιτήσεις του φορτίου εργασίας — προσφέροντας τόσο αποδοτικότητα όσο και ευελιξία στην επεξεργασία transformer.

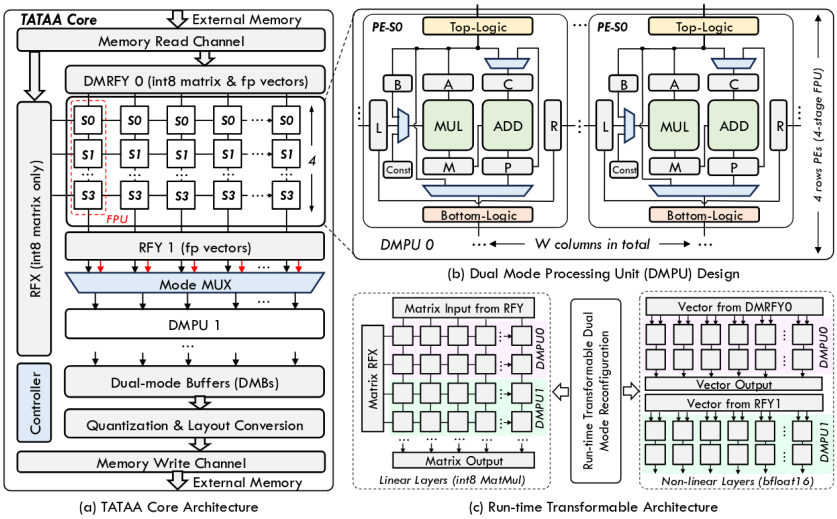


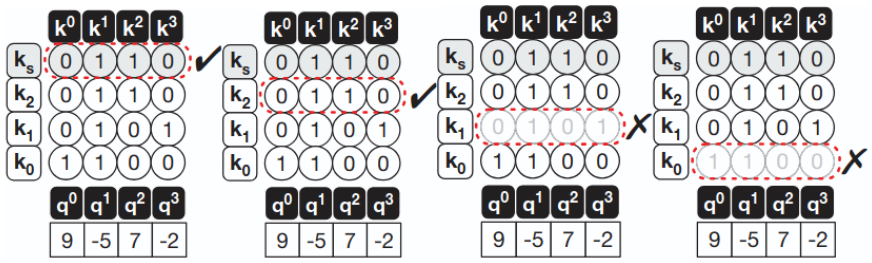
Figure 1.2.3: Αρχιτεκτονική υλικού του TATAA και μονάδα διπλής λειτουργίας επεξεργασίας

Στην αρχιτεκτονική TATAA, η υποστήριξη μη γραμμικών λειτουργιών στη λειτουργία SIMD με `bfloat16` επιτυγχάνεται μέσω προσεγγιστικών τεχνικών και ειδικού υλικού. Λειτουργίες όπως GELU, SoftMax και LayerNorm προσεγγίζονται με πολυωνυμικές ή λογισμικές συναρτήσεις χαμηλής τάξης, οι οποίες αποδίδονται αποδοτικά σε ακέραιες πράξεις. Αυτές οι προσεγγίσεις εκτελούνται μέσω αφοσιωμένων σωληνωτών σταδίων και λειτουργικών μονάδων, επιτρέποντας απρόσκοπτη ενσωμάτωση με γραμμικούς υπολογισμούς. Η χρήση υψηλής ακρίβειας `bfloat16` εξασφαλίζει την απαιτούμενη ακρίβεια, ενώ το μεταβαλλόμενο αριθμητικό σύστημα και ο παράλληλος υπολογισμός διατηρούν υψηλή απόδοση. Ο σχεδιασμός αυτός επιτρέπει την αποδοτική εκτέλεση σύνθετων μη γραμμικών λειτουργιών, επιταχύνοντας τα μοντέλα transformer χωρίς συμβιβασμούς στην ευελιξία.

LeOPard – Εκμάθηση Κατωφλίων με Βαθμίδες για Εκτέλεση Περικοπής κατά το Runtime

Στα μηχανισμούς προσοχής (attention) που χρησιμοποιούνται σε μεγάλα γλωσσικά μοντέλα (LLMs) κατά τη διάρκεια της εκτέλεσης (inference), μόνο ένα μικρό υποσύνολο των tokens παρουσιάζει υψηλή συσχέτιση με το token υπό προσοχή, και αυτό το υποσύνολο καθορίζεται κατά το χρόνο εκτέλεσης. Ως εκ τούτου, μεγάλο μέρος των υπολογισμών γίνεται ασήμαντο λόγω χαμηλών βαθμολογιών προσοχής. Στα στρώματα αυτοπροσοχής (self-attention), το κύριο υπολογιστικό βάρος συνδέεται με τον υπολογισμό του πίνακα βαθμολογιών, $\text{Scores} = Q \cdot K^T$, και τον υπολογισμό των τιμών προσοχής, $\text{Atts} = P \cdot V$.

Το LeOPard επιταχύνει τους υπολογισμούς προσοχής σε μοντέλα transformer με τη χρήση τεχνικών περικοπής κατωφλίων κατά το runtime βασισμένων σε πληροφορίες βαθμίδων (gradients) και πρόωρου τερματισμού υπολογισμού σε επίπεδο bit. Με αυτόν τον τρόπο, περιορίζει δυναμικά τα μη σημαντικά μέρη των βαθμολογιών προσοχής, μειώνοντας τον υπολογιστικό φόρτο, την κατανάλωση ενέργειας και τη λανθάνουσα χωρίς να θυσιάζει ακρίβεια. Η στρατηγική εκμάθησης κάθε στρώματος επιτρέπει την επιλογή διαφορετικών κατωφλίων προσοχής, ενώ ο πρόωρος τερματισμός σταματά περιττούς υπολογισμούς σε πρώιμα στάδια, αυξάνοντας την αποδοτικότητα. Οι πολλαπλασιασμοί πινάκων στο $\text{Atts} = P \cdot V$ γίνονται με τον περικομμένο πίνακα Softmax P, ενώ η συνάρτηση Softmax εφαρμόζεται μόνο στα διατηρημένα στοιχεία βαθμολογίας.



(a) Cycle = 1 (b) Cycle = 2 (c) Cycle = 3 (d) Cycle = 4

Cycle	$\mathcal{P}$ = Partial Sum	$\mathcal{M}$ = Conservative Margin	Early Termination? ($\mathcal{T}h$ = 5)
1	$\mathcal{P}_1 = 0$	$\mathcal{M}_1 = (9+5)(2^{-1}+2^{-2}+2^{-3}) = 12.25$	$\mathcal{P}_1 + \mathcal{M}_1 = 12.25 \geq 5$; ✗
2	$\mathcal{P}_2 = \mathcal{P}_1 + (5-7)2^{-1} = -1$	$\mathcal{M}_2 = (9+5)(2^{-2}+2^{-3}) = 5.25$	$\mathcal{P}_2 + \mathcal{M}_2 = 4.25 < 5$; ✓
3	$\mathcal{P}_3 = \mathcal{P}_2 + (5-2)2^{-2} = -0.25$	$\mathcal{M}_3 = (9+5)(2^{-3}) = 1.75$	$\mathcal{P}_3 + \mathcal{M}_3 = 1.5 < 5$;
4	$\mathcal{P}_4 = \mathcal{P}_3 + (9+5)2^{-3} = 1.5$	$\mathcal{M}_4 = 0$	$\mathcal{P}_4 + \mathcal{M}_4 = 1.5 < 5$;

Figure 1.2.4: Σύνοψη πρόωρου τερματισμού υπολογισμού για τη λειτουργία εσωτερικού γινομένου $Q \times K$. Στο παράδειγμα, το K παρουσιάζεται σε μορφή bit-serial, ενώ το Q σε ακέραιη ολοκλήρωση με πλήρη ακρίβεια. Οι στήλες (a-d) απεικονίζουν μεμονωμένα στοιχεία του διανύσματος K και οι γραμμές τα αντίστοιχα bits (MSB → LSB). Το σήμα K υποδηλώνει το bit πρόσημο. Για απλοποίηση, τα στοιχεία του K είναι κλιμακωμένα μεταξύ -1.0 και +1.0. Ο πίνακας εμφανίζει τις μερικές τιμές αθροίσματος μετά από κάθε κύκλο.

Το LeOPard υποστηρίζει κυρίως bit-serial επεξεργασία για τους υπολογισμούς των βαθμολογιών προσοχής ($Q \times K^T$) και των τιμών πολλαπλασιασμού ($\cdot V$). Το υλικό λειτουργεί με ακρίβεια σε επίπεδο bit (π.χ. μονάδες 12-bit σειριακές), επιτρέποντας λεπτομερή και πρόωρο τερματισμό των υπολογισμών βάσει των αποφάσεων περικοπής. Ο σχεδιασμός απευθύνεται σε μοντέλα transformer όπως BERT και Vision Transformers, που συνήθως χρησιμοποιούν 16-bit κινητής υποδιαστολής ή μορφές χαμηλότερης ακρίβειας. Ωστόσο, η αρχιτεκτονική του LeOPard είναι βελτιστοποιημένη για bit-serial αριθμητική ώστε να υλοποιεί αποδοτικά τόσο την περικοπή όσο και τον πρόωρο τερματισμό.

Στο πλαίσιο αυτό, οι ενεργοποιήσεις και τα βάρη αναπαρίστανται με μορφές ποσοτικοποίησης σταθερής υποδιαστολής. Η επιλογή της ακρίβειας 12-bit καθορίζεται από την συμβατότητα με το υλικό και την επιθυμητή ισορροπία μεταξύ ακρίβειας υπολογισμών και απόδοσης.

Το LeOPard εκτελεί αριθμητικές πράξεις σειριακά σε bit για μεγαλύτερη ευελιξία και ενεργειακή αποδοτικότητα. Ο βασικός του πυρήνας, το **QK-DPU**, διαχειρίζεται πολλαπλασιασμούς bit-serial με λογική ελέγχου που επιτρέπει τον *πρόωρο τερματισμό* όταν τα ενδιάμεσα αποτελέσματα υπολείπονται των εκμαθημένων κατωφλίων.

Βάρη και ενεργοποιήσεις αποθηκεύονται σε bit-serial buffers, και τα μερικά αποτελέσματα συσσωρεύονται σε κύκλους. Μόλις ένα αποτέλεσμα θεωρηθεί αρκετά ακριβές, ο υπολογισμός σταματά νωρίτερα για εξοικονόμηση ενέργειας.

Όπως φαίνεται στο Σχήμα 1.2.5, η μπροστινή μονάδα **QK-PU** περικόπτει τις χαμηλές βαθμολογίες προσοχής κατά τον υπολογισμό $Q \times K^T$ αξιολογώντας μερικά αποτελέσματα bit-serial. Μόνο οι βαθμολογίες που υπερβαίνουν τα κατώφλια προωθούνται στη μονάδα **V-PU** για την επεξεργασία Softmax και $P \cdot V$. Η περικοπή αυτή μειώνει τους περιττούς υπολογισμούς, βελτιώνοντας τόσο την ταχύτητα όσο και την ενεργειακή απόδοση.

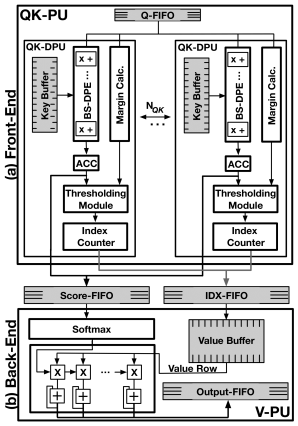


Figure 1.2.5: Γενική μικροαρχιτεκτονική ενός πλακιδίου LeOPard.

Το σύστημα προσφέρει αρκετά πλεονεκτήματα. Υποστηρίζει λειτουργίες όπως $Q \times K^T$, Softmax, και $P \times V$. Μόνο οι πίνακες Q και V αποθηκεύονται σε εσωτερικούς buffers. Εφαρμόζει **περικοπή σε πραγματικό χρόνο** με διαφορετικά κατώφλια ανά στρώμα. Ο σχεδιασμός δίνει έμφαση στην **κλιμακωσιμότητα** και το υψηλό παράλληλο επίπεδο μέσω αρχιτεκτονικής βασισμένης σε πλακίδια. Συνολικά, επιτυγχάνει επιτάχυνση 1.9 έως 2.4 φορές σε σύγκριση με απλές μεθόδους bit-serial, καθώς και μείωση κατανάλωσης ενέργειας 3.9 έως 4.0 φορές.

Αρχιτεκτονική Soft SIMD με Ενεργειακή Αποδοτικότητα

Η υλοποίηση κβαντισμένων CNN σε συσκευές άκρης (edge devices) απαιτεί ευέλικτο και ενεργειακά αποδοτικό υλικό. Οι παραδοσιακές αρχιτεκτονικές SIMD με σταθερό εύρος bit περιορίζουν την απόδοση και την κλιμακωσιμότητα. Η προτεινόμενη μικροαρχιτεκτονική Soft SIMD αντιμετωπίζει αυτό το πρόβλημα, υποστηρίζοντας αυθαίρετα εύρη bit και αποτελεσματικές αριθμητικές πράξεις σταθερής υποδιαστολής, επιτρέποντας υψηλής απόδοσης inference υπό περιορισμό σε χώρο και κατανάλωση ισχύος.

Η αρχιτεκτονική υλοποιεί διάφορες βασικές τεχνικές: **guardbits** χρησιμοποιούνται αντί για πολυπλέκτες για τον διαχωρισμό των υπολέξεων SIMD με ελάχιστο κόστος υλικού· **κωδικοποίηση CSD** μειώνει τον αριθμό των κύκλων πολλαπλασιασμού· **μοντέλα SIMD παραμετροποιήσιμα σε πραγματικό χρόνο** επιτρέπουν αυθαίρετα εύρη bit (π.χ. 3–24 bits)· μηχανή **shift-add** παρέχει αποδοτικούς πολλαπλασιασμούς· και η **μονάδα Data Pack (DPU)** επανασυσκευάζει δυναμικά τα δεδομένα μεταξύ διαφορετικών μορφοποιήσεων SIMD για ευέλικτο υπολογισμό.

Η αρχιτεκτονική υποστηρίζει κυρίως αριθμητικά έντονα έργα απαραίτητα για το inference των CNN, όπως στοιχειώδεις προσθέσεις και αφαιρέσεις, λειτουργίες μετατόπισης (shift), πολλαπλασιασμούς βασισμένους σε shift-add, και λειτουργίες πολλαπλασιασμού-αθροίσματος (MAC) με ασφαλή συσσώρευση για υπερχείλιση μέσω guardbits. Αντιμετωπίζει αποδοτικά τα πυκνά (fully-connected) και συνελικτικά στρώματα μέσω MAC διαφορετικών εύρων bit, αλλά δεν υλοποιεί άμεσα μη γραμμικές ενεργοποιήσεις όπως ReLU στο υλικό, καθώς αυτές συνήθως ελέγχονται από περιφερειακή λογική ή συνδυάζονται σε στρώματα.

Επιπλέον, υποστηρίζει **ετερογενή κβαντικοποίηση ανά στρώμα**, όπου βάρη και ενεργοποιήσεις μπορούν να έχουν κατά το runtime διαφορετικά εύρη bit, ακόμα και μέσα στο ίδιο CNN ανά στρώμα.

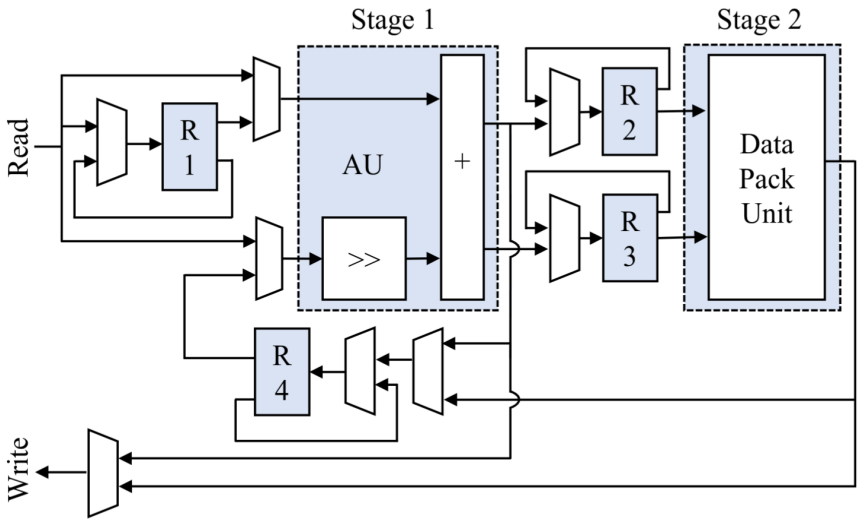


Figure 1.2.6: Διάγραμμα μπλοκ της προτεινόμενης μικροαρχιτεκτονικής Soft SIMD. Το πρώτο στάδιο είναι η Μονάδα Αριθμητικών (Arithmetic Unit - AU), και το δεύτερο η Μονάδα Επανασυσκευασίας Δεδομένων (Data Pack Unit - DPU). R1 έως R4 είναι καταχωρητές.

Architecture	Datatype / Precision	Linear Ops	ReLU	Softmax	Multipliers	Reconfigurable
Gemmini	int8/16/32, uint, float	✓	✓	×	✓	WS / OS mode
Carat	FP8 (low precision)	✓	×	×	×	×
TATAA	int8 /bfloat16	✓	✓	✓	✓	SA / SIMD mode
LeOPArd	int12	✓	×	✓	×	×
Soft SIMD	Arbitrary bits (3–24) int	✓	×	×	×	Precision (bitwidths)

Table 1.1: Comparison of architectures and supported operations/features.

1.2.2 Επιταχυντές Βασισμένοι σε LUT για Μικτής Ακρίβειας Κβαντισμένα Νευρωνικά Δίκτυα

Μέχρι στιγμής, παρουσιάσαμε σημαντικά έργα στον τομέα των επιταχυντών για μεγάλα γλωσσικά μοντέλα (LLMs), που συχνά αξιοποιούν τεχνικές κβαντισμού και καλύπτουν διάφορες αρχιτεκτονικές και στρατηγικές βελτιστοποίησης. Στη συνέχεια, επικεντρωνόμαστε σε επιταχυντές βασισμένους σε LUT (Lookup Table), οι οποίοι σχετίζονται στενά με την αρχιτεκτονική που προτείνεται στη δική μας εργασία. Αυτές οι προσεγγίσεις αντιπροσωπεύουν μια ενδιαφέρουσα κατηγορία, καθώς επιτρέπουν τη σύγκριση και αξιολόγηση με τον σχεδιασμό μας, ο οποίος εισάγει λειτουργίες GEMM βασισμένες σε LUT.

FIGLUT: Επιταχυντής Βασισμένος σε LUT για GEMM FP-INT

Ο **FIGLUT** [19] είναι ένας ενεργειακά αποδοτικός σχεδιασμός επιταχυντή προσαρμοσμένος για λειτουργίες Γενικού Πολλαπλασιασμού Πινάκων (GEMM) με είσοδο κινητής υποδιαστολής (FP) και βάρη ακέραια (INT), χρησιμοποιώντας Πίνακες Αναζήτησης (LUTs). Στοχεύει στην αντιμετώπιση των προκλήσεων που δημιουργούνται από μεγάλα γλωσσικά μοντέλα (LLMs), μειώνοντας την πολυπλοκότητα των υπολογισμών μέσω ανάκτησης δεδομένων από LUT αντί για παραδοσιακές αριθμητικές λειτουργίες. Πολλές υπάρχουσες μέθοδοι αποσυμπιέζουν τα βάρη σε μορφή κινητής υποδιαστολής πριν τον υπολογισμό, εκμεταλλευόμενες όμως ανεπαρκώς την χαμηλή ακρίβεια, μειώνοντας έτσι την αποδοτικότητα. Επιπλέον, υλοποιήσεις βασισμένες σε LUT σε GPUs συχνά αντιμετωπίζουν **συγκρούσεις τραπεζών** (bank conflicts) κατά την πρόσβαση στη μνήμη, που εμποδίζουν την αποδοτική παράλληλη εκτέλεση.

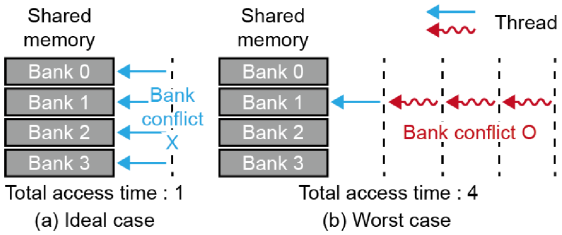


Figure 1.2.7: Σύγκριση συγκρούσεων τραπεζών κατά την πρόσβαση σε κοινή μνήμη

Η υλοποίηση της μνήμης LUT γίνεται με αρχιτεκτονική Flip-Flop LUTs (FFLUT), αντί για κλασικούς καταχωρητές, επιτρέποντας παράλληλες αναγνώσεις από πολλαπλές μονάδες RAC χωρίς συγκρούσεις τραπεζών, με χαμηλή καθυστέρηση και χωρίς ανάγκη διαιτησίας. Αυτό διατηρεί υψηλό ρυθμό εκτέλεσης χωρίς να αυξάνει την κατανάλωση ισχύος ή την πολυπλοκότητα στον προγραμματισμό.

Η αρχιτεκτονική βασίζεται σε διδιάστατο σιστολικό πίνακα, εμπνευσμένο από το TPU της Google, με στρατηγική σταθεροποίησης βαρών (weight-stationary), επιτυγχάνοντας αποδοτική ροή εισόδων κινητής υποδιαστολής, μεγιστοποιώντας την επανάχρηση των βαρών, ελαχιστοποιώντας τις προσβάσεις στη μνήμη και επιτρέποντας την άμεση ενσωμάτωση των τιμών LUT χωρίς καθυστερήσεις.

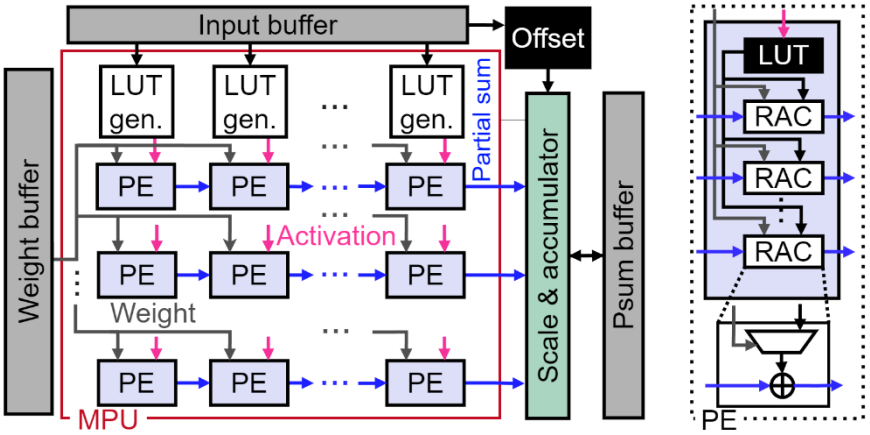


Figure 1.2.8: Συνολική αρχιτεκτονική MPU του FIGLUT

Το FIGLUT παρουσιάζει υψηλή ενεργειακή αποδοτικότητα, με βελτίωση TOPS/W έως 59% σε κβαντοποίηση 3-bit και έως 98% σε 2.4-bit, σε σύγκριση με προηγμένες λύσεις. Παρέχει ευελιξία υποστηρίζοντας ποικίλες μεθόδους κβαντοποίησης όπως FP-INT, BCQ, μικτής ακρίβειας, καθώς και ομοιόμορφη και μη ομοιόμορφη κβαντοποίηση. Χάρη στη χρήση Flip-Flop LUTs, αποφεύγονται οι συγκρούσεις τραπεζών, επιτρέποντας αποδοτικό σιστολικό πλέγμα και επανάχρηση δεδομένων.

Ωστόσο, τα πλεονεκτήματα συνοδεύονται από περιορισμούς, όπως η αύξηση της μνήμης LUT με εκθετικό ρυθμό ως προς το μ, απαιτώντας προσεκτικές σχεδιαστικές επιλογές. Επίσης, η αρχιτεκτονική εισάγει πολυπλοκότητα υλικού, κυρίως λόγω διαχείρισης τάσεων και οδών σημάτων, γεγονός που δυσχεραίνει τη φυσική διαρρύθμιση. Τέλος, το FIGLUT είναι εξειδικευμένος επιταχυντής, βελτιστοποιημένος κυρίως για μοντέλα μόνο με βάρη, και μπορεί να μην είναι αποδοτικός σε γενικότερες εργασίες υπολογισμού.

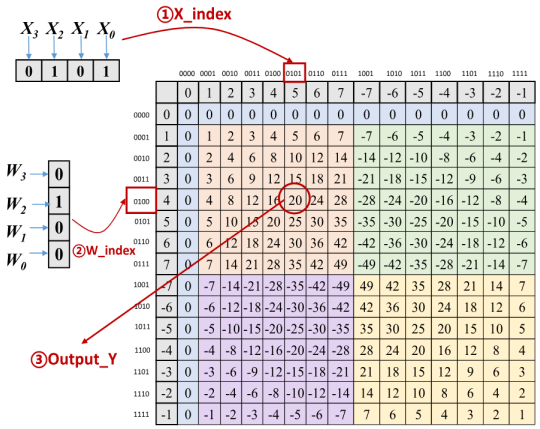


Figure 1.2.9: Βασική αρχιτεκτονική CLM για πολλαπλασιασμό 4-bit

Κβαντισμός CNN 4-bit με Συμπαγείς Πολλαπλασιαστές Βασισμένους σε LUT

Στην εργασία των Zhao et al. (2023) [16] αντιμετωπίζεται η πρόκληση της υλοποίησης συνελικτικών νευρωνικών δικτύων (CNN) σε **ενσωματωμένες συσκευές με περιορισμένους πόρους**, όπως **FPGAs** χωρίς αρκετές μονάδες πολλαπλασιασμού (DSPs). Τα παραδοσιακά CNN απαιτούν μεγάλη υπολογιστική ισχύ και εύρος μνήμης, καθιστώντας τα μη αποδοτικά για edge συσκευές (π.χ. drones, βιομηχανικούς αισθητήρες).

Οι βασικές προκλήσεις είναι:

1. Η μείωση της αριθμητικής ακρίβειας χωρίς απώλεια ακρίβειας.
2. Η υλοποίηση αποδοτικού πολλαπλασιασμού σε FPGAs χωρίς χρήση μπλοκ DSP.

Η υλοποίηση στοχεύει αρχικά μόνο στο *inference*, όπου το μοντέλο εκπαιδεύεται με ακρίβεια `fp32` και στη συνέχεια εφαρμόζεται η **Threshold-Aware Quantization (TAQ)** σε βάρη και ενεργοποιήσεις, κβαντίζοντας τα σε 4-bit ακέραιους χωρίς πρόσημο (0 έως 15) [8, 33, 17].

Η μέθοδος TAQ ανήκει στον κλάδο της **post-training quantization**, επιδιώκοντας αποδοτική υλοποίηση σε FPGA χωρίς εκ νέου εκπαίδευση. Χρησιμοποιεί μη ομοιόμορφο χάρτη τιμών με **custom thresholds** για ελαχιστοποίηση σφάλματος κβαντισμού, και **μεικτή στρογγυλοποίηση** που εναλλάσσει μεθόδους όπως round-nearest και floor ανάλογα με την εγγύτητα σε κατώφλια και την κατανομή δεδομένων.

Κρίσιμο και καινοτόμο στοιχείο της αρχιτεκτονικής είναι η πλήρης **απομάκρυνση των αριθμητικών πολλαπλασιαστών** από το υλικό. Αντί για παραδοσιακούς πολλαπλασιαστές (π.χ. DSPs), υλοποιείται **πλήρης προ-υπολογισμός** σε Πίνακες Αναζήτησης (LUT) λόγω του περιορισμένου συνόλου δυνατών πολλαπλασιασμών που προκύπτουν από την 4-bit κβαντοποίηση.

Κάθε ζεύγος τιμών (0–15) οδηγεί σε 256 δυνατές εκβάσεις πολλαπλασιασμού. Η αρχιτεκτονική εκμεταλλεύεται αυτό, αποθηκεύοντας όλα τα δυνατά γινόμενα σε LUTs υλοποιημένους με μονάδες LUT6, βασικό στοιχείο σε σύγχρονα FPGAs. Το τελικό αποτέλεσμα είναι ο **Compact LUT-based Multiplier (CLM)**, που χρησιμοποιεί μόλις 13 blocks LUT6 ανά πολλαπλασιαστή, επιτυγχάνοντας υψηλή αποδοτικότητα σε χώρο και ενέργεια.

Κατά την εκτέλεση, ο CLM παίρνει τις 4-bit τιμές ενεργοποίησης και βάρους ως διεύθυνση για γρήγορη ανάκτηση του προϋπολογισμένου γινομένου από τον LUT, αντικαθιστώντας πλήρως τον πολλαπλασιασμό με ταχύτατη μνήμη.

Η τεχνική μειώνει πολύπλοκο κύκλωμα, εξαλείφει την ανάγκη DSP και επιτρέπει **μαζική παραλληλοποίηση πολλαπλασιασμών**, κρίσιμη για inference σε CNNs.

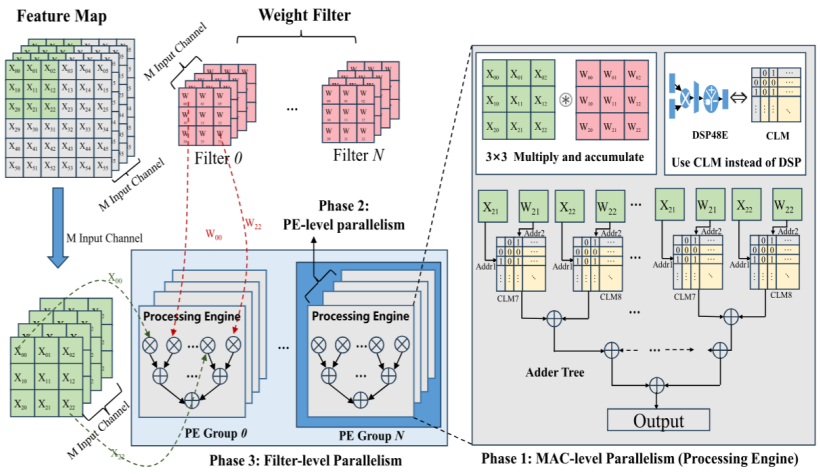


Figure 1.2.10: Παράλληλοι πολλαπλασιασμοί για συνελικτικούς υπολογισμούς σε FPGA επιταχυντή

Οι μονάδες CLM αντικαθιστούν παραδοσιακούς πολλαπλασιαστές όπως DSPs, με κάθε πολλαπλασιασμός να διαχειρίζεται ανεξάρτητα η δική του μονάδα CLM. Για παράδειγμα, για παράθυρο 3×3 σε συνελικτικό στρώμα απαιτούνται 9 μονάδες CLM.

Η παράλληλη σχεδίαση αποτρέπει συγκρούσεις δεδομένων, καθώς κάθε ζεύγος ενεργοποίησης-βάρους έχει τη δική του μονάδα CLM που ανακτά το προϋπολογισμένο γινόμενο, διασφαλίζοντας υψηλή διαμέσου και πλήρως παράλληλη εκτέλεση.

LUT Tensor Core

Η εργασία παρουσιάζει το **LUT Tensor Core**, μια αρχιτεκτονική συν-σχεδιασμού υλικού-λογισμικού προσαρμοσμένη για αποδοτικές λειτουργίες GEMM χαμηλής ακρίβειας σε inference μεγάλων γλωσσικών μοντέλων (LLMs). Χρησιμοποιεί προϋπολογισμένους Πίνακες Αναζήτησης (LUTs) για να αντικαταστήσει παραδοσιακούς πολλαπλασιαστές, ιδίως σε περιβάλλοντα μικτής ακρίβειας.

Οι παραδοσιακές λύσεις βασισμένες σε LUT αντιμετωπίζουν προκλήσεις όπως αναποτελεσματική αποκβαντοποίηση, περιορισμένη υποστήριξη μικτής ακρίβειας από το υλικό, μεγάλο κόστος αποθήκευσης και υπολογισμού για τους πίνακες LUT, και υποβέλτιστες στρατηγικές τοποθέτησης (tiling). Επιπλέον, η έλλειψη ειδικών συνόλων εντολών και υποστήριξης μεταγλωττιστή περιορίζει την απόδοση.

Η προτεινόμενη αρχιτεκτονική λύνει αυτά τα προβλήματα, υποστηρίζοντας ενεργοποιήσεις υψηλής ακρίβειας (π.χ. FP16, INT8) και βάρη πολύ χαμηλής ακρίβειας (1–4 bits). Είναι **παράλληλα παραμετροποιήσιμη** σε πραγματικό χρόνο, επιτρέποντας ευέλικτη υποστήριξη διαφοροποιημένων συνδυασμών ακρίβειας ανάλογα με το φόρτο.

Η βασική ιδέα είναι η φόρτωση ενός διανύσματος ενεργοποιήσεων και ο προϋπολογισμός όλων των πιθανών εσωτερικών γινομένων βάσει της ακρίβειας βαρών. Κατά το runtime, τα βάρη επιλέγουν τα κατάλληλα μερικά αθροίσματα από τον LUT, αποφεύγοντας τον υπολογισμό πολλαπλασιασμών σε πραγματικό χρόνο. Τα μερικά αποτελέσματα συσσωρεύονται σε εξόδους, οι οποίες παραμένουν **εντός τσιπ** μέχρι την ολοκλήρωση, ακολουθώντας το πρότυπο δεδομένων **output-stationary**.

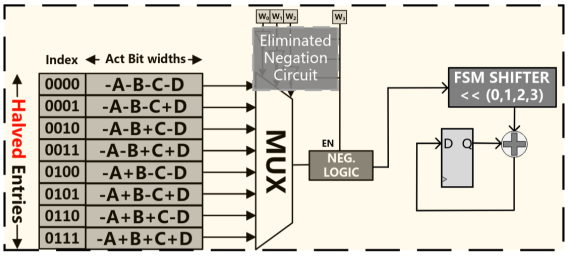


Figure 8: Optimized LUT unit with bit-serial.

Figure 1.2.12: Βελτιστοποιημένη μονάδα LUT με bit-serial προσέγγιση

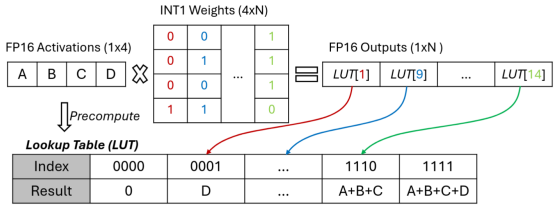


Figure 1.2.11: Παράδειγμα στοιχείου LUT-based mpGEMM για ενεργοποιήσεις FP16 και βάρη INT1. Η αναζήτηση στον πίνακα αντικαθιστά το εσωτερικό γινόμενο 4-στοιχείων.

Για μείωση της χωρητικότητας αποθήκευσης του LUT, η αρχιτεκτονική εκμεταλλεύεται τη **συμμετρία των τιμών βαρών**: π.χ. δυαδικά βάρη $\{0, 1\}$ μπορούν να αναπαρασταθούν ως $\{-1, 1\}$, μειώνοντας κατά το ήμισυ τις καταχωρήσεις.

Για υποστήριξη αυθαίρετου εύρους bit βαρών (1–4), χρησιμοποιείται **bit-serial αρχιτεκτονική**, όπου κάθε βάρος επεξεργάζεται σε W_{BIT} κύκλους, επιτρέποντας σειριακό υπολογισμό μικτής ακρίβειας με ισορροπία ευελιξίας και αποδοτικότητας περιοχής.

Η εικόνα 3.2.10 παρουσιάζει τη συμβατική τριών βημάτων διαδικασία LUT-based mpGEMM: (1) προϋπολογισμός πίνακα, (2) ανάκτηση τιμών, και (3) συσσώρευση μερικών αθροισμάτων. Ταυτόχρονα, επισημαίνονται περιορισμοί όπως ανάγκη μεγάλης αποθήκευσης με αυξημένη περιοχή και καθυστέρηση, πολυπλοκότητα σε υποστήριξη πολλαπλών εύρων bit, μη βέλτιστη τοποθέτηση πινάκων και έλλειψη εξειδικευμένων εντολών και compiler, δυσχεραίνοντας την ενσωμάτωση και αποδοτικό προγραμματισμό.

Η αρχιτεκτονική φορτώνει διανύσματα K ενεργοποιήσεων, απαιτώντας έναν πίνακα LUT ανά διάνυσμα. Για M διανύσματα, απαιτούνται M πίνακες LUT, καθένας με 2^{K-1} θέσεις (με βελτίωση συμμετρίας), προσπελάσιμες μέσω λογικής επιλογής (MUX) που ελέγχεται από δυαδικά βάρη.

- M: αριθμός γραμμών ενεργοποιήσεων — καθορίζει πόσοι LUT πίνακες χρειάζονται,
- N: αριθμός βαρών ανά ενεργοποίηση — κάθε LUT τροφοδοτεί N υπολογισμούς βαρών (μέσω μονάδων MUX),
- K: αριθμός bit δυαδικών βαρών — καθορίζει εύρος διεύθυνσης LUT και γραμμές επιλογής MUX (βάθος κβαντισμού).

Ο αριθμός **δυαδικών ομάδων βαρών** (bit depth) ορίζει τα bits ανά ομάδα βαρών. Κάθε bit χρησιμοποιείται για επιλογή από τον LUT μέσω MUX.

Ο LUT Tensor Core υπερτερεί σημαντικά έναντι υλοποιήσεων LUT καθαρά λογισμικού, επιτυγχάνοντας 1.44× βελτίωση στην πυκνότητα υπολογισμών και ενεργειακή απόδοση από προηγμένους επιταχυντές βασισμένους σε LUT.

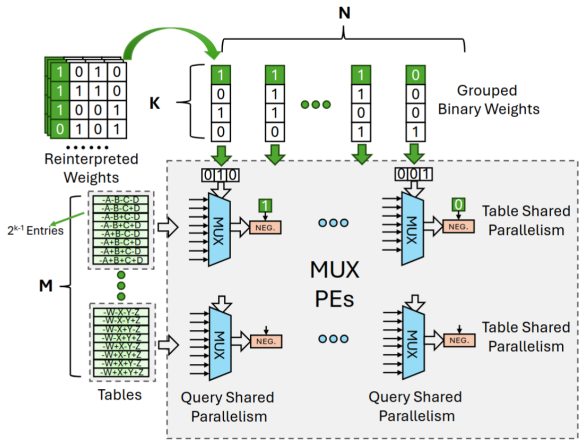


Figure 1.2.13: Επιμηκυμένη τοποθέτηση MNK του LUT-based Tensor Core. Απαιτεί μεγάλο N (π.χ. 64/128) για μέγιστη επαναχρησιμοποίηση και κατάλληλα K (π.χ. 4) για αποδοτικό μέγεθος πίνακα.

Table 1.2: **LUMAX compared to LUT-based accelerators**

	LUT Tensor [21]	**FIGLUT [19]**	**CLM [16]**	**LUMAX**
Act.	FP/INT8, FP/INT16	FP16	INT4	INT16,INT8
Wgt.	INT1–INT4	INT1–INT4	INT4	INT1–INT8
Bit-serial	✓	✓	×	×
Generate	✓	✓	×	✓
Stationary	Output	Weight	Weight	Output
Platform	ASIC	ASIC	FPGA	FPGA
LUT entries	$2^{(N-1)}$	$2^{(N-1)}$	$N \times 64$	$N \times 2^{(\mathrm{W\_width}-2)}$

1.3 Θεωρητικό Υπόβαθρο

1.3.1 Το Περιβάλλον Ανάπτυξης Chipyard

Το Chipyard είναι ένα ανοικτού κώδικα πλαίσιο σχεδιασμού υλικού [7] βασισμένο στη γλώσσα καθορισμού υλικού *Chisel* [30]. Αναπτύχθηκε ειδικά για την υποστήριξη ευέλικτου και κλιμακούμενου σχεδιασμού αρχιτεκτονικών SoC βασισμένων στον RISC-V. Το Chipyard συνδυάζει διάφορα εργαλεία και βιβλιοθήκες σε ένα ενιαίο περιβάλλον που διευκολύνει τον γρήγορο πρωτοτυποποίηση, προσομοίωση, σύνθεση και ανάπτυξη λογισμικού.

- **Σχεδιασμός με βάση το Chisel:** Χρησιμοποιεί το Chisel, μια ισχυρή γλώσσα περιγραφής υλικού ενσωματωμένη στη Scala, επιτρέποντας τη δημιουργία αρθρωτών, παραμετροποιήσιμων και επαναχρησιμοποιήσιμων υποσυστημάτων.
- **Δημιουργία RISC-V SoC:** Υποστηρίζει τη δημιουργία πλήρων σχεδίων SoC με γεννήτριες όπως το Rocket Chip και BOOM, με παραμετροποιήσιμους πυρήνες και συνιστώσες.
- **Ενσωματωμένο εργαλείο ανάπτυξης:** Περιλαμβάνει εργαλεία για:
  - *RTL προσομοίωση:* Υποστήριξη μέσω Verilator και εμπορικών εργαλείων όπως Synopsys VCS.
  - *Εξομοίωση FPGA:* Επιταχυνόμενη προσομοίωση μέσω FireSim σε cloud FPGA.
  - *VLSI ροές:* Περιλαμβάνει Hammer για αυτόματο φυσικό σχεδιασμό και τοποθέτηση.
  - *Κατασκευή λογισμικού:* Το FireMarshal βοηθά στην ανάπτυξη bare-metal και Linux-based εφαρμογών.

- **Επιτάχυνση Υλικού:** Διευκολύνει την ενσωμάτωση και δοκιμή προσαρμοσμένων επιταχυντών (π.χ. Hwacha, Gemmini), επιτρέποντας το συν-σχεδιασμό υλικού και λογισμικού.
- **Ανοικτός και επεκτάσιμος:** Διατηρείται από το Berkeley Architecture Research Group και υποστηρίζει συνεισφορές από την κοινότητα ανοικτού κώδικα.
- **Ενσωμάτωση στο οικοσύστημα RISC-V:** Ενσωματώνεται ομαλά στο ευρύτερο οικοσύστημα RISC-V, παρέχοντας ευέλικτη πλατφόρμα για έρευνα και πειραματισμό.
- **Πρωτοτυποποίηση και επικύρωση FPGA:** Το FireSim επιτρέπει γρήγορη πρωτοτυποποίηση και έλεγχο σε επίπεδο συστήματος χρησιμοποιώντας cloud FPGA.
- **Υποστήριξη συν-σχεδιασμού:** Επιτρέπει παράλληλη ανάπτυξη υλικού και λογισμικού, ενθαρρύνοντας ολιστική προσέγγιση σχεδιασμού.

Συνοψίζοντας, το Chipyard λειτουργεί ως ολοκληρωμένο περιβάλλον για το σχεδιασμό, την πρωτοτυποποίηση και την αξιολόγηση SoC RISC-V, καθιστώντας το ισχυρό εργαλείο τόσο για ακαδημαϊκή έρευνα όσο και για βιομηχανική ανάπτυξη υλικού.

1.3.2 Chisel

To Chisel δεν είναι μια παραδοσιακή γλώσσα σύνθεσης υψηλού επιπέδου (HLS), ούτε μια χαμηλού επιπέδου γλώσσα περιγραφής υλικού (HDL) όπως η Verilog ή VHDL. Αντιθέτως, είναι μια **κατασκευαστική γλώσσα περιγραφής υλικού**, ενσωματωμένη στη Scala, που τελικά παράγει κώδικα Verilog ή SystemVerilog για σύνθεση.

Το Chisel επιτρέπει στους σχεδιαστές να περιγράφουν ψηφιακό υλικό σε υψηλό επίπεδο, διατηρώντας παράλληλα τη δομή και τον λεπτομερή έλεγχο που παρέχουν οι παραδοσιακές HDL γλώσσες. Παρέχει πλούσια API, αντικειμενοστραφείς κατασκευές και λειτουργίες λειτουργικού προγραμματισμού χάρη στη Scala. Όταν ενσωματώνεται στο περιβάλλον `Chipyard`, καθίσταται εύκολη η εισαγωγή ή τροποποίηση σχεδίων, όπως η δημιουργία προσαρμοσμένου συνεπεξεργαστή RISC-V με λίγα βήματα.

Το Chisel προάγει τον σχεδιασμό βασισμένο σε συνιστώσες, όπου κάθε module έχει καλά ορισμένες εισόδους και εξόδους. Επιπλέον, υποστηρίζει τη χρήση ήδη υπαρχόντων Verilog modules μέσω περιτυλίξεων `BlackBox`, διευκολύνοντας την επαναχρησιμοποίηση παλαιού κώδικα HDL.

1.3.3 RocketChip

Ο *RocketChip* [20] είναι ένας πυρήνας βασισμένος στο RISC-V και ήταν το πρώτο έργο περιγραφής υλικού για το ISA RISC-V. Πρόκειται για έναν παραμετροποιήσιμο ανοικτού κώδικα γεννήτρια SoC, ικανό να ενσωματώνει πλήθος πυρήνων και επιταχυντών. Η διαμόρφωση του RocketCore είναι ιδιαίτερα αρθρωτή, δίνοντας τη δυνατότητα παραμετροποίησης πυρήνων και στοιχείων. Χάρη σε αυτή την αρθρωτότητα, μπορούν να δημιουργηθούν εφαρμογοκεντρικοί επιταχυντές με διαφορετικές ρυθμίσεις. Η αρχιτεκτονική υποστηρίζει επίσης επεκτάσιμα μπλοκ πέραν του πυρήνα, όπως L1 και L2 κρυφές μνήμες, μονάδα διαχείρισης μνήμης (MMU), μονάδα κινητής υποδιαστολής (FPU), μονάδες εκτέλεσης διανυσμάτων, μονάδα αποσφαλμάτωσης, μετρητές απόδοσης, ελεγκτές διακοπών, καθώς και υποδομές επικοινωνίας μεταξύ αυτών.

Το *RocketTile* είναι το επόμενο επίπεδο διαμόρφωσης μέσα στο RocketChip, επιτρέποντας προσαρμογές όπως cache, MMU, FPU και άλλα στοιχεία ελέγχου. Το RocketTile αποτελεί τη βασική μονάδα του RocketChip και μπορεί να διαμορφωθεί ως μονοπύρηνο, πολυπύρηνο ή πολλαπλού cluster. Βασικές διαμορφώσεις είναι:

- **BigCore**: Υψηλής απόδοσης με 16 KiB 4-way set-associative cache και υποστήριξη FPU.
- **MediumCore**: Με μικρότερες 4 KiB direct-mapped caches και χωρίς υποστήριξη FPU από προεπιλογή.
- **SmallCore**: Χαμηλής απόδοσης με περιορισμένα cache και χωρίς FPU από προεπιλογή.

- **TinyCore**: Λιγότερο διαδεδομένη διαμόρφωση, υποστηρίζει μόνο 32-bit αρχιτεκτονική.

1.3.4 Επιταχυντής RoCC

Ο *RoCC* (Tightly-Coupled Rocket Custom Coprocessor) είναι ένας προσαρμοσμένος επιταχυντής στενά ενσωματωμένος με τους πυρήνες RISC-V σε ένα SoC, προσφέροντας υψηλή απόδοση μέσω στενής επικοινωνίας με τη CPU. Χαρακτηριστικά:

- **Επικοινωνία μέσω προσαρμοσμένων εντολών:** RoCC ελέγχεται με ειδικές εντολές μορφής *customX rd, rs1, rs2, funct*, όπου X (0–3) καθορίζει τον επιταχυντή-στόχο (έως 4 RoCCs), και το πεδίο funct (7-bit) διαχωρίζει διαφορετικές λειτουργίες.
- **Άμεση πρόσβαση σε πόρους CPU:** Διαμοιράζεται καταχωρητές (rs1, rs2, rd) και cache L1 με τον πυρήνα. Μέσω της διεπαφής mem, μπορεί να εκτελεί load/store απευθείας στη μνήμη, ενώ το ptw παρέχει πρόσβαση στον page-table walker για διαχείριση εικονικής μνήμης. Μπορεί επίσης να τροφοδοτεί interrupt στην CPU μετά την ολοκλήρωση εργασίας.
- **Ενσωμάτωση στο σύστημα:** Συνδέεται με το δίκτυο TileLink μέσω tlNode ή atlNode.
- **Προσαρμοσμένο εργαλείο ανάπτυξης:** Απαιτεί προσαρμοσμένη αλυσίδα εργαλείων, καθώς οι customX εντολές δεν είναι στάνταρ στο ISA του RISC-V. Η χρήση γίνεται μέσω μακροεντολών (macros).

Σύγκριση με περιφερειακές συσκευές MMIO: Οι MMIO περιφερειακές συσκευές επικοινωνούν μέσω μνημονικά χαρτογραφημένων καταχωρητών και απαιτούν στάνταρ αλυσίδα εργαλείων, ενώ οι RoCC προσφέρουν μικρότερη καθυστέρηση και στενότερη ολοκλήρωση με την CPU, απαιτώντας εξειδικευμένη γνώση υλικού (Chisel) και λογισμικού (μακροεντολές).

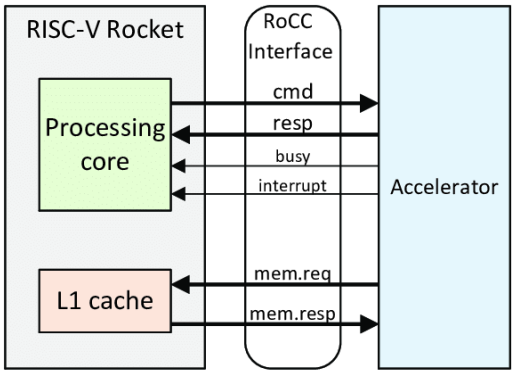


Figure 1.3.1: Απλοποιημένη άποψη της διεπαφής RoCC

1.3.5 TileLink και Diplomacy

Το *TileLink* [21] είναι ένα ανοιχτό πρωτόκολλο διασύνδεσης για σχεδιασμούς SoC, αρχικά αναπτυγμένο για RISC-V αλλά ανεξάρτητο από συγκεκριμένο ISA. Προσφέρει συνεπή, μνημονικά χαρτογραφημένη επικοινωνία μεταξύ πολλών masters (π.χ. επεξεργαστών, DMA) και slaves (π.χ. μνήμες, περιφερειακά), με έμφαση σε χαμηλή καθυστέρηση, υψηλή διαμέσου και κλιμακωσιμότητα.

Υποστηρίζει πλήρη συνοχή μνήμης μέσω πρωτοκόλλου τύπου MOESI [27,28], διατηρώντας την εγκυρότητα των γραμμών cache με πέντε καταστάσεις — Modified, Owned, Exclusive, Shared και Invalid. Αυτό επιτρέπει αποδοτικές μεταφορές cache-to-cache και μειώνει περιττή κίνηση μνήμης, ενώ είναι εγγενώς αποθήκευση αποφυγή αδιεξόδων (deadlock-free). Υποστηρίζει εκτέλεση παραγγελιών εκτός

σειράς, αποσυνδεδεμένα interfaces και ιεραρχική σύνθεση δικτύων point-to-point, επιτρέποντας κλιμάκωση από απλές σε πολύπλοκες αρχιτεκτονικές. Επιπλέον, χρησιμοποιεί τεχνικές χαμηλής ισχύος όπως αποδοτικό κωδικοποίηση σημάτων.

Το πρωτόκολλο υλοποιείται μέσω του *Diplomacy*, που αυτοματοποιεί τη σύνδεση και παραμετροποίηση των συστημικών συνιστωσών (π.χ. cache L1, ελεγκτές μνήμης) και τη διαχείριση διευθύνσεων. Το TileLink ορίζει πέντε κανάλια επικοινωνίας (A, B, C, D, E) με συγκεκριμένες διευθύνσεις και προτεραιότητες, εξασφαλίζοντας αδιάκοπη ροή δεδομένων μεταξύ masters και slaves, όπως φαίνεται στο Σχήμα 4.0.2.

Τα *Diplomatic Widgets* που παρέχει η βιβλιοθήκη RocketChip διευκολύνουν τη διασύνδεση ετερογενών συνιστωσών, παρέχοντας λειτουργίες buffering, αναδιάταξης, κατακερματισμού, μετατροπές μεταξύ TileLink και AXI4 [29,30], και την κατασκευή σύνθετων συστημάτων SoC μέσω crossbars, FIFOs και μετατροπέων πρωτοκόλλων. Το προέκτασιο *Advanced eXtensible Interface* (AXI) είναι ένα ευρέως χρησιμοποιούμενο πρωτόκολλο επικοινωνίας επί τσιπ, που υποστηρίζει υψηλή απόδοση, χαμηλή καθυστέρηση και μεταφορές σε burst, επιτρέποντας αποδοτική και ευέλικτη επικοινωνία σε σύνθετα SoCs.

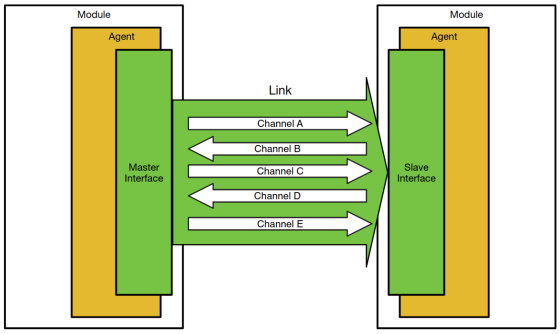


Figure 1.3.2: Τα πέντε κανάλια TileLink μεταξύ master και slave. Η ιεραρχική προτεραιοποίηση αποτρέπει αδιέξοδα και διασφαλίζει ροή δεδομένων.

1.3.6 Συγχρονισμένες Μνήμες Ανάγνωσης (Sync Read Memories)

Η γλώσσα Chisel παρέχει API για ορισμό μονάδων μνήμης έτοιμων προς χρήση, υποστηρίζοντας τόσο ROM όσο και μνήμες ανάγνωσης/εγγραφής.

ROM: Οι χρήστες μπορούν να ορίσουν ROM κατασκευάζοντας `Vec` με `VecInit`, δεχόμενο είτε λίστα σταθερών δεδομένων είτε ακολουθία `Seq[Data]` για αρχικοποίηση. Για παράδειγμα, μπορεί να δημιουργηθεί μικρό ROM με τιμές `1, 2, 4, 8` και πρόσβαση μέσω γεννήτριας διευθύνσεων όπως ένας απαριθμητής.

Μνήμες Ανάγνωσης/Εγγραφής: Οι υλοποιήσεις μνήμης ποικίλουν μεταξύ FPGA και ASIC. Το Chisel παρέχει τα κατασκευάσματα `Mem` και `SyncReadMem`:

- `Mem`: Συνδυαστική (ασύγχρονη ανάγνωση), σειριακή (σύγχρονη εγγραφή)
- `SyncReadMem`: Σύγχρονη ανάγνωση και εγγραφή

Στον σχεδιασμό μας, χρησιμοποιούμε `SyncReadMem` ως Πίνακες Αναζήτησης (LUT) για αποθήκευση μερικών προϊόντων κατά τους υπολογισμούς.

Συμπεριφορά `SyncReadMem`: Το `SyncReadMem` είναι κατασκεύασμα με συγχρονισμένη ανάγνωση και εγγραφή. Οι μνήμες αυτές αναμένεται να υλοποιούνται κυρίως ως SRAM ειδικής τεχνολογίας και όχι flip-flop banks. Εάν γίνεται εγγραφή και ανάγνωση στην ίδια ακμή ρολογιού ή αν η ενεργοποίηση ανάγνωσης αναιρείται, η τιμή ανάγνωσης είναι μη καθορισμένη. Επίσης, η θύρα ανάγνωσης δεν διατηρεί δεδομένα μεταξύ κύκλων εκτός αν καταγραφεί εξωτερικά.

Διεπαφή Ανάγνωσης/Εγγραφής: Η πρόσβαση γίνεται μέσω δείκτη `UInt`. Παραδείγματος χάριν, ορίστε SRAM 1024 θέσεων με μία θύρα ανάγνωσης και εγγραφής:

1.3.7 Θεωρητικό Υπόβαθρο: LLMs και Κβαντισμός

Ο ρόλος των DNN και των Πολλαπλασιασμών Πινάκων στα LLMs

Τα βαθιά νευρωνικά δίκτυα (DNNs) έχουν φέρει επανάσταση στην αντιμετώπιση σύνθετων εργασιών όπως όραση, αναγνώριση ομιλίας, ρομποτική και επεξεργασία φυσικής γλώσσας. Από παραδοσιακά CNN έως σύγχρονες αρχιτεκτονικές transformer, τα DNN αποτελούν το βασικό στοιχείο της σύγχρονης τεχνητής νοημοσύνης.

Τα Μεγάλα Γλωσσικά Μοντέλα (LLMs) όπως GPT, BERT, PaLM και LLaMA αναδεικνύουν εξαιρετικές ικανότητες στη δημιουργία κειμένου, περίληψη, μετάφραση και απάντηση σε ερωτήματα, επιτυγχάνοντας πολλές φορές ανθρώπινα επίπεδα απόδοσης. Βασίζονται στην αρχιτεκτονική transformer [12], εκπαιδεύονται σε τεράστιες συλλογές δεδομένων και μαθαίνουν στατιστικά μοτίβα γλώσσας.

Η φάση inference — όπου εκπαιδευμένα μοντέλα χρησιμοποιούνται για παραγωγή αποτελεσμάτων — είναι κρίσιμη για εφαρμογές όπως chatbots, μηχανές μετάφρασης και έξυπνους βοηθούς. Ωστόσο, το inference στα LLMs είναι ιδιαίτερα υπολογιστικά απαιτητικό λόγω μεγέθους μοντέλου και των υποκείμενων πράξεων πολλαπλασιασμού πινάκων.

Οι πράξεις GEMM (Γενικός Πολλαπλασιασμός Πινάκων) είναι ο πυρήνας των υπολογισμών σε DNNs, όπως σε fully connected layers, μηχανισμούς προσοχής και στρώματα προσομοίωσης διαστάσεων.

Οι πράξεις GEMM ευθύνονται για το μεγαλύτερο τμήμα του χρόνου υπολογισμού και της κατανάλωσης ενέργειας στα LLMs. Πρόσφατες μελέτες δείχνουν ότι τα στρώματα προσοχής — που στηρίζονται βαριά σε GEMM — καταναλώνουν 70-90% του χρόνου inference σε CPUs [14]. Έτσι, η βελτιστοποίηση των πυρήνων GEMM είναι κρίσιμη για την επιτάχυνση και την ενεργειακή αποδοτικότητα.

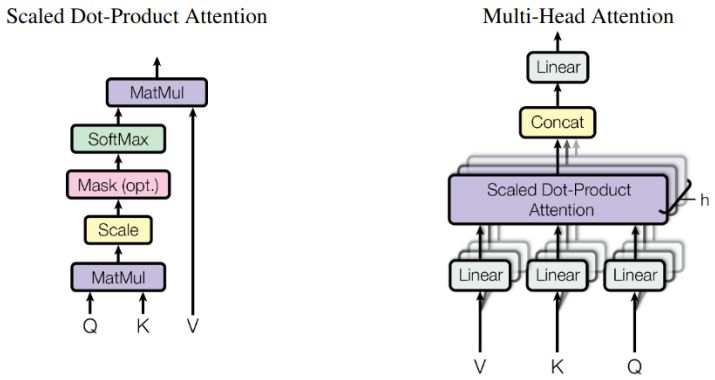


Figure 1.3.3: Μπλοκ αυτοπροσοχής και πολυκεφαλικής προσοχής

Η αποδοτική υλοποίηση των GEMM είναι βασικός στόχος για την επιτάχυνση του inference, ιδιαίτερα σε σενάρια edge computing με περιορισμένους πόρους ισχύος και μνήμης.

Κβαντισμός για Αποδοτική Εκτέλεση

Ο κβαντισμός αποτελεί βασική τεχνική μείωσης του υπολογιστικού κόστους σε DNNs και LLMs. Αντί για 32-bit κινητής υποδιαστολής (FP32), βάρη και ενεργοποιήσεις μετατρέπονται σε μορφές χαμηλότερης ακρίβειας όπως INT8, FP16, FP8 ή INT4. Πρόσφατες έρευνες εξετάζουν μορφές υπερχαμηλής ακρίβειας για αύξηση αποδοτικότητας χωρίς απώλεια ακρίβειας.

Η μείωση του μεγέθους των δεδομένων βελτιώνει το αποθηκευτικό χώρο, τους χρόνους μεταφοράς και τη λανθάνουσα των υπολογισμών, ιδιαίτερα σε αποκλειστικούς επιταχυντές όπως GPUs, TPUs και ειδικά AI chips.

Ο κβαντισμός είναι αναγκαίος για την εκτέλεση LLMs με δισεκατομμύρια παραμέτρους σε περιορισμένα συστήματα, βελτιώνοντας επίσης τη λανθάνουσα σε real-time εφαρμογές.

Δύο βασικές προσεγγίσεις είναι η **Post-Training Quantization (PTQ)**, που εφαρμόζεται μετά την εκπαίδευση χωρίς επανεκπαίδευση, και η **Quantization-Aware Training (QAT)**, που ενσωματώνει τον κβαντισμό κατά την εκπαίδευση για αυξημένη ακρίβεια, ειδικά σε χαμηλές ακρίβειες.

Προηγμένες τεχνικές, όπως ο κβαντισμός ανά κανάλι και μικτής ακρίβειας, βελτιώνουν την απόδοση. Ο PTQ με σταθερό σημείο είναι πιο συμβατός με υλικό, ενώ οι μικτής ακρίβειας μέθοδοι απαιτούν πιο σύνθετη διαχείριση.

Η μικτής ακρίβειας κβαντοποίηση αναθέτει διαφορετικά επίπεδα ακρίβειας σε διάφορα μέρη του δικτύου για βέλτιστη ισορροπία μεταξύ απόδοσης και ακρίβειας. Π.χ., βάρη με πολύ χαμηλή ακρίβεια (INT4 ή INT2) και ενεργοποιήσεις με υψηλότερη (INT8, FP16).

Συνολικά, ο κβαντισμός αποτελεί θεμελιώδη στρατηγική για την αποδοτική εκτέλεση και πρακτική χρήση μεγάλων γλωσσικών μοντέλων σε πόρους περιορισμένα περιβάλλοντα.

1.4 Σχεδασμός Επιταχυντή

Ο επιταχυντής LUMAX αποτελείται από διακριτές μονάδες υπολογισμού και μνήμης, με την συνολική αρχιτεκτονική να απεικονίζεται στο Σχήμα 5.1.1. Υλοποιείται ως συνεπεξεργαστής RocketChip, με διαχείριση εντολών μέσω της διεπαφής RoCC και μεταφορές δεδομένων μέσω καναλιών DMA. Τα δεδομένα εισόδου επεξεργάζονται ώστε να αποθηκεύονται μόνο οι θετικοί άρτιοι πολλαπλασιασμοί κάθε ενεργοποίησης, ενώ μηχανισμός επιλογής ανασυνθέτει τη σωστή ακολουθία μερικών γινομένων και κωδικοποιεί πληροφορίες για περιττούς/άρτιους πολλαπλασιασμούς και πρόσημα. Το αποτέλεσμα προωθείται στον αθροιστή για το τελικό προϊόν. Όλες οι λειτουργίες εισόδου και εξόδου διαχειρίζονται από αφιερωμένους buffers στον επιταχυντή.

Η εργασία εστιάζει στο σχεδιασμό συστήματος επιταχυντή υλικού DNN που υποστηρίζει **μικτής ακρίβειας κβαντισμένα δεδομένα**, επιτρέποντας αυθαίρετους συνδυασμούς bitwidth μεταξύ βαρών και ενεργοποιήσεων. Αυτή η ευελιξία βοηθά στον βέλτιστο συντονισμό παραμέτρων για βελτίωση της απόδοσης. Παρέχεται σετ συναρτήσεων σε C που χρησιμοποιούν τις προσαρμοσμένες εντολές ελέγχου για ευκολία χρήσης.

Ο επιταχυντής αναπτύχθηκε με τη γλώσσα περιγραφής υλικού Chisel μέσα στο οικοσύστημα Chipyard. Η αρχιτεκτονική βασίζεται σε νέα μέθοδο πολλαπλασιασμού πινάκων με Πίνακες Αναζήτησης (LUT), προσαρμοσμένη σε μικτής ακρίβειας κβαντισμένα μοντέλα όπου τα βάρη έχουν χαμηλότερη ακρίβεια (π.χ. int8/int4/int2) σε σχέση με τις ενεργοποιήσεις (π.χ. int16/int8).

Το σύστημα περιλαμβάνει έναν **Rocket Core**, μία μονάδα **DMA** με υλικό επικοινωνίας βάσει πρωτοκόλλου TileLink και τον επιταχυντή RoCC. Για την αποθήκευση των παρτίδων δεδομένων υπάρχουν buffers για τις ενεργοποιήσεις (**Input Buffer**), τα βάρη (**Weight Buffer**) και τα αποτελέσματα (**Output Buffer**).

Χρησιμοποιούνται **Ping-Pong Buffers** για τα βάρη ώστε να γράφονται τιμές στη μνήμη ενώ διαβάζονται παράλληλα για δεικτοδότηση. Ο **Product Generator** παράγει τα γινόμενα με προσθέσεις, τα αποθηκεύει σε block μνήμης (σειριακής ανάγνωσης), με καθυστέρηση ενός κύκλου.

Η μονάδα **Select Module** εντοπίζει τον δείκτη βάρους και θέση προϊόντος σε κάθε μνήμη, ενώ ο **Accumulator Block** αθροίζει όλα τα προϊόντα σχηματίζοντας σωστά τελικά στοιχεία εξόδου. Τέλος, η μονάδα **Scale Factor** κλιμακώνει γραμμές εξόδου με σταθερό σημείο.

Ο Rocket Core ελέγχει με προσαρμοσμένες RoCC εντολές που διαχειρίζονται σήματα ελέγχου, προκαταβάλλουν διευθύνσεις δεδομένων και διαχειρίζονται την εκτέλεση.

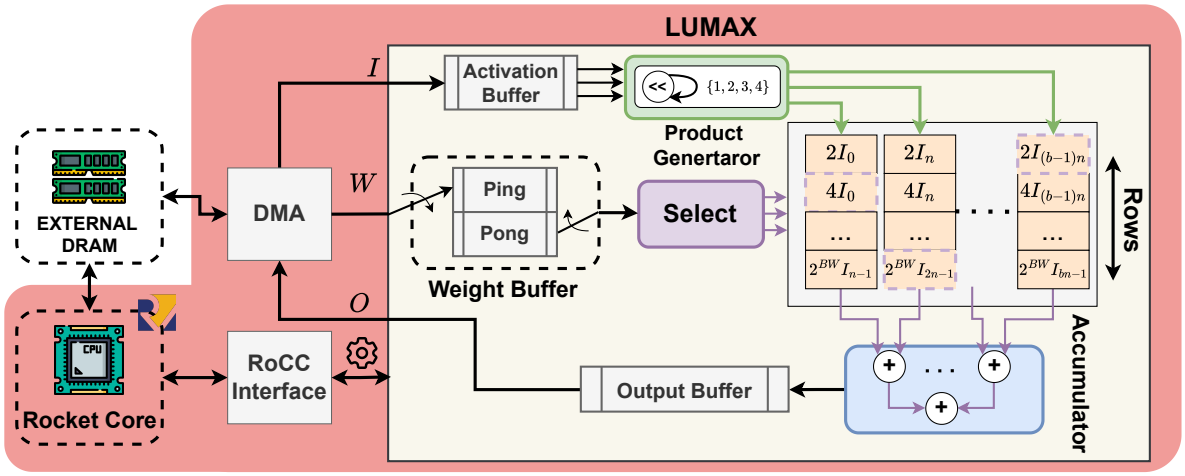


Figure 1.4.1: The Proposed LUMAX Architecture

Στο λογισμικό, παρέχεται API σε C με βασικές λειτουργίες *preload* (ρύθμιση παραμέτρων και φόρτωση βαρών), *start* (έναρξη υπολογισμού) και *fence* (αναμονή ολοκλήρωσης).

1.4.1 Σύνοψη του σχεδιασμού

Όπως αναφέρθηκε, ο επιταχυντής αποτελεί επέκταση της αρχιτεκτονικής RISC-V Rocket Chip ως RoCC επιταχυντής, με την επικοινωνία με τον επεξεργαστή να γίνεται μέσω προσαρμοσμένων εντολών. Ο επιταχυντής συνδέεται με την υπάρχουσα ιεραρχία μνήμης του συστήματος μέσω συνεκτικής διεπαφής TileLink, που υποστηρίζει ομαλή μεταφορά δεδομένων με κοινώς χρησιμοποιούμενη μνήμη. Για ελαχιστοποίηση καθυστέρησης και βελτιστοποίηση εύρους ζώνης, χρησιμοποιείται ελαφρύ DMA για οργάνωση μεγάλης κλίμακας μετακινήσεων δεδομένων μεταξύ κύριας μνήμης και εσωτερικών buffers

Η αρχιτεκτονική υποστηρίζει πολλαπλασιασμό πινάκων με μεταβλητή ακρίβεια ενεργοποιήσεων και βαρών, ορισμένη κατά το runtime. Χρησιμοποιούνται τεχνικές βασισμένες σε LUT για προ-υπολογισμό και αποθήκευση δυνατών γινομένων, τα οποία ανακτώνται μέσω δεικτοδότησης, αντικαθιστώντας κλασικούς πολλαπλασιαστές.

Το σύστημα περιλαμβάνει έναν **Rocket Core**, μία μονάδα **DMA** και τον **RoCC επιταχυντή**. Για αποθήκευση δεδομένων ενεργοποιήσεων, βαρών και εξόδου, χρησιμοποιούνται buffers: **Input Buffer** για προσωρινή αποθήκευση ενεργοποιήσεων, **Weight Buffer** για βάρη από παράθυρο ενεργοποιήσεων, και **Output Buffer** για συγκέντρωση αποτελεσμάτων, περιορίζοντας τις επιστροφές στον Rocket Core.

Για τα βάρη χρησιμοποιούνται **Ping-Pong Buffers** ώστε να γράφονται τιμές σε έναν buffer και να διαβάζονται ταυτόχρονα από άλλον για δεικτοδότηση. Ο **Product Generator** παράγει προϊόντα μέσω προσθέσεων, τα αποθηκεύει σε block μνήμες σειριακής ανάγνωσης, με καθυστέρηση ενός κύκλου.

Η μονάδα **Select Module** εντοπίζει στις μνήμες τα αντίστοιχα βάρη και υπολογίζει τη θέση των προϊόντων, ενώ ο **Accumulator Block** συλλέγει και αθροίζει όλα τα γινόμενα για τις ίδιες γραμμές εισόδου, αποθηκεύοντας το σωστό αποτέλεσμα στην έξοδο. Την κλιμάκωση των δεδομένων αναλαμβάνει η μονάδα **Scale Factor**.

Η επικοινωνία του Rocket Core με τον επιταχυντή γίνεται μέσω ειδικών RoCC εντολών που ελέγχουν τα σήματα, προκαταβάλλουν διευθύνσεις και διαχειρίζονται το ροή υπολογισμών.

Στο λογισμικό, το API σε C παρέχει βασικές συναρτήσεις: *preload* για ρύθμιση παραμέτρων και φόρτωση βαρών, *start* για έναρξη υπολογισμών, και *fence* για αναμονή ολοκλήρωσης.

1.4.2 Ροή Δεδομένων Επιταχυντή

Αυτός ο επιταχυντής χρησιμοποιεί τεχνικές βασισμένες σε Πίνακες Αναζήτησης (LUT), όπου ορισμένα προϊόντα προϋπολογίζονται και αποθηκεύονται, αποφεύγοντας πολλαπλασιασμούς σε πραγματικό χρόνο. Όμως, ο προϋπολογισμός όλων των δυνατών γινομένων για κάθε πιθανό συνδυασμό bitwidths είναι απαγορευτικός λόγω υπερβολικής χρήσης πόρων, ειδικά όταν οι ενεργοποιήσεις απαιτούν υψηλή ακρίβεια (π.χ. 16 ή 32 bits). Αυτό προκαλεί εκθετική αύξηση σε λογική και μνήμη.

Για τη μείωση αυτού του προβλήματος, ο επιταχυντής φορτώνει ένα παράθυρο ενεργοποιήσεων στην εσωτερική μνήμη και προϋπολογίζει περιορισμένο σύνολο μερικών γινομένων ανάλογα με το bitwidth των βαρών. Τα βάρη φορτώνονται κατά στήλες και χρησιμοποιούνται για την επιλογή των κατάλληλων προϋπολογισμένων προϊόντων για το τρέχον παράθυρο ενεργοποιήσεων. Κάθε βάρος επιλέγει μία ή περισσότερες τιμές που αποθηκεύονται και συσσωρεύονται στα κατάλληλα στοιχεία εξόδου.

Η λειτουργία μοντελοποιείται ως πολλαπλασιασμός πινάκων όπου οι είσοδοι είναι πίνακας διαστάσεων $R_{in} \times C_{in}$, πολλαπλασιαζόμενος με βάρη μεγέθους $C_{in} \times C_{out}$. Τα παράθυρα επεξεργασίας ορίζονται από τις παραμέτρους **XS** και **YS**, που καθορίζουν πόσα στοιχεία ενεργοποίησης φορτώνονται ανά διάνυσμα και πόσες σειρές επεξεργάζονται ταυτόχρονα, αντίστοιχα.

Ο επιταχυντής εκτελεί πέντε βασικά στάδια (αν και στην υλοποίηση μπορούν να εκτελούνται παράλληλα):

Ας περιγράψουμε τη ροή δεδομένων του υλικού με βάση τις προηγούμενες απλοποιήσεις και αναφορά στο Σχήμα 5.2.2. Αρχικά, διαβάζουμε ένα παράθυρο 2×2 στοιχείων ενεργοποίησης. Για κάθε στοιχείο, παράγουμε και αποθηκεύουμε στις μνήμες όλα τα πιθανά γινόμενα με βάρη της κατάλληλης ακρίβειας. Είναι εμφανές πως για χαμηλότερη ακρίβεια βαρών παράγονται λιγότερα προϊόντα.

Έπειτα, φορτώνουμε βάρη ανά στήλη, με κάθε βάρος να επιλέγει τις εκάστοτε αποθηκευμένες τιμές προκειμένου να επιλέξει το σωστό προϊόν. Στο παράδειγμα, φορτώνονται 2 βάρη ανά στήλη και πραγματοποιούνται 2 επιλογές προϊόντων από δύο διαφορετικά μπλοκ, όπου το κάθε μπλοκ αντιστοιχεί σε μία ενεργοποίηση.

Ξεκινάμε με τα δύο πρώτα βάρη της πρώτης στήλης και συνεχίζουμε στη δεύτερη στήλη διαβάζοντας βάρη για τις ήδη φορτωμένες ενεργοποιήσεις στην ίδια σειρά. Συνεχίζουμε στήλη-στήλη μέχρι να φτάσουμε στο τέλος. Μέχρι τότε, τα πρώτα δύο στοιχεία εξόδου έχουν συσσωρεύσει τιμές, αλλά δεν είναι πλήρως υπολογισμένα.

Στη συνέχεια, φέρνουμε το επόμενο παράθυρο ενεργοποιήσεων και επαναλαμβάνουμε τη διαδικασία. Όταν φτάσουμε στο τελευταίο παράθυρο των δύο πρώτων σειρών, διαβάζουμε τα τελευταία βάρη κάθε στήλης και εκτελούμε τις τελικές συσσωρεύσεις.

Οι δύο πρώτες γραμμές της εξόδου έχουν πλέον υπολογιστεί πλήρως. Αν υπάρχουν επιπλέον σειρές εισόδου, επαναλαμβάνουμε τη διαδικασία για αυτές, με τα ίδια βάρη αλλά νέες ενεργοποιήσεις.

Γενικότερα, υποθέτουμε ότι κάθε μπλοκ μνήμης μπορεί να κρατήσει τα προϊόντα για N ενεργοποιήσεις, συνολικά $N \times$ `Y_slice` στοιχεία. Η παράμετρος `X_slice` σχετίζεται με το N και θα αναλυθεί αργότερα.

Για κάθε μπλοκ $N \times$ `Y_slice`, φορτώνουμε για κάθε στήλη βαρών C_{out} ακριβώς N βάρη από τις σειρές που καλύπτει το μπλοκ. Αυτό διασφαλίζει ότι όλα τα απαραίτητα βάρη εισάγονται για σωστή επιλογή και συσσώρευση των προϋπολογισμένων προϊόντων.

Ας εξηγήσουμε σύντομα τη ροή δεδομένων του υλικού βασισμένοι στις απλοποιήσεις και το Σχήμα 5.2.2. Φορτώνουμε ένα παράθυρο 2×2 ενεργοποιήσεων και παράγουμε όλα τα πιθανά γινόμενα με τα αντίστοιχα βάρη. Φορτώνουμε βάρη στήλη-στήλη, με κάθε βάρος να επιλέγει τα σωστά προϊόντα από τις μνήμες. Η διαδικασία συνεχίζεται στήλη-στήλη και παράθυρο-παράθυρο, με τα αποτελέσματα να συσσωρεύονται στα σωστά στοιχεία εξόδου. Η ροή γενικεύεται σε μπλοκ μνήμης που κρατούν μερικά προϊόντα για N ενεργοποιήσεις και εξασφαλίζει ότι όλα τα απαραίτητα βάρη φορτώνονται σωστά για την επιλογή και συσσώρευση των προϊόντων.

This section describes the most important hardware components of the accelerator and how they interconnect. Furthermore, it explains how different generator parameters modify the RTL of the

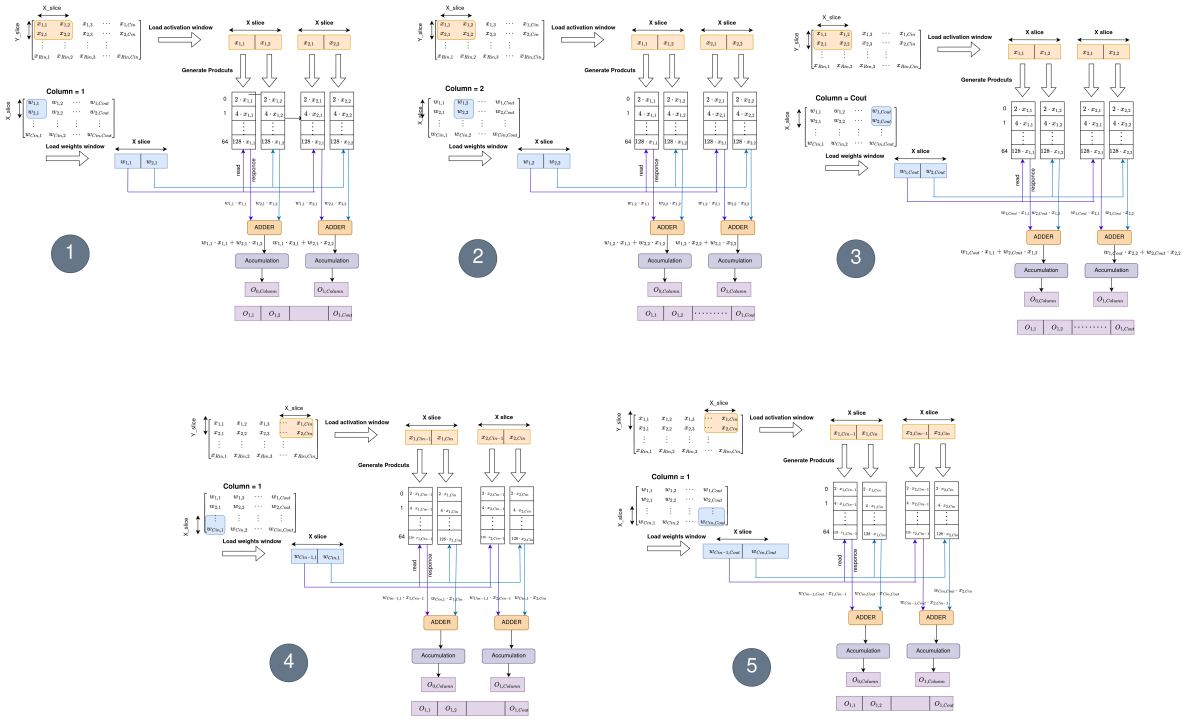


Figure 1.4.2: Simplified Example of the Hardware Dataflow

design, enabling flexible customization according to these parameters. In this way, developers can experiment with and evaluate various design trade-offs easily.

We will discuss the following key components of the accelerator architecture:

- Product generator component
- memory Blocks (On-chip memories)
- Select and accumulate component

These elements form the core of the hardware structure and determine how data is stored, transferred, computed, and accumulated within the accelerator.

1.4.3 Παραγωγή Προϊόντων

Ένα σημαντικό στοιχείο είναι ο *Product Generator*, ο οποίος παράγει τις δυνατές τιμές για κάθε ενεργοποίηση μέσα στο παράθυρο ενεργοποιήσεων. Όπως έχει ήδη αναφερθεί, αυτές οι τιμές αποθηκεύονται σε μνήμες με συγχρονισμένη ανάγνωση (Sync Read Mems). Εδώ περιγράφουμε πώς παράγονται. Τα στοιχεία αναπαρίστανται ως προσημασμένοι ακέραιοι.

Για να παραχθούν όλα τα πιθανά προϊόντα μιας ενεργοποίησης, υπολογίζονται προ-υπολογισμένα όλα τα δυνατά αποτελέσματα πολλαπλασιασμού με κάθε πιθανή τιμή βάρους, σύμφωνα με το bit width των βαρών.

Εφαρμόζονται οι ακόλουθες βελτιστοποιήσεις:

1. **Απόλυτα προϊόντα:** Παραγόμενες είναι μόνο οι απόλυτες τιμές των γινομένων, δηλαδή πολλαπλασιάζεται το απόλυτο της ενεργοποίησης με το απόλυτο όλων των πιθανών βαρών. Επειδή τα προσημασμένα γινόμενα είναι συμμετρικά ως προς το μηδέν (π.χ. 8×2 και $8 \times (-2)$ έχουν ίση απόλυτη τιμή), με την αποθήκευση μόνο των απόλυτων μειώνεται κατά το ήμισυ ο αριθμός των

προϋπολογισμένων τιμών. Αργότερα, μέσω ειδικής λογικής υπολογίζεται το σωστό πρόσημο του τελικού προϊόντος, λαμβάνοντας υπόψη τα πρόσημα ενεργοποίησης και βάρους.

2. **Προϊόντα με άρτια βάρη:** Προϋπολογίζονται μόνο τα γινόμενα που προκύπτουν από ενεργοποίηση πολλαπλασιαζόμενη με άρτια βάρη. Δηλαδή, δεν υπολογίζονται όλα τα πιθανά ενεργοποίηση-βάρος προϊόντα, αλλά μόνο αυτά που αντιστοιχούν σε άρτια βάρη. Η μέθοδος αυτή μειώνει επιπλέον κατά το ήμισυ την ποσότητα των προϋπολογισμένων δεδομένων. Τα προϊόντα που αφορούν περιττά βάρη υπολογίζονται στη συνέχεια με ειδική λογική κατ' απαίτηση, ολοκληρώνοντας το πλήρες σύνολο προϊόντων.

Κατά τη φάση παραγωγής προϊόντων, υποβάλλεται σε επεξεργασία ενεργοποιητικό διάνυσμα N στοιχείων. Σε κάθε κύκλο, ειδικό υλικό για κάθε MEM παράγει ένα προϊόν με τεχνική shift-and-accumulate, που γράφεται σειριακά, γραμμή-γραμμή, στη σχετική μνήμη. Η διαδικασία συνεχίζεται μέχρι να αποθηκευτούν όλα τα προϊόντα. Αναλυτικά, για κάθε ενεργοποίηση του διανύσματος παράγονται όλα τα θετικά, άρτια γινόμενα σύμφωνα με το ρυθμισμένο bit-width βάρους. Ο συνολικός αριθμός των προϊόντων είναι $2^{W_{\text{width}}-2} \cdot N$, όπου W_{width} είναι το bit-width των βαρών.

1.4.4 Τμήματα Μνήμης

Οι **MEMs** είναι κρίσιμο στοιχείο του σχεδιασμού, καθώς αποθηκεύουν τις προϋπολογισμένες τιμές για κάθε ενεργοποίηση και καθορίζουν το μέγεθος του *παραθύρου ενεργοποιήσεων*, δηλαδή το πλήθος των ενεργοποιήσεων που επεξεργάζεται παράλληλα ο επιταχυντής.

Η μνήμη οργανώνεται βασισμένη σε τρεις παραμέτρους: **M**, **Y** και **Rows Factor (RF)**. Το M υποδηλώνει τον αριθμό των μνημών ανά διάνυσμα ενεργοποιήσεων, το Y τον αριθμό των διανυσμάτων που υποστηρίζονται ταυτόχρονα και ο συνολικός αριθμός μνημών είναι $M \times Y$, όπου Y μνήμες μοιράζονται την ίδια τιμή βάρους για ανάκτηση δεδομένων. Ο *Rows Factor* καθορίζει πόσες φορές πολλαπλασιάζεται η προεπιλεγμένη χωρητικότητα γραμμών κάθε μνήμης, επιτρέποντας αποθήκευση προϊόντων για πολλαπλές ενεργοποιήσεις, διατηρώντας ταυτόχρονα τη διάταξη που καθορίζεται από το μέγιστο bit-width βάρους.

Τα προϊόντα κάθε ενεργοποίησης, ή **activation blocks**, διανέμονται ισομερώς στις M μνήμες. Εάν το σύνολο των blocks είναι μικρότερο από τη συνολική χωρητικότητα, αφήνεται κενός χώρος για διευκόλυνση παράλληλων αναγνώσεων. Αν γεμίσουν πλήρως, κάθε μνήμη αποθηκεύει όσα χωράει, μεγιστοποιώντας το μέγεθος παραθύρου ενεργοποιήσεων και την αποδοτικότητα των μνημών.

Οι εγγραφές ανά block εκμεταλλεύονται συμμετρία δεδομένων: αποθηκεύονται μόνο μη αρνητικές άρτιες τιμές, οι περιττές τιμές βαρών αντιμετωπίζονται με προσαρμογή στη κοντινότερη μικρότερη άρτια τιμή και διόρθωση μέσω τελεστή πρόσημου, ενώ οι μηδενικές τιμές θεωρούνται μηδενικές χωρίς ανάκληση μνήμης. Με αυτόν τον τρόπο μειώνονται οι απαιτήσεις μνήμης ανά ενεργοποίηση κατά το ένα τέταρτο.

Ένα ακόμη optimization αφορά το πλάτος κάθε γραμμής μνήμης, ώστε να χωρά το γινόμενο της μέγιστης ενεργοποίησης επί το μέγιστο βάρος. Όταν χρησιμοποιείται χαμηλότερη ακρίβεια ενεργοποιήσεων, το πλάτος της γραμμής δεν γεμίζει ολόκληρο, οπότε διαιρείται σε τμήματα ώστε να χωρούν πολλαπλά activation blocks στη σειρά, αυξάνοντας την εκμετάλλευση διάρκειας και το μέγεθος παραθύρου.

Έστω $I_{\max}$ και $W_{\max}$ τα μέγιστα bit-width για ενεργοποιήσεις και βάρη, και I_{bits}, W_{bits} τα τρέχοντα bit-widths. Η χωρητικότητα των ενεργοποιήσεων που μπορεί να αποθηκεύσει μία μνήμη δίνεται από τον τύπο:

$$\frac{I_{\max}}{I_{\text{bits}}} \cdot 2^{(W_{\max}-W_{\text{bits}})} \cdot \text{Rows Factor},$$

ο οποίος πολλαπλασιάζεται με M, τον αριθμό ενεργοποιήσεων ανά διάνυσμα. Η παράμετρος Y αυξάνει τον συνολικό απαιτούμενο χώρο μνήμης, αλλά επιτρέπει παράλληλη επεξεργασία πολλαπλών διανυσμάτων ενεργοποιήσεων και επαναχρησιμοποίηση βαρών, ιδανικό για πολλαπλασιασμούς πινάκων.

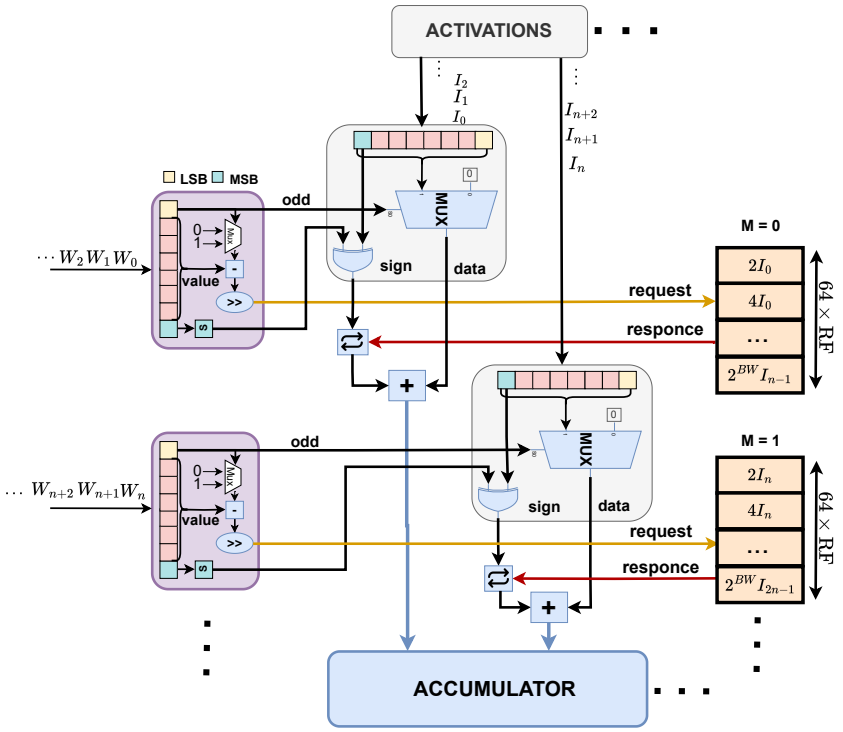


Figure 1.4.4: Detailed Representation of the Select Module for Y =1

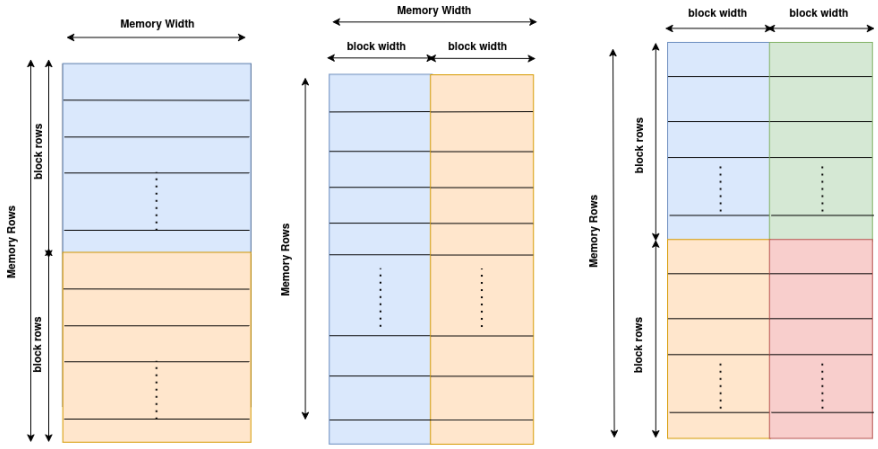


Figure 1.4.3: Mapping of activation blocks onto physical memory blocks, where each color corresponds to a different activation block.

1.4.5 Επιλογή Προϊόντος - Συσσώρευση

Όπως φαίνεται στο Σχήμα **??**, το module επιλογής τροφοδοτείται με βάρη και έχει την ευθύνη να αντιστοιχίσει τη σωστή γραμμή μνήμης στα MEMs όπου αποθηκεύεται το επιθυμητό προϊόν. Μια αναλυτική αναπαράσταση του module παρουσιάζεται και στο Σχήμα 1.4.4.

Κάθε MEM απαιτεί ένα module επιλογής που λαμβάνει ως είσοδο το βάρος που αντιστοιχεί στο τρέχον activation block και το χρησιμοποιεί ως δείκτη για τον εντοπισμό του προϊόντος. Αυτή η διαδικασία χωρίζεται σε τρία στάδια:

Πρώτα, το λιγότερο σημαντικό bit (LSB) του βάρους χρησιμοποιείται για να αποφασιστεί αν το βάρος

είναι περιττό, επειδή αποθηκεύονται μόνο προϊόντα για άρτια βάρη. Αν το βάρος είναι περιττό, επιλέγεται η κοντινότερη μικρότερη άρτια τιμή μειώνοντας το βάρος κατά ένα, διαφορετικά το βάρος χρησιμοποιείται ως έχει. Στη συνέχεια, το επιλεγμένο βάρος μετατρέπεται σε σωστό δείκτη αποθήκευσης μέσω δεξιάς μετατόπισης (shift right). Τέλος, για περιττά βάρη, οι ενεργοποιήσεις προστίθενται στο επιλεγμένο προϊόν ώστε να προκύψει η σωστή τελική τιμή.

Το δεύτερο στάδιο είναι η επιλογή της γραμμής μνήμης, που εκτελείται από τα modules επιλογής (με μοβ χρώμα στο Σχήμα **??**). Αυτά καθορίζουν τη σωστή γραμμή μέσα στο activation block, χρησιμοποιώντας έναν καταμετρητή offset που δείχνει την πρώτη γραμμή του τρέχοντος ενεργοποιητικού block εντός της μνήμης. Ο καταμετρητής αυτός είναι κοινός για όλα τα MEMs και αυξάνεται σε κάθε κύκλο επιλογής.

Τέλος, το πρόσημο του προϊόντος προκύπτει από το XOR των πιο σημαντικών bit (MSB) του βάρους και της ενεργοποίησης, καθώς αποθηκεύονται μόνο απόλυτες τιμές. Αν και η προσθήκη του αθροιστή για περιττά προϊόντα αυξάνει το υλικό ελάχιστα, η εξοικονόμηση χώρου αποθήκευσης που προκύπτει από τη μείωση κατά το ήμισυ των αποθηκευμένων προϊόντων είναι σημαντική.

Μόλις προσδιοριστεί η στοχευόμενη γραμμή μνήμης, αποστέλλεται αίτημα στην μνήμη χρονικής ανάγνωσης (synchronous memory) και η απόκριση γίνεται διαθέσιμη μετά από έναν κύκλο. Για τη διαχείριση αυτής της καθυστέρησης, εισάγονται καταχωρητές καθυστέρησης (delay registers) στα αντίστοιχα σήματα ελέγχου ώστε να παραμένουν συγχρονισμένα με τα επιστρεφόμενα δεδομένα. Το σχήμα αυτό εφαρμόζεται ομοιόμορφα σε κάθε MEM χωρίς να προσθέτει καθυστέρηση pipeline, καθώς το σύστημα είναι πλήρως pipeline.

Μετά τη δημιουργία των σημάτων επιλογής και πρόσημου, τα προ-επεξεργασμένα δεδομένα προωθούνται στον αθροιστή, ο οποίος χρησιμοποιεί δένδρο αθροιστών (adder tree) για να παράγει το τελικό αποτέλεσμα, όπως φαίνεται με μπλε χρώμα στα Σχήματα 5.1.1 και 1.4.4.

1.4.6 Βελτιστοποιήσεις Σχεδιασμού

Σε αυτό το κεφάλαιο περιγράφουμε περαιτέρω σημαντικές βελτιστοποιήσεις που εφαρμόζονται στον σχεδιασμό του επιταχυντή μας με στόχο την αύξηση της απόδοσης. Συγκεκριμένα, συζητάμε:

- τη σωλήνωση *επιλογής και συσσώρευσης* (Select-and-Accumulate Pipeline),
- την τεχνική *Ping-Pong Buffer* για τη φόρτωση βαρών (Load Weights Ping-Pong Buffer Technique),
- και την *αύξηση του μεγέθους της μνήμης* (Increasing Memory Size).

Σωλήνωση Επιλογής και Συσσώρευσης

Όπως αναφέραμε νωρίτερα, σημαντικό στοιχείο του σχεδιασμού μας είναι η μονάδα *Επιλογής και Συσσώρευσης*, της οποίας η δομή αναλύθηκε στα Σχήματα 4.3.13 και 4.3.14 των προηγούμενων ενοτήτων. Η μονάδα αυτή λειτουργεί σε δύο φάσεις: η **Φάση Επιλογής**, υπεύθυνη για την προώθηση των στοιχείων των βαρών στις φυσικές μνήμες ώστε να επιλέγονται (αναγιγνώσκονται) τα σωστά προϊόντα, και η **Φάση Συσσώρευσης**, που αθροίζει αυτά τα προϊόντα και αποθηκεύει τα συσσωρευμένα αποτελέσματα στον καταχωρητή εξόδου.

Δεδομένου ότι χρησιμοποιούμε συγχρονισμένες μνήμες, υπάρχει καθυστέρηση ενός κύκλου όταν διαβάζουμε από αυτές. Για να αποφύγουμε απώλειες απόδοσης, σωληνώνουμε (pipeline) αυτές τις δύο φάσεις. Δηλαδή, αντί να εκδίδουμε ένα αίτημα ανάγνωσης και να παραμένουμε αδρανείς έναν κύκλο πριν τη συσσώρευση, επικαλύπτουμε τα στάδια αιτήματος και συσσώρευσης. Μετά την αρχική καθυστέρηση, το pipeline διατηρεί και τα δύο στάδια ενεργά σε κάθε επόμενο κύκλο.

Με τη σωλήνωση αυτών των δύο σταδίων, αποφεύγουμε την επαναλαμβανόμενη καθυστέρηση ενός κύκλου ανά ανάγνωση. Αντιθέτως, μετά τον πρώτο κύκλο, το σύστημα συνεχίζει να συσσωρεύει δεδομένα και ταυτόχρονα εκδίδει το επόμενο αίτημα ανάγνωσης, μεγιστοποιώντας την απόδοση.

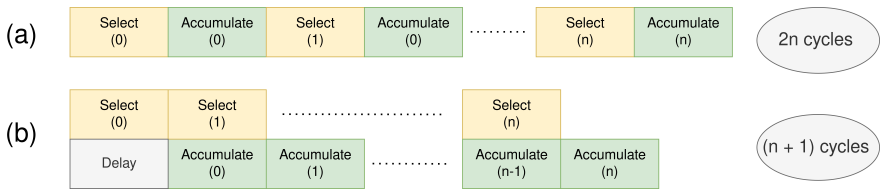


Figure 1.4.5: Select-and-Accumulate Phases: (a) Without Pipelining, (b) With Pipelining

Είναι σημαντικό να τονιστεί ότι αυτή η βελτιστοποίηση της σωλήνωσης **δεν** αυξάνει τους απαιτούμενους πόρους υλικού, καθώς σε κάθε κύκλο εκτελείται μία μόνο φάση Επιλογής και μία μόνο φάση Συσσώρευσης. Η μόνη αλλαγή είναι η προσθήκη λογικής ελέγχου για την επικαλυπτόμενη εκτέλεση των δύο φάσεων, με αποτέλεσμα να επαναχρησιμοποιείται το ίδιο υλικό μέσα στον ίδιο κύκλο. Έτσι, αν απαιτούνται να διαβαστούν n στοιχεία ανά μπλοκ, η συνολική καθυστέρηση μειώνεται από περίπου $2n$ κύκλους σε μόλις $n + 1$ κύκλους χάρη στη σωλήνωση.

Buffers Ping-Pong για τα Βάρη

Η βασική λειτουργία του επιταχυντή είναι να φορτώνει τα βάρη, να τα χρησιμοποιεί για δεικτοδότηση, να διαβάζει τα σωστά προϋπολογισμένα προϊόντα και να τα συσσωρεύει. Αυτά τα δύο στάδια περιγράφονται ως *Φόρτωση Βαρών* και *Επιλογή & Συσσώρευση*.

Κατ' αρχάς, πρέπει να φορτωθούν από την κύρια μνήμη μέσω DMA τα $\frac{N}{Y_S}$ βάρη που αντιστοιχούν σε μία στήλη της μήτρας βαρών. Αυτή η διαδικασία δεν ολοκληρώνεται σε έναν κύκλο, αλλά απαιτεί πολλούς κύκλους ανάλογα με το N, το Y_S, το τρέχον bitwidth των βαρών και τις παραμέτρους DMA. Μόνο αφού ολοκληρωθεί αυτή η φόρτωση, τα δεδομένα μπορούν να χρησιμοποιηθούν ως δείκτες στη φάση Επιλογής & Συσσώρευσης.

Ο αριθμός κύκλων που απαιτούνται για τη φόρτωση αυτών των βαρών στο buffer και ο αριθμός κύκλων για την πλήρη χρήση τους ως δείκτες δημιουργούν ένα κλασικό πρόβλημα παραγωγού-καταναλωτή. Σε αυτό το σενάριο, ο παραγωγός είναι η φάση Φόρτωσης Βαρών και ο καταναλωτής η φάση Επιλογής & Συσσώρευσης, με το Weight Buffer να λειτουργεί ως κοινός αποθηκευτικός χώρος.

Αν χρησιμοποιηθεί μόνο ένας Weight Buffer, η φάση Επιλογής & Συσσώρευσης πρέπει να ολοκληρώσει την κατανάλωση των δεδομένων πριν η φάση Φόρτωσης Βαρών μπορέσει να γεμίσει ξανά το buffer. Αυτό προκαλεί παύσεις και υποχρησιμοποίηση των πόρων υλικού. Για την επίλυση αυτού, χρησιμοποιούμε τη τεχνική **Buffers Ping-Pong** (επίσης γνωστή ως διπλό buffering), όπου δύο buffers, $W_{buffer}(0)$ και $W_{buffer}(1)$, λειτουργούν εναλλάξ. Όσο ο ένας buffer χρησιμοποιείται για ανάγνωση από τη φάση Επιλογής & Συσσώρευσης, ο άλλος γεμίζει ταυτόχρονα με νέα βάρη από τη φάση Φόρτωσης Βαρών.

Η τεχνική buffering ping-pong είναι καλά εδραιωμένη στους τομείς του υψηλής απόδοσης υπολογισμού και της επεξεργασίας σήματος, γιατί κρύβει την καθυστέρηση μεταφοράς δεδομένων και επιτρέπει συνεχή ροή δεδομένων χωρίς διακοπές. Ταιριάζει ιδανικά στον σχεδιασμό μας, όπου η ανάγνωση βαρών από μνήμη με μεταβλητή καθυστέρηση πρέπει να συνδυαστεί με ένα pipeline Επιλογής & Συσσώρευσης υψηλής ροής.

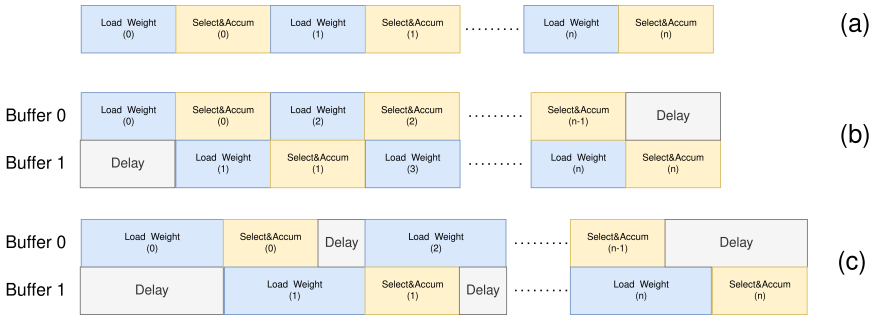


Figure 1.4.6: Load Weights and Select & Accumulate phases: (a) without double buffering, (b) with double buffering where Load Weights and Select & Accumulate require the same number of cycles, (c) with double buffering where the Load Weights stage needs more cycles than the Select & Accumulate stage.

Από την άποψη της απόδοσης, είναι σημαντικό να κατανοήσουμε ότι οι κύκλοι που απαιτούνται για τη φόρτωση των βαρών (c_1) και οι κύκλοι που απαιτούνται για τη φάση Επιλογής & Συσσώρευσης (c_2) είναι γενικά διαφορετικοί και εξαρτώνται τόσο από τις παραμέτρους του γεννήτρια όσο και από τη χρονική ρύθμιση bitwidth. Αν σημειώσουμε με C_{out} τον αριθμό των παραθύρων ενεργοποίησης που πρέπει να επεξεργαστούν, τότε σε σύστημα χωρίς διπλό buffer ο συνολικός χρόνος εκτέλεσης θα είναι

$$C_{\text{out}} \times (c_1 + c_2),$$

ενώ με διπλό buffering μειώνεται σε

$$C_{\text{out}} \times \max(c_1, c_2).$$

Επομένως, είναι κρίσιμο, όπως θα αναλυθεί περαιτέρω, ο λόγος c_1/c_2 να παραμένει όσο το δυνατόν πιο κοντά στη μονάδα. Όταν $c_1 > c_2$, ο επιταχυντής χαρακτηρίζεται ουσιαστικά ως *περιορισμένος από τη μνήμη* (memory-bound), ενώ όταν $c_2 > c_1$ είναι *περιορισμένος από την επιλογή* (Select-bound), που στην πράξη σημαίνει περιορισμός από τις συγχρονισμένες αναγνώσεις μνήμης. Συνεπώς, από άποψη βελτιστοποίησης κύκλων, είναι πολύ σημαντικό να ισορροπηθούν σωστά αυτές οι δύο φάσεις.

1.5 Αξιολόγηση και Αποτελέσματα

1.5.1 Πειραματική Ρύθμιση

Όπως περιγράφηκε προηγουμένως, ο επιταχυντής LUMAX κατασκευάστηκε ως RocketChip Co-processor (RoCC), που σημαίνει ότι είναι υλοποιημένος ως σύστημα SoC με το RocketChip ως κύριο επεξεργαστή και επικοινωνεί με τον LUMAX μέσω ιδιόκτητου πρωτοκόλλου (TileLink). Το σύνολο του SoC μαζί με τον επιταχυντή LUMAX προσομοιώνεται στην πλατφόρμα ZCU106 βασισμένη σε Zynq[31]. Η υλοποίηση DMA του LUMAX χρησιμοποιεί την εξωτερική DRAM του ZCU106 για την ανάκτηση των εισερχόμενων ενεργοποιήσεων και βαρών και παρέχει έξοδο. Η σύνθεση του υλικού και η ανάλυση ισχύος πραγματοποιήθηκαν με το εργαλείο Vivado 2022.1. Οι σχεδιασμοί αξιολογήθηκαν με διαφορετικό αριθμό ενεργοποιήσεων που φορτώνονται ανά διάνυσμα ενεργοποίησης ($MEM \in \{4, 8, 16, 32\}$) και αριθμό γραμμών ανά MEM (Row Factor, $RF \in \{1, 8\}$). Οι δύο αυτές παράμετροι είναι κρίσιμες για τη λειτουργία του σχεδιασμού, καθώς το μέγεθος του διανύσματος ενεργοποίησης εξαρτάται τόσο από το MEM όσο και το RF. Το MEM αναπαριστά τον αριθμό των ξεχωριστών μνημών, ενώ το RF είναι το μέγεθος κάθε μνήμης. Το συνολικό μέγεθος όλων των μνημών καθορίζει τον διαθέσιμο χώρο για τα προϊόντα και, σε συνδυασμό με το τρέχον bit-width βάρους, πόσες ενεργοποιήσεις μπορούν να χωρέσουν, ενώ το MEM καθορίζει επίσης πόσα στοιχεία μπορούν να επιλεγούν και να συσσωρευτούν ανά κύκλο.

Ο Πίνακας 1.3 συνοψίζει το χώρο σχεδιασμού για την εξερεύνηση της αρχιτεκτονικής LUMAX, με σκοπό να καταλήξουμε στους καλύτερους συνδυασμούς που μπορούμε να πετύχουμε σε σύγκριση με το προηγμένο έργο του επιταχυντή Gemmini[2], ο οποίος χρησιμοποιείται και ως βάση αναφοράς.

Table 1.3: Design Space Definition

Parameter	Values
MEM	4, 8, 16, 32
Row Factor	1, 8

Ως βάση για τη συγκριτική μας μελέτη ορίσαμε το προηγμένο έργο του επιταχυντή Gemmini [2]. Συγκεκριμένα, για δίκαιες συγκρίσεις, υλοποιήσαμε τις διαμορφώσεις του Gemmini [2] με ισοπόρους πόρους σε σχέση με τον LUMAX. Οι υλοποιήσεις Gemmini έχουν επίσης τα ισοδύναμα χαρακτηριστικά διεπαφής με τον LUMAX, όπως εύρος ζώνης δεδομένων και ελεγκτές DMA.

Οι μετρήσεις ισχύος και επιτάχυνσης αναφέρονται σε σχέση με αυτή τη βάση, και οι μετρημένοι κύκλοι αντιστοιχούν στην εκτέλεση όλων των λειτουργιών GEMM (πολλαπλασιασμός διανύσματος-πίνακα στην περίπτωσή μας) στην υλοποίηση υψηλού επιπέδου LLaMA2[32] με χρήση του dataset TinyStories-15M[34]. Η υλοποίηση LLaMa2 έχει μεταγλωττιστεί στον επεξεργαστή RocketChip, με ενσωμάτωση intrinsics για κατάλληλη επικοινωνία με τον LUMAX.

1.5.2 Πειραματικά Αποτελέσματα

Table 1.4: Power gains and Speedup comparison to Gemmini 4x4 PEs for different bitwidths

Config MEM-RF	Power Gain	Cycles Speedup Per Config (I_{width}, W_{width})					
		(16,8)	(16,4)	(16,2)	(8,8)	(8,4)	(8,2)
4-1	0.291	0.2	1.7	1.73	0.4	1.7	1.82
8-1	0.317	0.4	2.35	3.56	0.73	2.29	3.6
16-1	0.329	0.72	2.36	4.56	1.17	2.4	4.63
32-1	0.402	1.21	2.38	4.56	1.21	2.4	4.64
4-8	0.329	1.05	1.79	1.83	1.13	1.80	1.81
8-8	0.382	1.12	2.35	3.55	1.17	2.38	3.6
16-8	0.462	1.15	2.37	4.56	1.21	2.4	4.63
32-8	0.683	1.2	2.37	4.56	1.22	2.4	4.73

Σε Πίνακα 6.4 παρουσιάζουμε την αξιολόγηση του LUMAX υπό διάφορες διαμορφώσεις μνήμης, όπου ο αριθμός των MEMs (MEM) και ο Παράγοντας Γραμμής (RF) καθορίζουν το πραγματικό μέγεθος των buffer σαντιοπάδων. Τα αποτελέσματα συγκρίνονται με την προεπιλεγμένη διαμόρφωση Gemmini με μια συστοιχία systolic 4×4 PE που υποστηρίζει ενεργοποίηση και βάρη INT8, διατηρώντας σταθερές τις παραμέτρους επικοινωνίας και τη διαμόρφωση DMA για όλα τα σχέδια. Όπως φαίνεται, η κατανάλωση ισχύος αυξάνεται τόσο με τον αριθμό των μνημών όσο και με τον Παράγοντα Γραμμής, καθώς περισσότερες μνήμες απαιτούν πρόσθετη λογική και επιτρέπουν περισσότερες παράλληλες λειτουργίες, ενώ ένας υψηλότερος Παράγοντας Γραμμής απαιτεί μεγαλύτερη χωρητικότητα αποθήκευσης στον σαντιοπάδο. Παρ' όλ' αυτά, ακόμα και στις μεγαλύτερες δοκιμασμένες διαμορφώσεις, η κατανάλωση ισχύος παραμένει σταθερά χαμηλότερη από του Gemmini, κυμαινόμενη περίπου από 0.3× έως 0.6×. Αυτό αναδεικνύει την ενεργειακή αποδοτικότητα του LUMAX σε ένα ευρύ σχεδιαστικό φάσμα.

Όσον αφορά την απόδοση, οι μετρήσεις κύκλων παρουσιάζουν πιο σύνθετη συμπεριφορά. Όπως επίσης φαίνεται στην εξίσωση (1), το μέγεθος του παραθύρου ενεργοποίησης — και επομένως ο αριθμός των στοιχείων ενεργοποίησης που μπορεί να επεξεργαστεί παράλληλα το LUMAX — εξαρτάται από το MEM×RF και το πλάτος bit των βαρών, συμπεριφορά συμβατή με την πειραματική μας αξιολόγηση.

Συνεχίζοντας την ανάλυση με την επίδραση του πλάτους των bit των βαρών, δείχνουμε ότι τα υψηλότερα πλάτη bit (π.χ. 8 bit) μειώνουν την απόδοση λόγω του ότι απαιτούνται περισσότεροι κύκλοι για την παραγωγή μεγαλύτερου αριθμού γινομένων, ενός περιορισμένου παραθύρου ενεργοποίησης που περιορίζει τον παραλληλισμό και αυξημένων μεταφορών DMA, ενώ τα χαμηλότερα πλάτη bit (π.χ. 2 bit) επιφέρουν σημαντικές επιταχύνσεις (έως και 4.73×). Η αύξηση του Παράγοντα Γραμμής (RF) σε σταθερό αριθμό

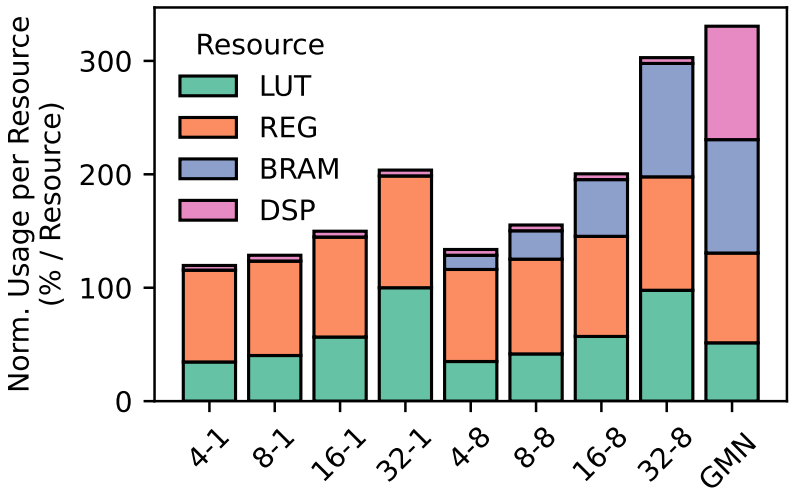


Figure 1.5.1: Normalized utilization of resources on the Xilinx ZCU106 FPGA for different M-RF configurations of LUMAX and the Gemmini 4×4 (GMN) baseline. Each resource type is normalized with respect to its maximum across all designs, enabling a fair comparison of resource usage distribution rather than absolute counts.

MEMs (M) βελτιώνει την επιτάχυνση διευρύνοντας το διάνυσμα ενεργοποίησης, επιτρέποντας περισσότερες γραμμές ανά μπλοκ για την αποθήκευση των γινομένων ενεργοποίησης· αυτή η ωφέλεια είναι πιο εμφανής για βάρους υψηλής ακρίβειας (π.χ. 16×8, επιτάχυνση από 0.2 σε 1.05 όταν RF = 1 έως 8, M = 4), ενώ τα βάρη χαμηλής ακρίβειας (π.χ. 8×2) παρουσιάζουν ελάχιστη βελτίωση (1.81–1.82) καθώς σε αυτές τις περιπτώσεις το παράθυρο ενεργοποίησης έχει ήδη κλιμακωθεί. Η αύξηση των MEMs σε σταθερό RF βελτιώνει σταθερά την επιτάχυνση για όλα τα πλάτη bit των βαρών, επιτρέποντας την επιλογή πολλαπλών στοιχείων ανά κύκλο αντί να περιμένει κύκλο-κύκλο για να μετακινηθεί σε όλα τα μπλοκ ενεργοποίησης. Από την άλλη, αυτή η επίδραση μειώνεται για βάρη χαμηλής ακρίβειας ή μεγάλο RF (π.χ. RF = 8), όπου το παράθυρο ενεργοποίησης κλιμακώνεται σε βαθμό που έχουμε περιπτώσεις περιορισμένες από τη μνήμη. Όταν περαιτέρω αυξήσεις του MEM ή του RF δεν βελτιώνουν πλέον την επιτάχυνση, ο σχεδιασμός γίνεται περιορισμένος από τη μνήμη, που σημαίνει ότι τα στάδια επιλογής και αθροίσεων καταναλώνουν περισσότερα βάρη ανά κύκλο από όσα μπορεί να παρέχει το DMA, οπότε επιπρόσθετη επιτάχυνση απαιτεί μεγαλύτερο εύρος ζώνης μνήμης ή ταχύτερη ανάκτηση βαρών

Στον Πίνακα 6.4, διαφορετικά χρώματα επισημαίνουν τα όρια κλιμάκωσης. Τα πράσινα κελιά δείχνουν περιπτώσεις όπου η αύξηση μόνο του Παράγοντα Γραμμής βελτιώνει την επιτάχυνση λόγω μεγαλύτερων παραθύρων ενεργοποίησης, ιδιαίτερα για βάρη 8-bit, όπου το παράθυρο ενεργοποίησης διαφορετικά θα συρρικνωνόταν επειδή απαιτούνται περισσότερα γινόμενα ανά ενεργοποίηση. Τα πορτοκαλί κελιά δείχνουν ότι η αύξηση μόνο του Παράγοντα Γραμμής δεν είναι πλέον αρκετά αποτελεσματική, και χρειάζονται επιπλέον MEMs για να επιτραπεί περισσότερες επιλογές ανά κύκλο στο στάδιο επιλογής και αθροίσματος. Τα μπλε κελιά αντιστοιχούν σε σχεδιασμούς περιορισμένους από τη μνήμη, όπου περαιτέρω επιτάχυνση περιορίζεται από την επικοινωνία DMA, παρόλο που το παράθυρο ενεργοποίησης είναι μεγάλο και υπάρχουν διαθέσιμα πολλαπλά MEMs.

Όσον αφορά την αποδοτικότητα πόρων, το Σχήμα 6.2.2 παρουσιάζει την αξιοποίηση πόρων FPGA του σχεδίου LUMAX σε διαφορετικές διαμορφώσεις στην πλακέτα ZCU106, σε σύγκριση με την υλοποίηση Gemmini. Αρχικά, το LUMAX χρησιμοποιεί ελάχιστα DSP, μόνο 8 DSP ανεξαρτήτως διαμόρφωσης, σε αντίθεση με το Gemmini που καταναλώνει 197 DSP ακόμα και για μια σχετικά μικρή συστοιχία systolic. Επομένως, η έμφαση στον σχεδιασμό μας δίνεται στα LUTs και τα BRAMs, που είναι οι βασικοί πόροι που χρησιμοποιούνται.

Οι τέσσερις πρώτες μπάρες στο Σχήμα 6.2.2 αντιστοιχούν σε διαμορφώσεις με Παράγοντα Γραμμής = 1, όπου κάθε μνήμη έχει μόνο 64 γραμμές. Κατά συνέπεια, η μνήμη μπορεί να υλοποιηθεί εξολοκλήρου

με LUTs αντί για BRAMs. Αντίθετα, οι επόμενες τέσσερις μπάρες αντιστοιχούν σε Παράγοντα Γραμμής = 8, όπου κάθε μνήμη έχει 512 γραμμές. Σε αυτή την περίπτωση, χρησιμοποιούνται BRAMs για την αποθήκευση τόσο των γινομένων όσο και των buffer ενεργοποίησης και βαρών, καθώς το μέγεθος του παραθύρου ενεργοποίησης αυξάνεται και απαιτείται μεγαλύτερη αποθήκευση εντός του chip για τον υπολογισμό ενός παραθύρου ενεργοποίησης.

Σε αμφότερες τις περιπτώσεις, η αύξηση του αριθμού των MEMs οδηγεί σε μεγαλύτερη χρήση LUTs, καθώς απαιτείται πολλαπλή λογική για κάθε MEM ώστε να υλοποιηθεί τόσο ο γεννήτορας γινομένων όσο και το στάδιο επιλογής και αθροίσματος για παράλληλη δεικτοδότηση και ανάκτηση γινομένων. Οι καταχωρητές αυξάνονται επίσης αναλογικά, αποθηκεύοντας προσωρινές τιμές και διατηρώντας την συγχρονικότητα μεταξύ των πολλαπλών MEMs.

Συνολικά, καθώς αυξάνεται ο αριθμός των MEMs, η χρήση των LUTs και των καταχωρητών προσεγγίζει τα επίπεδα των υπολοίπων πόρων του Gemmini (εκτός των DSPs), αλλά ο σχεδιασμός μας επιτυγχάνει αυτήν την απόδοση με βελτιωμένη απόδοση και χαμηλότερη κατανάλωση ισχύος.

Chapter 2

Introduction

Nowadays, Deep Neural Networks (DNNs) have become an integral part of many emerging research activities, challenging modern system architects for continuous adaptation and design optimizations of faster and more accurate solutions [1, 2, 3, 4, 5, 6]. Following this explosion, modern research approaches investigate solutions that differ from the simple CPU execution model[7], by focusing either on GPU-based solutions or on even more specific accelerators, i.e. the Tensor Processing Units (TPUs) or Neural Processing Units (NPUs) [8, 9, 10, 7].

Modern edge deployments of LLM-based [11, 12, 13, 14] AI applicatios are pushing even more the energy efficiency envelope. Methods like NN quantization [15, 16, 17, 10, 18] have emerged to represent the weights and/or the inputs of certain layers with fewer bits than the fully floating point representation, saving both power, area, and computational resources. Hardware specialization through aggressive design techniques like LUT-based multiplication [19, 20, 21] are utilized to further optimize the accelerators in terms of Performance-Power-Area (PPA) characteristics. Thus, we are witnessing a growing interest in domain-specific accelerators that exploit reduced numerical precision in data representations. Such designs achieve significant performance improvements, energy efficiency, and better utilization of hardware resources [1, 15, 22, 23, 24, 25, 26, 27]. In contrast, general-purpose accelerators that primarily support high-precision computations are not well-suited for large language models (LLMs) that rely on low-bitwidth inference [21, 29].

Modern LLMs, typically featuring high-precision activations combined with low-precision weights, have motivated the design of novel accelerator architectures. These accelerators aim to efficiently support **mixed-precision** operations while maximizing data reuse [1, 15, 22]. Among the most promising design techniques are **LUT-based methods**. This is mainly because when some models try to operate in lower bitwidths, then the gains of utilizing LUT-based fetching come as a great optimization compared with doing the multiplication operation itself. In this approach, accelerators either store in tables the complete product space for low-precision activations and weights [16], or maintain a reduced set of values for a specific activation window performing table precomputation on-the-fly for each LUT unit. The corresponding weights are then fetched and combined to generate the final outputs. This methodology is particularly effective only when the weights are quantized at lower precision, while activations remain at higher precision e.g. 8–16 bits [19, 21]. The primary disadvantage of previous implementations [19, 21] is its limited scalability in performance gains as weight bit-precision increases because the use of bit serial architecture. In this work, we present LUMAX, a LUT-based GeMM accelerator that supports mixed-precision inputs and weights, targeting the LLM domain and providing a low-energy and flexible solution for inferring modern NN models on the edge.

In contrast to prior LUT-based accelerators, which precompute and store combinations of activations and then rely on bit-serial logic and shift–accumulate steps to reconstruct products, LUMAX adopts a fundamentally different storage strategy. For each activation value, LUMAX directly stores all possible products according to weight precision, organizing them and grouping them into memory

blocks(MEMs). During execution, the weight simply serves as an index to select the corresponding product in a single cycle, enabling concurrent accumulation without the need for iterative bit-serial operations. This design not only reduces latency but also improves scalability for mixed-precision workloads.

Moreover, this accelerator framework is capable of dynamically reconfiguring the input and weights precision, providing a flexible environment, which can fit many of the modern novel quantized LLM applications. The accelerator is being built as a RocketChip Co-processor (RoCC), using a RISC-V core as its host processor. Furthermore, a LUT-based multiplication approach further optimizes the energy efficiency of our accelerator by exploring indexing certain weight/input features, instead of computing them. Architecture-specific optimizations are also built upon the acceleration framework, such as weights pre-fetching and data compression.

A LUMAX-based System-on-Chip was prototyped using Chipyard [30], mapped to the ZCU106 [31]. Different micro-architectural configurations of LUMAX were explored, showing reductions in LUT and DSP utilization up to 33% and 96% respectively, when compared to its iso-resource counterpart of TPU-based SoC based on Gemini accelerator architecture [2]. Using the LLaMA2 [32] network as our driver application, up to 4.7× cycle reduction and 70% improved energy efficiency was achieved compared to prior GeMM accelerators.

Chapter 3

Related Work

3.1 Dedicated Accelerators for Quantized Neural Networks

3.1.1 Gemmini: Systolic Architecture for GEMM

Systolic arrays are a popular architecture for accelerating matrix multiplication, a fundamental operation in deep neural networks (DNNs). Their structure is based on a grid of processing elements that operate in parallel and in synchronization, transferring data across the array in a pipelined manner. This architecture enables high resource utilization and reduces the need for off-chip memory access, significantly improving energy efficiency and performance.

The use of systolic arrays in DNNs is ideal, as matrix multiplications are repetitive and can be effectively parallelized. Additionally, their topology facilitates the execution of multiple multiply-and-accumulate operations with low latency.

However, systolic arrays also present several limitations. Their implementation typically relies on multipliers and DSP units, which are energy-intensive and limited in availability — especially on resource-constrained FPGA platforms. Furthermore, their fixed topology hinders flexibility in adapting to different matrix sizes or data formats, and scaling up to larger configurations may increase power consumption and area overhead significantly.

Gemmini [2] is an open-source matrix multiplication accelerator based on a systolic array architecture, developed within the Chipyard ecosystem. It supports two main operational modes: Output Stationary (OS) and Weight Stationary (WS), which define how data is held within the array elements during the multiply-accumulate operations. In OS mode, output values are held stationary in the compute elements, while in WS mode, weights remain fixed throughout computation, reducing data movement overhead.

Gemmini supports both quantized (int and uint) and floating-point (fp32) arithmetic representations, making it versatile for a wide range of neural network applications — from quantized DNNs to full-precision models. Its int8 support, in particular, makes it well-suited for fast and energy-efficient inference on quantized networks.

Gemmini is a highly flexible accelerator based on a systolic array architecture. It supports both Output Stationary (OS) and Weight Stationary (WS) dataflows, with the ability to switch between them at runtime, allowing it to efficiently adapt to different types of deep learning workloads. The arithmetic precision is configurable, supporting `int8`, `int16`, `int32`, as well as their unsigned counterparts (`uint8`, `uint16`, `uint32`), and optionally floating-point numbers. Beyond basic matrix multiplication, Gemmini supports key deep learning operations including ReLU and ReLU6 activations, as well as max pooling and average pooling. Matrix and vector transposition are also supported natively through dedicated hardware paths. The accelerator integrates tightly with the RISC-V instruction set through custom

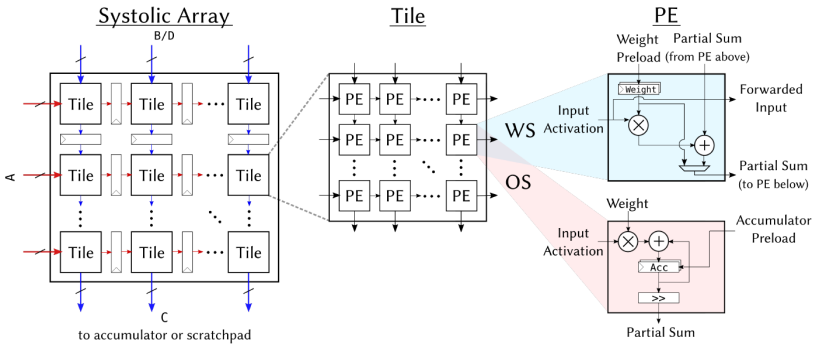


Figure 3.1.1: Gemmini Systolic Array.

instructions, enabling streamlined software-hardware interaction.

Although Gemmini offers high performance and integrates seamlessly with RISC-V processors via the Rocket Custom Coprocessor (RoCC) interface, it relies heavily on DSP units and multipliers. This reliance may limit efficiency in low-cost or resource-limited systems, such as small FPGAs. In contrast, alternative approaches, such as LUT-based accelerators, aim to reduce power consumption and hardware complexity by avoiding multipliers altogether.

3.1.2 Carat: Accelerator Architecture with Multiplier-Free GEMMs

Carat [22] is an innovative accelerator architecture designed specifically for general matrix multiplication (GEMM) operations. It introduces a multiplier-free approach that significantly reduces energy consumption and hardware complexity by eliminating traditional multiplication units. Instead, Carat transforms multiplications into additive operations using a novel technique known as *value-level parallelism* (VLP).

The key idea behind VLP is to exploit the redundancy in input values—especially prevalent in low-precision formats such as `INT4`, `FP8`, or `BF16`. Rather than performing a separate multiplication for every operand pair, Carat computes each unique product only once. Repeated operands are then linked to these products through a form of *temporal coding*, where timing signals indicate when a previously computed product should be reused. For instance, in a matrix of `INT4` inputs, there may be only 16 unique products out of thousands of multiplications, leading to substantial computational savings.

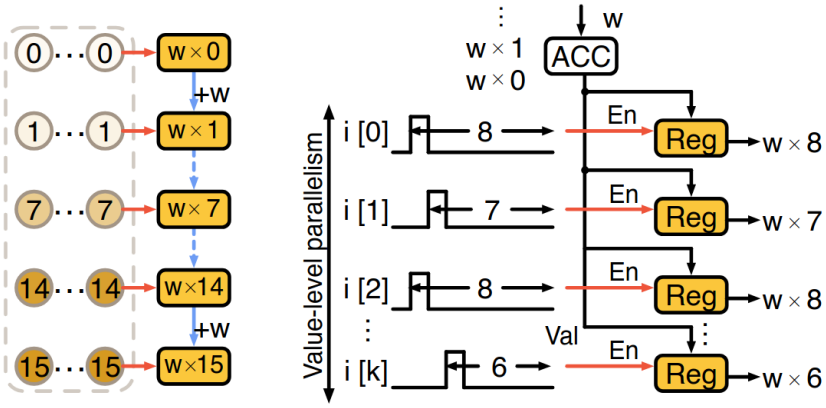


Figure 3.1.2: Illustration of value-level parallelism.

To implement this principle, Carat dynamically generates partial products during processing and associates them with input values using *temporal signals*. These signals act as temporal subscriptions

that determine when a product should be accessed and combined with corresponding data. This reuse at the value level enables Carat to process large volumes of data with significantly fewer multiplications.

Architecturally, Carat adopts a structure inspired by systolic arrays, where Processing Elements (PEs) are arranged in a 2D tile-based grid. These tiles operate on input tensors and communicate through broadcast and synchronization mechanisms that ensure coherent reuse of partial results. The architecture is scalable across nodes through a mesh-based Network-on-Chip (NoC), making it suitable for high-throughput and distributed workloads.

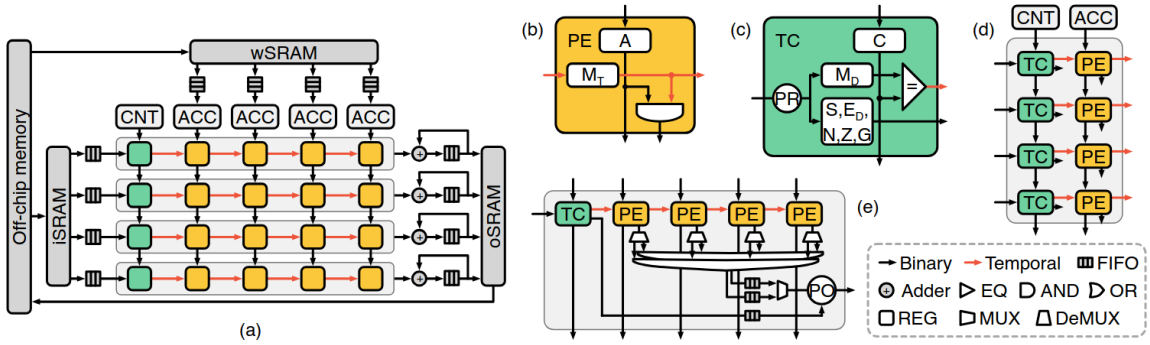


Figure 3.1.3: Carat accelerator architecture overview.

Carat also incorporates pipelining mechanisms that allow for concurrent processing of input data and reuse of partial products. This design choice improves throughput and allows multiple computation stages to proceed simultaneously, further reducing idle cycles and boosting overall efficiency.

The architecture is particularly optimized for low-precision, quantized inference tasks in deep neural networks (DNNs), where energy efficiency and performance per watt are crucial. Empirical evaluations report a throughput improvement of 1.02× to 3.2× and energy efficiency gains ranging from 1.06× to 4.3× compared to conventional systolic array accelerators.

Advantages. Carat's tile-based architecture is inherently scalable, supporting both single-node and multi-node deployments. The use of temporal coding and value reuse reduces redundant computation and improves energy efficiency. Additional benefits include minimized switching activity, efficient hardware utilization, and high compatibility with quantized DNN workloads.

Limitations. Despite its strengths, Carat presents several challenges. Its scheduling and control logic are relatively complex, introducing design overhead. The architecture is limited to GEMM operations and does not natively support non-linear layers such as activations or pooling. Its performance is sensitive to input data patterns, and improper workload alignment can degrade efficiency. Finally, managing multiple temporal signals may introduce energy and synchronization overheads that must be carefully tuned during implementation.

3.1.3 TATAA – A Transformable Accelerator for Transformers

Transformer models, which form the computational backbone of modern Large Language Models (LLMs), require both linear and non-linear operations. While linear operations dominate the overall compute load, non-linear functions—such as Softmax and normalization—demand higher precision. However, most accelerators focus almost exclusively on GEMM operations, often neglecting non-linear computation.

TATAA [1] addresses this gap by introducing a runtime programmable accelerator architecture capable of efficiently executing both linear and non-linear operations within Transformer models. Its key

capabilities include.The TATAA accelerator is a flexible, unified hardware architecture designed to efficiently support transformer model computations, including both linear matrix multiplications and complex non-linear functions. Its core innovation is a runtime-configurable, dual-mode Processing Unit (DMPU) that can switch seamlessly between int8 matrix multiplication (MatMul) mode and bfloat16 non-linear processing mode. It achieves this flexibility through a customized instruction set architecture (ISA) and a compilation framework that maps high-level transformer operations into basic supported operations, enabling end-to-end transformer inference without retraining or extensive hardware modifications.

The system features a **programmable processing architecture** that allows flexible reuse of compute units. Processing units are interconnected to support both a *systolic array* for high-throughput GEMM and a *SIMD architecture* for non-linear operations. It includes hardware support for `int8` and `bfloat16` formats, along with integrated **on-chip quantization**, removing the need for external preprocessing.

At the heart of TATAA as we can see in figure 3.1.4 lies its dual-mode processing unit architecture. Each *Dual Mode Processing Unit (DMPU)* can dynamically switch between two execution modes, adapting to the operation type without requiring hardware reconfiguration:

- In `int8` **systolic mode**, the DMPUs form a $W \times 4N$ systolic array, where Processing Elements (PEs) perform multiply-accumulate (MAC) operations. These are optimized for high-throughput linear functions such as matrix multiplications. The interconnect is configured via a *Mode MUX*, enabling efficient data flow between rows of PEs.
- In `bfloat16` **SIMD mode**, each DMPU operates independently as a vector processing unit. The PEs are organized into pipelined stages (S0 to S3), leveraging the internal register file (`RFY`) to execute floating-point operations with higher precision—ideal for non-linear operations like activations or normalization. Up to $W \times N$ vector elements can be processed in parallel.

The transition between the two modes is performed dynamically at runtime via the *Mode MUX* and a lightweight control unit. This allows the accelerator to morph between linear-optimized systolic execution and flexible SIMD execution, depending on workload requirements—delivering both efficiency and versatility in transformer processing.

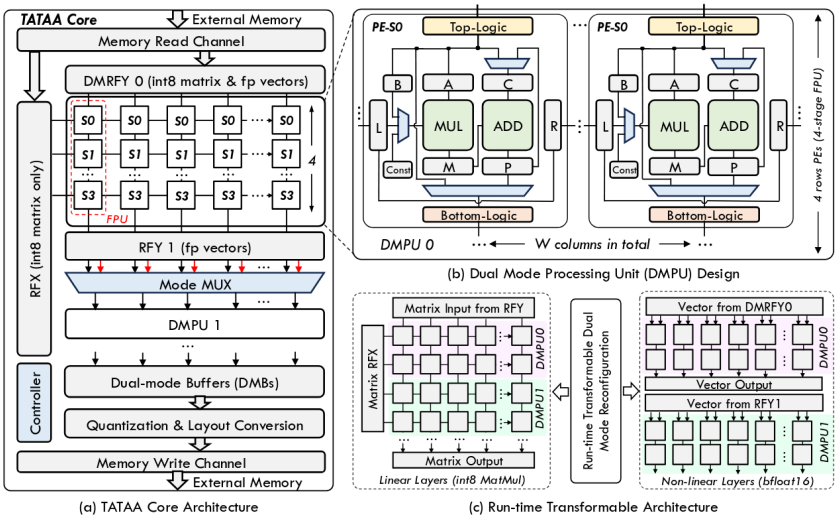


Figure 3.1.4: TATAA hardware architecture and dual-mode processing unit

In the TATAA architecture, support for non-linear functions within the SIMD mode using `bfloat16` is enabled through approximation techniques and specialized hardware. Functions like GELU, SoftMax, and LayerNorm are approximated using low-order polynomial or rational functions, which are efficiently mapped to integer operations. These approximations are executed via dedicated pipeline

stages and functional units, allowing seamless integration with linear computations. The use of high-precision `bfloat16` ensures sufficient accuracy, while the transformable arithmetic system and parallelism maintain high performance. This design enables efficient execution of complex non-linear functions, accelerating transformer models without compromising flexibility.

The TATAA accelerator offers several significant advantages that make it a compelling architecture for transformer model acceleration. Firstly, it supports both **linear and non-linear operations**, addressing a common limitation in existing accelerators that often prioritize only matrix computations. It offers **mixed-precision capabilities**, with support for both `int8` and `bfloat16` formats. In terms of performance, TATAA achieves **high throughput**, reaching up to **2935.2 GOPS** for `int8` operations and **169.8 GOPS** for `bfloat16` computations. A key innovation is its **transformable architecture**, which allows **dual-mode execution** within a unified processing pipeline. This design maintains **minimal accuracy loss**, typically ranging between **0.14% and 1.16%**, while remaining **power efficient**—delivering up to **2.19× better efficiency** compared to an RTX 4090 GPU. Furthermore, it includes a **custom Instruction Set Architecture (ISA)** tailored for efficient transformer execution.

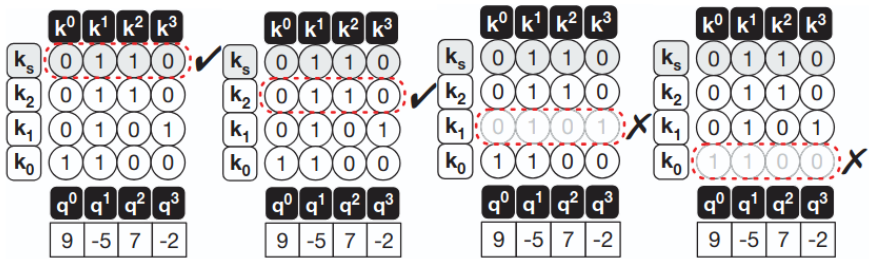
However, the architecture is not without drawbacks. One of the primary challenges is **compiler complexity**, particularly when mapping complex non-linear operations into the supported instruction primitives. Additionally, TATAA has **limited native support** for certain specialized functions, which could require software-level approximations or workarounds. There may also be **runtime overhead** due to frequent switching between execution modes, potentially affecting latency in tightly-coupled workloads. Lastly, while TATAA offers a flexible and general solution, **highly specialized hardware** might still outperform it in terms of **area efficiency or peak performance** for narrowly focused applications.

3.1.4 LeOPard – Gradient Based Learned Run time Pruning

In attention mechanisms used in large language models (LLMs) during inference, only a small subset of tokens highly correlates with the token under attention, and this subset is only determined at runtime. Therefore, a significant portion of the computations becomes inconsequential due to low attention scores. In self-attention layers, the main computational burden is associated with the score matrix calculation, $\text{Scores} = Q \cdot K^T$, and the computation of attention values, $\text{Atts} = P \cdot V$.

The main idea of LeOPArd is to accelerate transformer attention computations by integrating gradient-based learned runtime pruning and bit-level early compute termination techniques. This approach leverages gradient information to dynamically determine which parts of the attention scores can be pruned during inference, significantly reducing computational energy and latency without sacrificing accuracy. The hardware is designed to support efficient, adaptive, and energy-conscious processing of transformer models, especially focusing on multi-head self-attention mechanisms.

An innovative pruning strategy is applied using algorithmic advancements that enable the model to learn self-attention thresholds in a gradient-based fashion. This allows the model to be jointly fine-tuned for both parameter optimization and sparsity, ensuring minimal accuracy degradation while substantially reducing computational load. The approach removes unimportant computations—specifically targeting score calculations in the attention mechanism, where the score matrix ($\text{Scores} = Q \cdot K^T$) is computed. The Softmax function is applied only to the retained (non-pruned) score elements, further reducing workload. Moreover, the matrix multiplication in the attention output computation ($\text{Atts} = P \cdot V$) is performed using the pruned Softmax matrix P. This pruning strategy is implemented on a per-layer basis, allowing each layer to utilize a different learned threshold. Additionally, LeOPArd introduces the concept of *early-compute termination* as clear seen in Figure 3.1.5, which halts unnecessary computation at earlier stages—improving efficiency without compromising output quality.



Cycle	$\mathcal{P}$ = Partial Sum	$\mathcal{M}$ = Conservative Margin	Early Termination? ($\mathcal{T}h$ = 5)
1	$\mathcal{P}_1 = 0$	$\mathcal{M}_1 = (9+5)(2^{-1}+2^{-2}+2^{-3}) = 12.25$	$\mathcal{P}_1 + \mathcal{M}_1 = 12.25 \geq 5$; ✗
2	$\mathcal{P}_2 = \mathcal{P}_1 + (5-7)2^{-1} = -1$	$\mathcal{M}_2 = (9+5)(2^{-2}+2^{-3}) = 5.25$	$\mathcal{P}_2 + \mathcal{M}_2 = 4.25 < 5$; ✓
3	$\mathcal{P}_3 = \mathcal{P}_2 + (5-2)2^{-2} = -0.25$	$\mathcal{M}_3 = (9+5)(2^{-3}) = 1.75$	$\mathcal{P}_3 + \mathcal{M}_3 = 1.5 < 5$;
4	$\mathcal{P}_4 = \mathcal{P}_3 + (9+5)2^{-3} = 1.5$	$\mathcal{M}_4 = 0$	$\mathcal{P}_4 + \mathcal{M}_4 = 1.5 < 5$;

Figure 3.1.5: High-level overview of early-compute termination for dot-product operation Q × K . In this example, K is represented in bit-serial format, whereas Q is in full-precision fixed-point format. In Figure (a-d) each column illustrate one element of K vector and each row represents its corresponding bits (MSB → LSB). K indicates the sign bit. For simplicity, K elements are scaled to be between -1.0 and +1.0. The table shows the partial sum values after each cycle.

Every Cycle C:

1. **Partial Sum:**
$$P_c = P_{c-1} + (\vec{q} \cdot K_{\text{bits}}) \cdot 2^{-c} \tag{3.1.1}$$

2. **Conservative Margin:**
$$M_c = \text{Largest}(q_1 + q_2) \cdot \sum_{h=c}^{\text{BITS}-1} 2^{-h} \tag{3.1.2}$$

3. **Stopping Condition:**
After each cycle, calculate how much value could potentially contribute to the final sum. Stop at cycle C when:
$$P_c + M_c \leq \text{Threshold} \tag{3.1.3}$$

LeOPArd primarily supports bit-serial processing for the attention score computations ($Q \times K^T$) and value multiplications ($\cdot V$). The hardware operates with bit-level precision (e.g., 12-bit serial units), enabling fine-grained early termination of computations based on pruning decisions. The design targets transformer models such as BERT and Vision Transformers, which typically employ 16-bit floating point or lower-precision formats. However, LeOPArd's architecture is optimized for bit-serial arithmetic to efficiently implement both pruning and early stopping.

In this setup, activations and weights are represented using fixed-point quantization formats. The choice of 12-bit precision is motivated by compatibility with the hardware and the desired balance between computational accuracy and performance efficiency.

LeOPArd performs arithmetic bit-serially for greater flexibility and energy efficiency. Its core module, the **QK-DPU**, handles bit-serial multiplication with control logic that enables *early termination* when intermediate results fall below learned thresholds.

Weights and activations are stored in bit-serial buffers, and partial results are accumulated across cycles. Once a result is deemed sufficient, computation stops early to save energy.

As illustrated in Figure 3.1.6, the front-end **QK-PU** prunes low attention scores during $Q \times K^T$ computation by evaluating bit-serial partial results. Only scores exceeding thresholds are sent to the

back-end **V-PU** for Softmax and $P \cdot V$ processing. This pruning reduces unnecessary computation, improving both speed and energy efficiency.

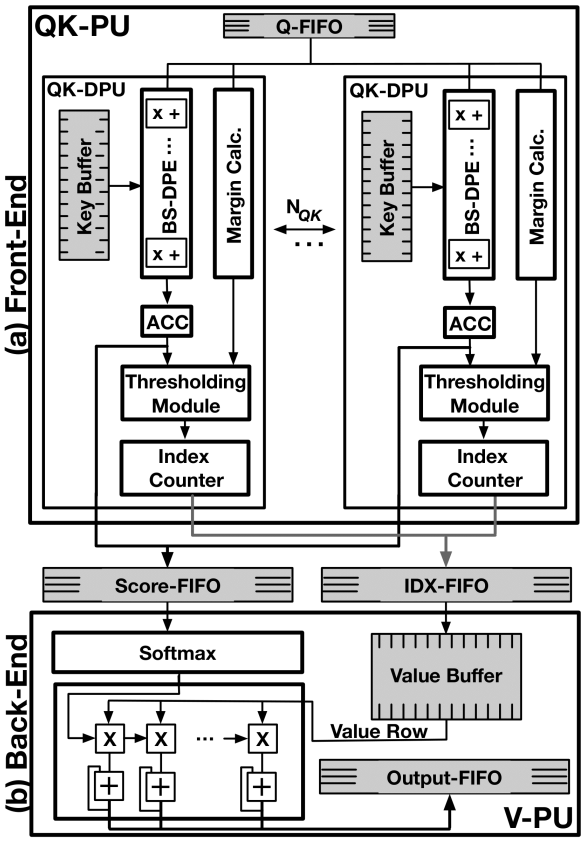


Figure 3.1.6: Overall microarchitecture of a LeOPArd tile

The system offers several advantages. It supports operations such as $Q \times K^T$, Softmax, and $P \times V$. Only the Q and V matrices are stored in on-chip buffers. It employs **runtime pruning** with different thresholds per layer. The design emphasizes **scalability** and high parallelism through a tile-based architecture. Overall, it achieves a speedup of 1.9 to 2.4 times compared to simple bit-serial methods, along with a 3.9 to 4.0 times reduction in energy consumption. On the downside, the system incurs a maximum accuracy loss of 2.2%. It requires approximately 15% increased area on the chip. There is limited generalizability for deep learning networks (DLNs), and performance depends on workload characteristics.

3.1.5 An Energy Efficient Soft SIMD Microarchitecture

Deploying quantized CNNs on edge devices requires flexible and energy-efficient hardware. Traditional SIMD architectures with fixed bitwidths limit performance and scalability. The proposed soft SIMD microarchitecture as illustrated in Figure 3.1.7 addresses this by supporting arbitrary bitwidths and efficient fixed-point operations, enabling high-performance inference under tight area and power constraints.

The architecture employs several key techniques: **guardbits** are used instead of multiplexers to separate SIMD subwords with minimal hardware cost; **CSD encoding** reduces the number of multiplication cycles; **runtime-configurable SIMD modes** allow arbitrary bitwidths (e.g., 3–24 bits); a **shift-add engine** enables efficient multiplications; and a **Data Pack Unit (DPU)** dynamically repacks data between SIMD formats for flexible computation.

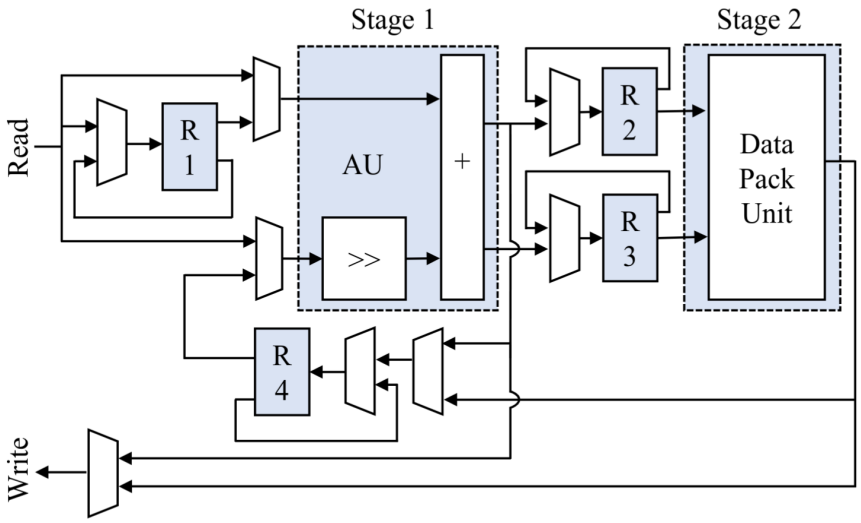


Figure 3.1.7: Block scheme of the proposed Soft SIMD microarchitecture. The first stage is an Arithmetic Unit (AU), and the second stage is a Data Pack Unit (DPU). R1 to R4 are registers.

The architecture supports **signed fixed-point integers** in **2's complement Q1.X format**, enabling efficient arithmetic with varying precision. It allows **arbitrary bitwidths** from **3 to 24 bits**, with tested configurations including 3-, 4-, 6-, 8-, 12-, 16-, and 24-bit subwords. Precision is **layer-dependent**, and experiments show **negligible errors** (e.g., $\tilde{0}.2\%$ for 8-bit multiplication), keeping overall CNN accuracy loss under **1%** compared to floating-point baselines.

The architecture primarily supports arithmetic-heavy operations essential for CNN inference, including element-wise addition and subtraction, shift operations, shift-add based multiplication, and multiply-accumulate (MAC) with overflow-safe accumulation using guardbits. It efficiently handles dense (fully-connected) and convolutional layers through variable bitwidth MACs but does not implement non-linear activations like ReLU directly in hardware, as these are usually managed by surrounding logic or fused layers.

It also support **eterogeneous quantization per layer** with weights and activaions can have at runtime differnt bitwiths even in the same CNN per layer.

In the following Table 3.1, we observe and compare the above architectures related to GEMM or LLM acceleration for quantized parameter representation.

Architecture	**Datatype / Precision**	**Linear Ops**	**ReLU**	**Softmax**	**Multipliers**	**Reconfigurable**
Gemmini	int8/16/32, uint, float	✓	✓	×	✓	WS / OS mode
Carat	FP8 (low precision)	✓	×	×	×	×
TATAA	int8 /bfloat16	✓	✓	✓	✓	SA / SIMD mode
LeOPArd	int12	✓	×	✓	×	×
Soft SIMD	Arbitrary bits (3–24) int	✓	×	×	×	Precision (bitwidths)

Table 3.1: Comparison of architectures and supported operations/features.

3.2 LUT-Based Accelerators for Mixed-Precision Quantized Neural Networks

So far, we have presented important and notable works in the field of accelerators for large language models (LLMs), which often leverage quantization techniques and span a wide range of architectures and optimization strategies. In the following, we focus on LUT-based (Lookup Table) accelerators, which are closely related to the architecture proposed in our work. These LUT-based approaches represent an interesting category, as they enable direct comparison and evaluation with our design, which introduces LUT-based GEMM operations.

3.2.1 FIGLUT: LUT-based Accelerator for FP-INT GEMM

FIGLUT [19] is an energy-efficient accelerator design specifically tailored for FP-INT (floating-point input, integer weight) General Matrix Multiplication (GEMM) operations using Look-Up Tables (LUTs). It aims to address the computational challenges of deploying large language models (LLMs) by reducing computational complexity through LUT-based data retrieval instead of traditional arithmetic operations. Many existing approaches decompress weights back to floating-point format before computation, missing the opportunity to exploit their low precision and resulting in reduced efficiency. Meanwhile, LUT-based implementations on GPUs often suffer from **bank conflicts** as shown in Figure 3.2.1 during memory access.

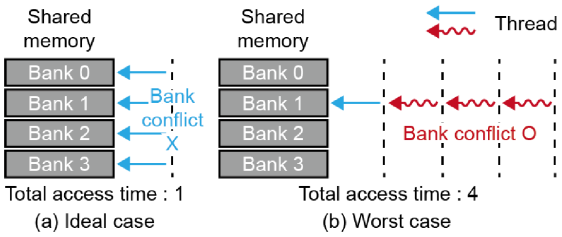


Figure 3.2.1: Comparison of bank conflicts during shared memory access

In FIGLUT, the activation elements are not quantized and remain in full precision (FP32), while the weights undergo aggressive quantization with a maximum bit-width of 4 bits. Ideally, the weights could be constrained to binary values $\{-1, 1\}$ to further simplify computations.

The architecture follows a **weight-stationary** approach, meaning that the neural network's weights are loaded into the processing units and remain fixed, or "stationary," in place for an extended period. Instead of moving the weights for each computation, the activations, which are the inputs to the network, are streamed through these stationary weights. This design choice is crucial because it allows the system to precompute all potential partial products between the fixed weights and the incoming activations. These precomputed values are then stored in Look-Up Tables (LUTs). Consequently, the actual runtime computation is significantly simplified to efficient LUT reads and subsequent accumulations, thereby drastically reducing computational complexity and energy consumption compared to traditional methods that would perform multiplications on the fly.

- Replaces multiply-accumulate (MAC) units with **Read Accumulate (RAC)** operations using precomputed values stored in a Look-Up Table (LUT).
- For each binary weight pattern of length μ, a **key** is generated to retrieve the corresponding sum of floating-point (FP) inputs.

In FIGLUT, model if weights are quantized into binary values 0 or 1. The architecture processes weights in **groups of size** μ. A **pattern of length** μ is a sequence of μ binary weights.

There are 2^{μ} possible patterns. As we can see, for example, $\mu = 3$ in Figure 3.2.2.

Binary Patterns	Key	Value
$\{-1,-1,-1\}$	0 (b'000)	$-x_1 - x_2 - x_3$
$\{-1,-1,+1\}$	1 (b'001)	$-x_1 - x_2 + x_3$
$\{-1,+1,-1\}$	2 (b'010)	$-x_1 + x_2 - x_3$
$\{-1,+1,+1\}$	3 (b'011)	$-x_1 + x_2 + x_3$
$\{+1,-1,-1\}$	4 (b'100)	$+x_1 - x_2 - x_3$
$\{+1,-1,+1\}$	5 (b'101)	$+x_1 - x_2 + x_3$
$\{+1,+1,-1\}$	6 (b'110)	$+x_1 + x_2 - x_3$
$\{+1,+1,+1\}$	7 (b'111)	$+x_1 + x_2 + x_3$

Figure 3.2.2: Example Of look-up Table when $\mu = 3$

Binary Coding Quantization (BCQ)

Binary Coding Quantization (BCQ) represents weights as a linear combination of binary basis vectors scaled by learnable coefficients. This reduces multiplications to simple additions and sign operations, enabling efficient execution. A constant offset can be added for accuracy. BCQ supports both *uniform* and *non-uniform* quantization, balancing low-precision efficiency with sufficient representational power, making it well-suited for energy-efficient accelerators.

LUT Memory Architecture with Flip-Flop LUTs (FFLUT)

The LUT memory architecture replaces register files with Flip-Flop LUTs (FFLUT), allowing conflict-free, parallel reads by multiple RAC units. Unlike conventional storage requiring arbitration, FFLUT offers simultaneous low-latency access, preserving throughput without added power or scheduling cost. This compact, table-based design matches quantized and LUT-driven computation, offering a scalable and energy-efficient memory solution.

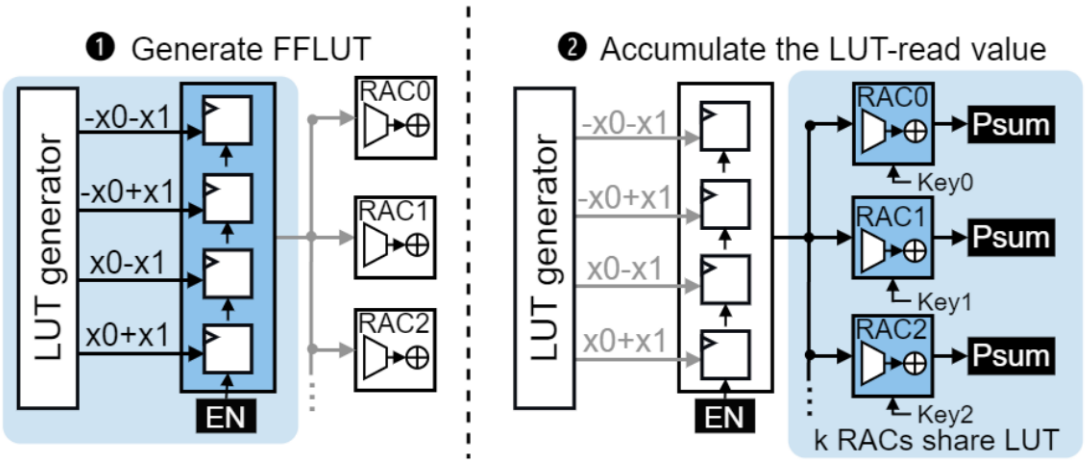


Figure 3.2.3: Architecture of the Flip-Flop based Look-Up Table (FFLUT)

Systolic Array-Based Architecture: A 2D systolic array inspired by Google's TPU employs a weight-stationary strategy for efficient FP input streaming as illustrated in Figure 3.2.4 , maximizing weight reuse, minimizing memory access, and enabling delay-free LUT value integration.

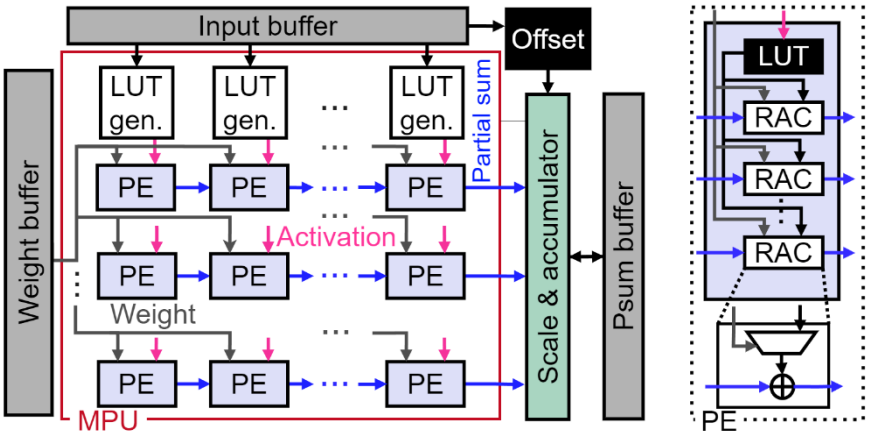


Figure 3.2.4: Overall MPU architecture of FIGLUT

FIGLUT offers several key advantages alongside a few important limitations. One of its most notable strengths is **high energy efficiency**, achieving up to 59% higher TOPS/W at 3-bit quantization and up to 98% improvement at 2.4-bit when compared to state-of-the-art solutions. It also provides **flexibility**, supporting a wide range of quantization schemes such as FP-INT, Binary Coding Quantization (BCQ), mixed precision, as well as both uniform and non-uniform quantization. Additionally, FIGLUT benefits from **parallelism**, enabled by its use of Flip-Flop LUTs (FFLUTs) which eliminate bank conflicts and support efficient systolic tiling and data reuse.

However, these advantages come with trade-offs. One major limitation is the **LUT memory overhead**, which grows exponentially with the pattern length parameter μ\mu, necessitating careful design choices. The architecture also introduces **hardware complexity**, particularly due to challenges in managing signal fan-out and routing, which complicates physical layout. Lastly, FIGLUT is a **specialized accelerator**, primarily optimized for weight-only models, and may not generalize well to broader computational workloads.

3.2.2 4-bit CNN Quantization with Compact LUT-Based Multipliers

The work of Zhao et al. (2023) [16] addresses the significant challenge of implementing convolutional neural networks (CNNs) on **resource-constrained embedded devices**, such as **FPGAs** that lack sufficient hardware multipliers (DSPs). Traditional CNN models demand substantial computational power and memory bandwidth, making them inefficient for deployment on edge devices (e.g., drones, industrial sensors, inspection systems).

The key challenges include:

1. How to reduce numerical precision without sacrificing accuracy.
2. How to implement efficient multiplication on FPGAs without using DSP blocks.

Initially, the implementation targets only *inference*, where the model is first trained using `fp32` precision, followed by the application of **Threshold-Aware Quantization (TAQ)** to both weights and activations. As a result, weights and activations are quantized to 4-bit integers (ranging from 0 to 15) [8, 33, 17].

Specifically, the authors propose a quantization method for CNNs using the **Threshold-Aware Quantization (TAQ)** technique. This method belongs to the category of **post-training quantization**, aiming for efficient deployment on FPGAs without retraining the model.

TAQ employs a non-uniform value mapping mechanism where weights and activations are transformed into 4-bit integers using **custom thresholds** to minimize quantization error. Additionally, a **mixed rounding strategy** is applied, alternating between methods such as round-nearest and floor, depending on proximity to thresholds and data distribution.

A critical and innovative aspect of the proposed architecture is the complete **elimination of arithmetic multipliers** in hardware. Instead of using traditional multipliers — such as the DSP blocks typically found in FPGAs — the implementation exploits the fact that both weights and activations are **quantized to 4-bit precision**, resulting in a **limited set of possible arithmetic combinations**.

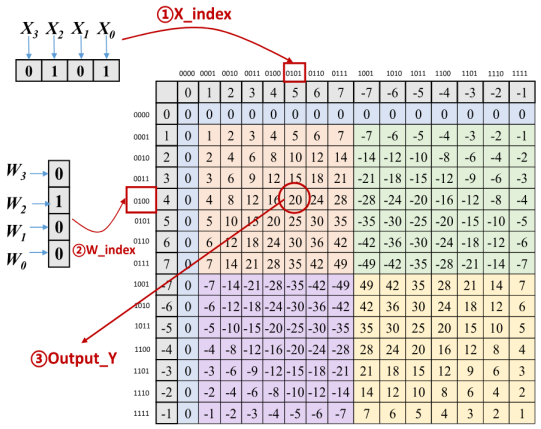


Figure 3.2.5: Basic CLM architecture for 4-bit multiplication.

Each input value can range from 0 to 15, leading to only $16 \times 16 = 256$ possible multiplication outcomes. The architecture takes advantage of this by constructing a **fully precomputed Look-Up Table (LUT)** that stores all possible products of any 4-bit activation and weight pair as shown in Figure 3.2.5.

This LUT is implemented on the FPGA using **LUT6 logic units**, which are fundamental six-input logic resources available in modern FPGAs. The final result is a **Compact LUT-based Multiplier (CLM)**, requiring only **13 LUT6 blocks per multiplier**, making it highly efficient in terms of both area and power consumption.

During inference, the CLM operates by taking a pair of 4-bit values (one activation and one weight) and using them as an address to **fetch the precomputed product from the LUT**. Thus, the arithmetic operation is fully replaced by a single, fast memory access as we clealry seen in Figure 3.2.6.

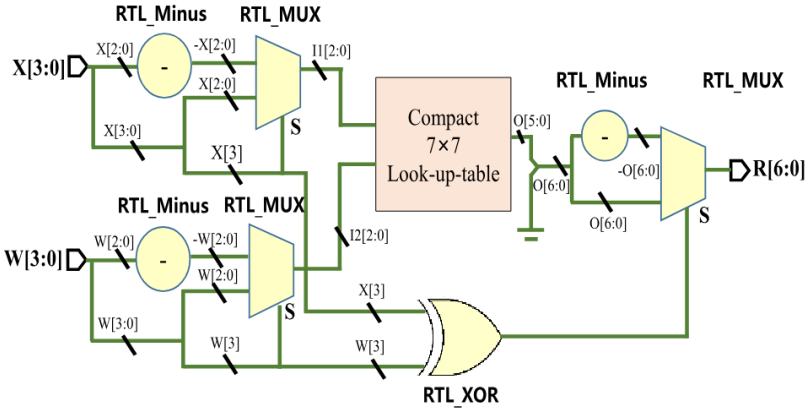


Figure 3.2.6: Peripheral RTL logic of the CLM implementation.

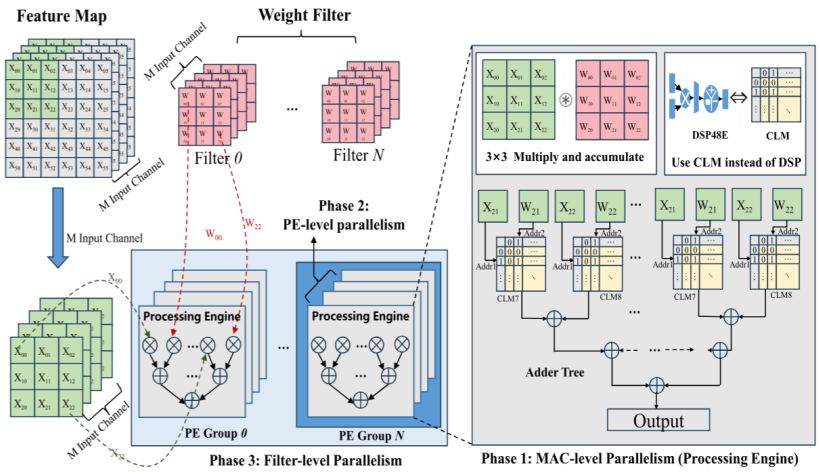


Figure 3.2.7: Multiple parallelism for convolution computation in FPGA accelerator.

This technique greatly reduces circuit complexity, eliminates the need for DSPs, and allows for **massive parallelization of multiplications**, which is particularly beneficial for CNN inference workloads. TThe complete architecture design is illustrated in Figure 3.2.7.

The CLM (Compact LUT-based Multiplier) units effectively replace traditional arithmetic multipliers such as DSP blocks. This means that for every multiplication that would normally require a DSP, a dedicated CLM unit is used instead. As a result, the architecture achieves **MAC-level parallelism**, since each multiplication is handled independently by its own CLM. For example, in the case of a typical 3×3 convolution window, we would need **9 CLM units** — one for each element in the window.

This parallel design eliminates data access conflicts, as each activation-weight pair is processed through its own CLM, which retrieves the precomputed product from a dedicated LUT. By allocating a separate LUT path for each activation window pair, the system avoids contention and ensures high-throughput, fully parallel inference execution.

Table 3.2: Advantages and Disadvantages of the TAQ Approach

Advantages	**Disadvantages**
(1) Improved performance in terms of **area** and **energy efficiency** through the use of LUT-based multipliers.	(1) Applicable **only to high quantization levels**, i.e., 4-bit precision for both weights and activations.
(2) **Lightweight implementation** suitable for **edge devices** with limited hardware resources (e.g., no DSPs).	(2) Supports **only 4-bit quantization**, limiting flexibility for other precision levels.

3.2.3 LUT Tensor Core

This work introduces the **LUT Tensor Core**, a hardware-software co-designed architecture tailored for efficient low-bit GEMM operations in LLM inference. It leverages precomputed **Lookup Tables (LUTs)** to replace traditional multipliers, especially useful in mixed-precision settings.

Conventional approaches to LUT-based GEMM face significant challenges, including inefficient dequantization logic, limited hardware support for mixed-precision operations, high storage and computation overhead for LUT tables, and suboptimal tiling strategies. Additionally, the lack of specialized instruction sets and compiler support further limits performance.

The proposed design addresses these limitations by supporting high-precision activations (e.g., FP16, INT8) and very low-bit weights (1–4 bits). It is **runtime reconfigurable**, allowing dynamic support for different precision combinations depending on the workload.

The core idea is to load a vector of activations and precompute all possible combination these activations. At runtime, weights bit by bit are used to indexers to select the appropriate partial sums from the LUT and via shift and accumuluate opeation combine them, avoiding real-time multiplication. These partial results are then accumulated into output elements, which remain **on-chip** until computation completes, following an **output-stationary** dataflow pattern.

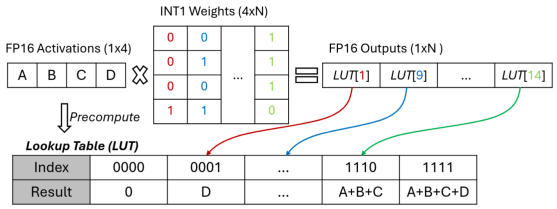


Figure 3.2.8: A naive LUT-based mpGEMM tile example of FP16 activations and INT1 weights. With the precomputed table, a table lookup can replace a dot product of 4-element vectors.

To reduce LUT storage overhead, the architecture exploits the **symmetry of weight values** as shown in Figure 3.2.9 . For instance, binary weights $\{0, 1\}$ can be reinterpreted as $\{-1, 1\}$, enabling a symmetric representation. This effectively halves the number of required LUT entries by eliminating redundant computations. Additionally, to support arbitrary weight bit-widths (1–4 bits), the design utilizes a **bit-serial architecture**. Each weight is processed across W_BIT cycles, allowing serialized computation of mixed-precision dot products. This approach balances flexibility and area efficiency without requiring dedicated logic for each bit-width combination.

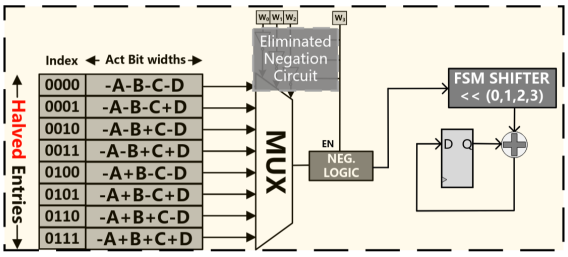


Figure 8: Optimized LUT unit with bit-serial.

Figure 3.2.9: Optimized LUT unit with bit-serial

Figure 3.2.10 illustrates the conventional three-step process for LUT-based mpGEMM: (1) table precomputation, (2) table lookup, and (3) partial sum accumulation. However, several limitations reduce the overall performance. First, precomputed tables require substantial storage, introducing area and latency overhead. Second, supporting multiple bit-width combinations (e.g., INT1/2/4 $\times$ FP16/FP8/INT8) increases complexity and chip area. Third, suboptimal LUT tiling shapes hinder table reuse and inflate storage costs. Finally, the lack of a dedicated instruction set and the mismatch with conventional compiler stacks make integration and efficient scheduling challenging.

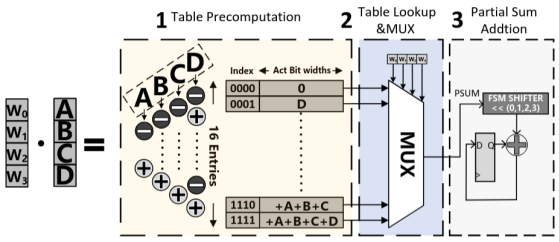


Figure 3.2.10: Conventional LUT hardware in three steps. Table precomputation and storage introduce heavy overhead.

The architecture loads a vector of K activations, requiring one table per activation vector. If M activation vectors are processed in parallel, M LUT tables are needed. Each table holds 2^{K-1} entries (with symmetry optimization), and these are accessed using MUX-based selection logic controlled by the binary weights.

- M: Number of activation rows → defines how many LUT tables are needed.
- N: Number of weights per activation → each LUT feeds N weight computations (MUX units).
- K: Number of binary weight bits → defines the LUT address width and MUX select lines (quantization depth).

Number of **binary weight groups (bit depth)** Defines the number of bits per weight group, i.e., the **quantization depth**. Each of the **K** bits is used to select from the LUT using a MUX. Figure 3.2.11 illustrates the data flow of the design along with the interpretation of its parameters.

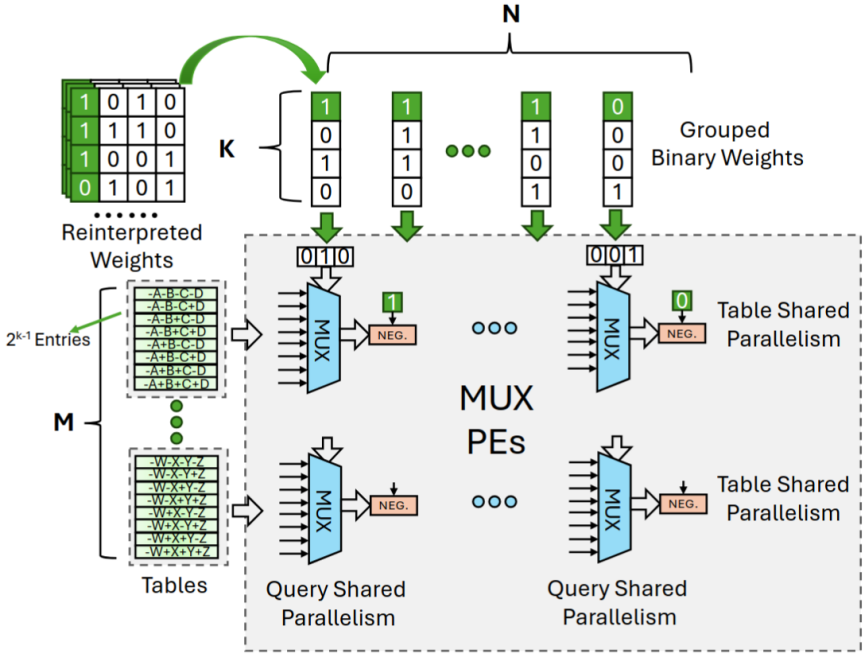


Figure 3.2.11: Elongated MNK tiling of LUT-based Tensor Core. LUT-based Tensor Core requires a larger N (e.g., 64/128) to maximize table reuse, along with a suitably sized K (e.g., 4) for a cost-efficient table size.

The LUT Tensor Core significantly outperforms existing software-based LUT implementations, achieving a 1.44× improvement in both compute density and energy efficiency compared to prior state-of-the-art LUT-based accelerators.

Despite their effectiveness [19, 21], this approach requires several cycles proportional to the weight bitwidth (W_width), making it prohibitive for higher-precision scenarios. Consequently, these implementations are typically restricted to 4-bit weights. Although they achieve favorable performance at low bit-widths, scaling to higher bit-widths remains limited, even if technically feasible.

Table 3.3 presents a quantitative comparison among all the related work observations with the LUMAX accelerator. In contrast, LUMAX proposes a less-complex yet fundamentally different approach. For each activation window, LUMAX generates, through successive shifts and accumulations, every possible product that can arise from combining the activation with all potential weight values, forming distinct activation blocks, with the number of elements in each block determined by the current weight bit-width. A key advantage of LUMAX approach is that the LUT space scales linearly with the size of the activation window, in contrast to previous implementations [19, 21] where LUT size scales exponentially. Furthermore, each activation maintains its products independently, such that W_width does not affect the cycles required to select values, which can be done in a single cycle. Although the activation window decreases for high W_width, the LUMAX design supports up to 8-bit weights, providing satisfactory performance for certain configurations. Consequently, this approach offers greater flexibility in scaling weight bit-widths compared to earlier LUT-based methods.

Table 3.3: **LUMAX compared to LUT-based accelerators**

	LUT Tensor [21]	**FIGLUT [19]**	**CLM [16]**	**LUMAX**
Act.	FP/INT8, FP/INT16	FP16	INT4	INT16,INT8
Wgt.	INT1–INT4	INT1–INT4	INT4	INT1–INT8
Bit-serial	✓	✓	×	×
Generate	✓	✓	×	✓
Stationary	Output	Weight	Weight	Output
Platform	ASIC	ASIC	FPGA	FPGA
LUT entries	$2^{(N-1)}$	$2^{(N-1)}$	$N \times 64$	$N \times 2^{(\mathrm{W\_width}-2)}$

$$\begin{cases} N = \text{number of elements in the activation window (i.e., how many activations are supported simultaneously)} \\ W_{\text{width}} = \text{current bitwidth (precision) of the weight element)} \end{cases}$$

Chapter 4

Background

4.0.1 The Chipyard Development Environment

Chipyard is an open-source hardware design framework [7] built on the *Chisel* hardware construction language [30]. It is specifically developed to support flexible and scalable design of RISC-V-based System-on-Chip (SoC) architectures. Chipyard combines various tools and libraries into a unified environment that facilitates rapid prototyping, simulation, synthesis, and software development.

- **Chisel-Based Design:** Built on Chisel, a powerful hardware description language embedded in Scala, enabling modular, parameterizable, and reusable hardware components.
- **RISC-V SoC Generation:** Supports the creation of complete SoC designs using generators like Rocket Chip and BOOM, with customizable cores and components.
- **Integrated Toolchain:** Chipyard bundles multiple tools into a consistent environment for:
  - *RTL Simulation:* Supports Verilator and commercial tools like Synopsys VCS.
  - *FPGA Emulation:* Accelerated simulation via FireSim on cloud FPGA instances.
  - *VLSI Flows:* Includes Hammer for physical design and layout automation.
  - *Software Build:* FireMarshal helps build bare-metal and Linux-based software workloads.
- **Hardware Acceleration:** Facilitates integration and testing of custom hardware accelerators (e.g., Hwacha, Gemmini), enabling software-hardware co-design.
- **Open-Source and Extensible:** Maintained by the Berkeley Architecture Research Group, Chipyard is actively developed and supports contributions from the open-source community.
- **Deep RISC-V Ecosystem Integration:** Seamlessly integrates with the broader RISC-V ecosystem, providing a flexible platform for research and experimentation.
- **FPGA Prototyping and Validation:** FireSim enables fast prototyping and system-level validation using cloud FPGA instances.
- **Co-Design Support:** Enables parallel hardware/software development, encouraging a holistic system-level design approach.

In summary, Chipyard serves as a comprehensive environment for designing, prototyping, and evaluating RISC-V SoCs, making it a powerful tool for both academic research and industrial hardware development.

4.0.2 Chisel: Constructing Hardware

Chisel is not a traditional high-level synthesis (HLS) language, nor is it a low-level hardware description language (HDL) like Verilog or VHDL. Instead, Chisel is a **constructive hardware description language**, embedded in Scala, that ultimately generates Verilog or SystemVerilog code for synthesis.

Chisel enables designers to describe digital hardware at a high level while preserving the structural and low-level control that traditional HDLs offer. It comes with a rich API, object-oriented constructs, and functional programming features thanks to Scala. When integrated with the `Chipyard` ecosystem, it becomes easy to plug in your own designs or modify existing ones, such as creating a custom RISC-V co-processor in just a few steps.

Chisel encourages component-based design, where each module (component) has well-defined inputs and outputs. Additionally, it supports wrapping existing Verilog modules using `BlackBox` wrappers, allowing for seamless reuse of legacy HDL code.

Chisel, like other digital design languages, works with binary signals that can be either `0` or `1`. These are often referred to as `low/high`, `false/true`, or `deasserted/asserted`. At the core of digital design are combinational circuits and registers (flip-flops), which Chisel supports through a variety of abstractions.

4.0.3 RocketChip

The RocketChip [35] is a RISC-V based core that was also the first hardware description project developed for the RISC-V ISA. It is a highly parameterizable open-source SoC generator capable of integrating a wide variety of cores and accelerators. The configuration of the RocketCore is highly modular, allowing the end-user to configure the cores, building blocks, and capabilities of the core. Taking advantage of this modularity, various application-specific accelerators can be constructed with different configurations. The RocketChip architecture can also provide reconfigurable extension blocks beyond the core, such as L1 and L2 caches, memory management unit (MMU), floating-point unit (FPU), vector execution units, debug unit, performance counters, and interrupt controllers, as well as communication infrastructure between all of these components.

The RocketTile represents the next level of configuration within RocketChip and allows tuning of all aspects of the core, such as L1 and L2 caches, MMU, FPU, debug unit, performance counters, and interrupt controller. The RocketTile is the basic building block of the RocketChip and is used to construct the final SoC. It can be configured for a single core, a multicore, or even a multi-cluster multiprocessor configuration. The main RocketTile configurations are:

- **BigCore**: A high-performance core with 16 KiB, 4-way set-associative instruction and data caches that supports FPU by default.
- **MediumCore**: A core with smaller 4 KiB direct-mapped caches that does not support FPU by default.
- **SmallCore**: A low-performance core with highly limited cache that does not support FPU by default.
- **TinyCore**: A less commonly used configuration that supports only 32-bit architecture.

4.0.4 RoCC Accelerator

The RoCC (Tightly-Coupled Rocket Custom Coprocessor) Accelerator is a custom accelerator tightly integrated with RISC-V cores in an SoC, offering high performance through close communication with the CPU. It features the following characteristics as represent in Figure 4.0.1 :

- **Communication via Custom Instructions:** RoCC accelerators are controlled using special instructions of the form *customX rd, rs1, rs2, funct,* where X (0–3) specifies the target accelerator

(a core may connect up to 4 RoCCs).The 7-bit `funct7` field enables the accelerator to differentiate between multiple operations.

- **Direct Access to CPU Resources:** RoCC shares resources with the RISC-V core, such as registers (rs1, rs2, rd) and the L1 cache. Through the mem interface, it can perform load/store operations directly to memory, while the ptw interface gives it access to the page-table walker for virtual memory management. It can also interrupt the CPU upon task completion using the interrupt signal.
- **System Integration:** RoCCs connect to the TileLink network through tlNode (direct connection to L1-L2 crossbar) or atlNode (connection via an arbiter).
- **Custom Toolchain:** Using RoCC requires a custom toolchain, as the customX instructions are not standardized in the RISC-V ISA. To invoke an accelerator operation, the programmer uses special macros that compile to customX instructions.

Comparison with MMIO Peripherals: While MMIO peripherals communicate through memory-mapped registers (requiring only standard toolchains), RoCC accelerators offer lower latency and tighter integration with the CPU, but require expertise in both hardware (Chisel) and software (macros).

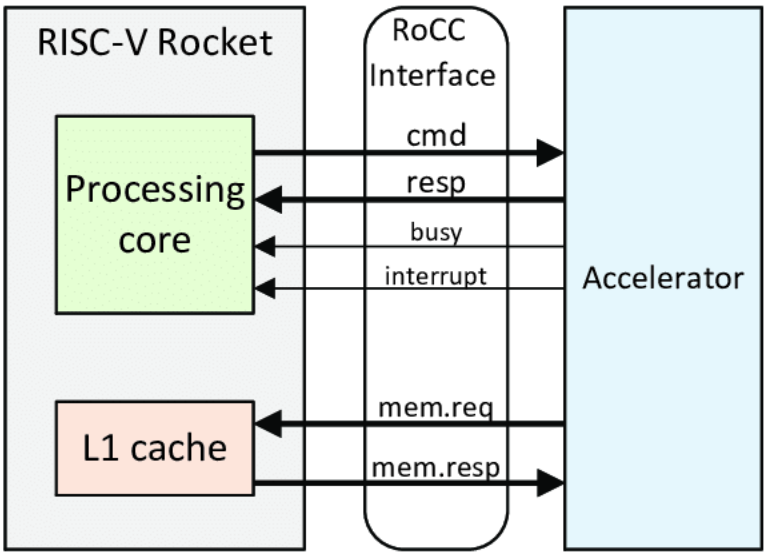


Figure 4.0.1: A simplified view of the RoCC interface

4.0.5 TileLink & Diplomacy

TileLink [36] is an open interconnect protocol for System-on-Chip (SoC) designs, originally developed for RISC-V but independent of any specific instruction set architecture (ISA). It provides coherent, memory-mapped communication between multiple masters (e.g., processors, DMA engines) and slaves (e.g., memories, peripherals), with an emphasis on low latency, high throughput, and scalability.

TileLink supports full memory coherence using a MOESI-style protocol [37, 38], which maintains cache consistency by defining five states—Modified, Owned, Exclusive, Shared, and Invalid—that track the ownership and validity of cache lines across multiple caches. This protocol enables efficient cache-to-cache data transfers and reduces unnecessary memory traffic, ensuring coherence while being inherently deadlock-free. TileLink also allows out-of-order execution of transactions, decoupled interfaces, and

hierarchical point-to-point network composition, enabling scalability from simple to highly complex architectures. Additionally, it employs low-power techniques, such as energy-efficient signal encoding, to enhance power efficiency.

The protocol is implemented through the Diplomacy framework, which automates the connection and parameterization of system components (such as L1 caches and memory controllers), as well as address space management. TileLink defines five communication channels (A, B, C, D, E), each with specific directions and priorities, ensuring uninterrupted data flow between masters and slaves, as shown in Figure 4.0.2.

The Diplomatic Widgets provided by the RocketChip library facilitate the interconnection of heterogeneous components. They support transaction buffering, reordering, fragmentation, conversions between TileLink and AXI4 [39, 40], and the construction of complex SoC systems through crossbars, FIFOs, and protocol converters. The Advanced eXtensible Interface (AXI) is a widely used on-chip communication protocol that supports high-performance, low-latency, and burst-based data transfers, enabling efficient and flexible communication in complex SoCs. As a result, the design process and scalability of SoC architectures are significantly improved.

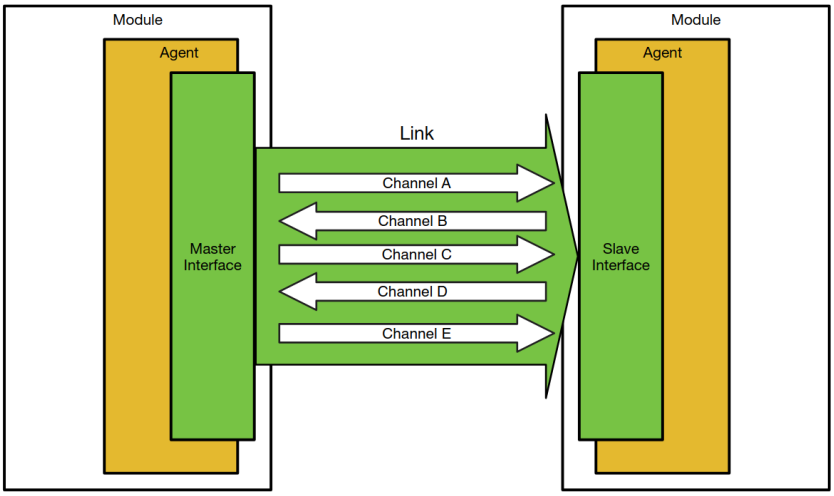


Figure 4.0.2: The five TileLink channels between master and slave agents. Hierarchical prioritization prevents deadlocks and ensures directed data flow.

4.0.6 Sync Read Memories

Chisel provides an API for defining memory elements with ready-to-use memory components. It includes support for both read-only memories (ROM) and read/write memories.

Read-Only Memories (ROM): Users can define ROMs by constructing a `Vec` with `VecInit`. `VecInit` can accept either a variable number of `Data` literals or a `Seq[Data]` sequence to initialize the memory content.

For example, one can create a small ROM initialized to values `1,2,4,8` and access them via an address generator such as a counter.

Read-Write Memories: In hardware, memory implementations vary widely between FPGAs and ASICs. Chisel abstracts this by providing the `Mem` and `SyncReadMem` constructs:

- `Mem`: Combinational (asynchronous-read), sequential (synchronous-write)
- `SyncReadMem`: Synchronous-read, synchronous-write

In our design, we use `SyncReadMem` as look-up tables (LUTs) to store partial products during computation.

Behavior of `SyncReadMem`: `SyncReadMem` is a construct in Chisel for defining synchronous-read, synchronous-write memories. These are likely to be mapped to technology-specific SRAMs rather than flip-flop banks.

If a memory address is written and read on the same clock edge, or if the read enable is de-asserted, the read value is undefined. Also, the read port does not hold data across cycles unless you explicitly register it outside the memory block.

Read/Write Interface: Access to a `SyncReadMem` memory block is done via indexing with a `UInt`. The following example shows how to define a 1024-entry SRAM with one write and one read port:

```
import chisel3._

class ReadWriteSmem extends Module {
  val width: Int = 32
  val io = IO(new Bundle {
    val enable = Input(Bool())
    val write = Input(Bool())
    val addr = Input(UInt(10.W))
    val dataIn = Input(UInt(width.W))
    val dataOut = Output(UInt(width.W))
  })

  val mem = SyncReadMem(1024, UInt(width.W))
  // Write operation
  when(io.write) {
    mem.write(io.addr, io.dataIn)
  }
  // Read operation
  io.dataOut := mem.read(io.addr, io.enable)
}
```

Listing 4.1: Single-Port SyncReadMem Example

Waveform Behavior: Figure 4.0.3 shows the typical waveform of a SyncReadMem block with a single read/write port. Note that actual RTL signal names may vary. If masking is used, Chisel may generate multiple RTL arrays to implement the desired logic.

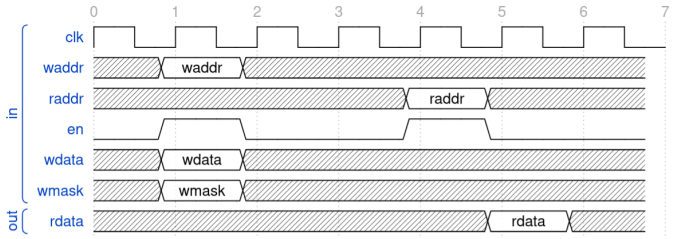


Figure 4.0.3: Waveform of SyncReadMem with one write and one read port.

Summary: A `SyncReadMem` memory must be parameterized at elaboration time with the desired number of rows (depth) and the bit-width of each row (data width). At runtime, you control whether a read or write operation occurs by setting mode signals, selecting an address (i.e., a row), and optionally providing data (for write). Only one operation (read or write) can be performed per memory per cycle, and read data becomes available in the next cycle after issuing the read command.

4.0.7 Theoretical Background: LLMs and Quantization

The Role of DNNs and Matrix Multiplication in LLMs

Deep Neural Networks (DNNs) have revolutionized how we tackle complex tasks across computer vision, speech recognition, robotics, and natural language processing. From traditional CNNs to modern transformer-based architectures, DNNs lie at the core of today's artificial intelligence systems.

Among the most impactful developments are Large Language Models (LLMs), such as GPT, BERT, PaLM, and LLaMA. These models demonstrate exceptional capabilities in text generation, summarization, translation, and question answering, often achieving human-level performance in specific tasks. LLMs are based on transformer architectures [41] where their architecture can been sheen in Figure 4.0.4 and are trained on massive corpora to learn statistical patterns in language, enabling them to generate coherent and contextually relevant text.

The inference phase—where trained models are deployed to generate outputs—is particularly critical for real-world applications such as chatbots, translation engines, and intelligent assistants. However, inference in LLMs is extremely computationally expensive due to the model size and the underlying matrix operations.

At the heart of LLM computation are matrix multiplications as is seen Figure 4.0.5, formally known as General Matrix-Matrix Multiplication (GEMM) operations. These operations are fundamental to the core building blocks of DNNs, including fully connected layers, attention mechanisms, and dimensionality projection layers.

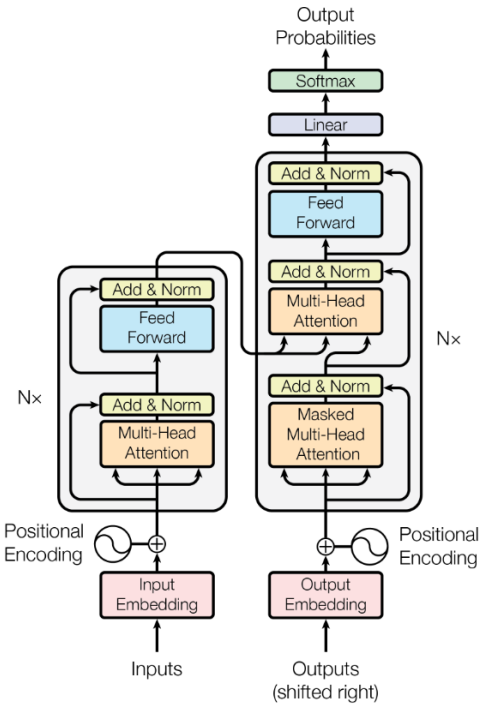


Figure 4.0.4: Transformer model architecture

GEMM operations are responsible for the majority of compute time and energy consumption in LLMs. According to recent studies, attention layers—which rely heavily on GEMM operations—can consume 70–90% of inference time on CPUs [41]. As a result, optimizing GEMM kernels is essential for speeding up model inference and improving energy efficiency.

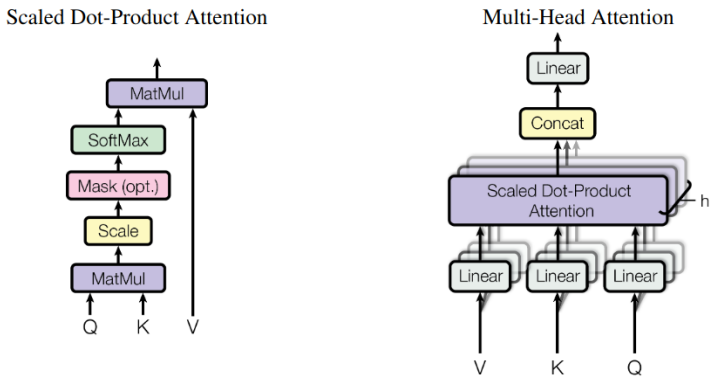


Figure 4.0.5: Self-attention and multi-head attention blocks

This makes the efficient implementation of GEMM operations a critical goal for accelerating inference, particularly in edge-computing scenarios where power and memory resources are limited.

Quantization for Efficient Inference

Quantization is a key technique used to reduce the computational cost of DNNs and, more recently, LLMs. Instead of using 32-bit floating-point (FP32) representations, model weights and activations are transformed into lower-precision formats such as INT8, FP16, FP8, or even INT4 as we can see in Figure 4.0.6 . Recent research [42] is pushing these limits further, investigating ultra-low-precision formats to improve efficiency without sacrificing accuracy.

The core motivation behind quantization is efficiency: smaller number formats reduce memory footprint, data transfer times, and computation latency. This is especially beneficial on specialized accelerators such as GPUs, TPUs, or custom AI chips.

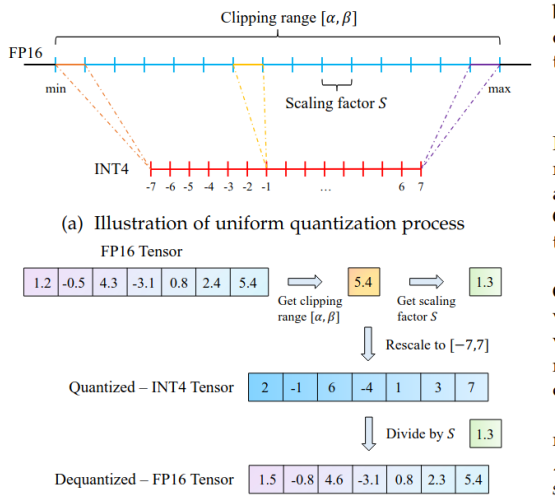


Figure 4.0.6: Quantization and De-quantization of a FP16 tensor

Quantization is particularly critical for LLMs, which can contain billions of parameters. Without quantization, deploying these models—especially on edge devices—would be impractical due to the high energy and compute demands. Moreover, quantization helps reduce latency, which is crucial for real-time applications such as conversational agents and recommender systems.

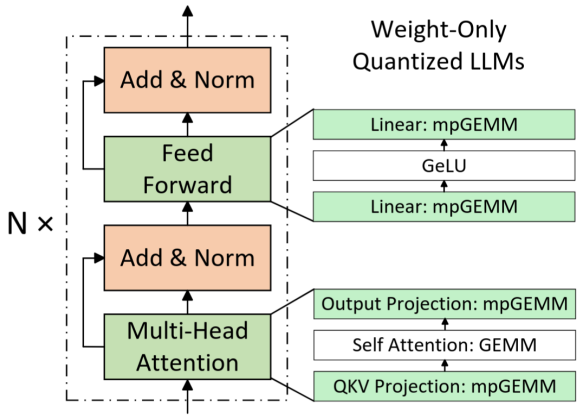


Figure 4.0.7: Decoder-only transformer blocks in LLMs. The primary computations are GEMM operations (or mpGEMM operations with weight quantization).

There are two primary approaches to quantization:

- **Post-Training Quantization (PTQ):** Applied after model training, without retraining the model. It is fast, simple, and hardware-friendly.
- **Quantization-Aware Training (QAT):** Incorporates quantization effects during training, resulting in higher accuracy, especially at low bit-widths.

Advanced techniques like per-channel quantization and mixed-precision quantization further improve accuracy and performance. While fixed-point PTQ (e.g., uniform INT8 or INT4) is most compatible with hardware, mixed-precision schemes typically require more sophisticated scheduling and memory handling.

From a hardware perspective, PTQ with uniform quantization is highly efficient, as it aligns with most existing accelerator designs (e.g., NVIDIA Tensor Cores, Google Edge TPU [43, 44]). In contrast, mixed-precision methods offer higher performance but demand more complex hardware support.

Mixed-precision quantization is a technique that assigns different numerical precisions to different parts of a neural network in order to optimize performance and efficiency. Instead of using a single fixed bit-width for all tensors, such as uniform INT8 quantization, mixed-precision allows certain tensors—typically the weights—to be represented with very low precision (e.g., INT4 or even INT2), while more sensitive components like activations retain higher precision such as INT8, INT16, or FP16. This approach is especially useful in large language models (LLMs), where massive matrices dominate both computation and memory usage. For example, it is often possible to quantize weights to INT4 without significant accuracy loss, while keeping activations at FP16 or INT8 ensures that intermediate computations remain stable and robust. This balance provides a practical tradeoff between model size and predictive accuracy. Mixed-precision quantization has several advantages: it reduces memory footprint and bandwidth requirements, speeds up inference by using lower-precision arithmetic for parts of the computation, and takes advantage of modern hardware accelerators that natively support multiple precisions. However, it also introduces challenges such as the need for careful scheduling, custom hardware or kernel support, and managing on-the-fly conversions between formats. Despite these challenges, mixed-precision quantization has become a widely adopted strategy for deploying LLMs efficiently, particularly in scenarios where computational resources are limited, such as in edge computing or latency-critical cloud applications.

In conclusion, quantization is not merely a compression technique [10, 7] ; it is a foundational strategy for computational efficiency. It enables the practical deployment of LLMs like GPT-4, LLaMA, and Mistral [45, 46] on resource-constrained environments. Without quantization, the use of such models would remain confined to high-end data centers with vast computational resources.

Chapter 5

Accelerator Design

The LUMAX accelerator consists of distinct computational and memory units, with its overall architecture illustrated in Fig. 5.1.1. It is integrated as a RocketChip coprocessor, managing instructions through the RoCC interface and handling Input/Weight/Output transfers via DMA channels. The input data are processed such that only the positive even multiplications for each activation are stored in the memory blocks. Subsequently, a selection mechanism reconstructs the correct sequence of partial products, while also encoding information about odd/even multiplications and operation sign. These are then forwarded to the accumulator, which produces the final output product. All I/O operations are managed by dedicated buffers inside the accelerator.

This work focuses on the system-level design of a DNN hardware accelerator for **mixed-precision quantized datatypes**, enabling arbitrary bitwidth combinations between activations and weights. Such flexibility allows architects to explore how different parameters interact and can be tuned to optimize overall performance. To support integration with high-level applications, a set of C functions has been developed that leverage custom instructions for accelerator control.

The LUMAX accelerator has been developed within the Chipyard ecosystem using the Chisel hardware description language. Its architecture employs a novel LUT-based approach for matrix multiplication, tailored to mixed-precision quantized models where weights often use lower precision (e.g., int8/int4/int2) compared to activations (e.g., int16/int8).

The remainder of this work is organized to progressively describe the hardware algorithm underlying the accelerator, its architectural design, the generator parameters, and its major components. The discussion concludes with the specification of the accelerator's ISA and the supporting software functions, demonstrating compatibility with high-level software integration.

5.1 Hardware Design Overview

As previously mentioned, this accelerator is an extension of the RISC-V Rocket Chip architecture as a RoCC accelerator, which implies that its communication with the processor is carried out via custom instructions. This approach has been chosen to control and re-configure the accelerator with custom instructions (e.g., specifying the operation to be performed by the accelerator or defining the precision of the activation and weight data). This environment is highly programmer-friendly, as appropriate functions have been developed in C to make use of the accelerator easier for the end-user, without the need to have hardware knowledge of the implementation.

The accelerator integrates with the existing system's memory hierarchy through a coherent TileLink interface, enabling seamless data transfers with shared memory. To reduce latency and improve bandwidth utilization, a lightweight Direct Memory Access (DMA) engine is employed to orchestrate bulk data movements between the main memory and the accelerator's internal buffers. This memory hier-

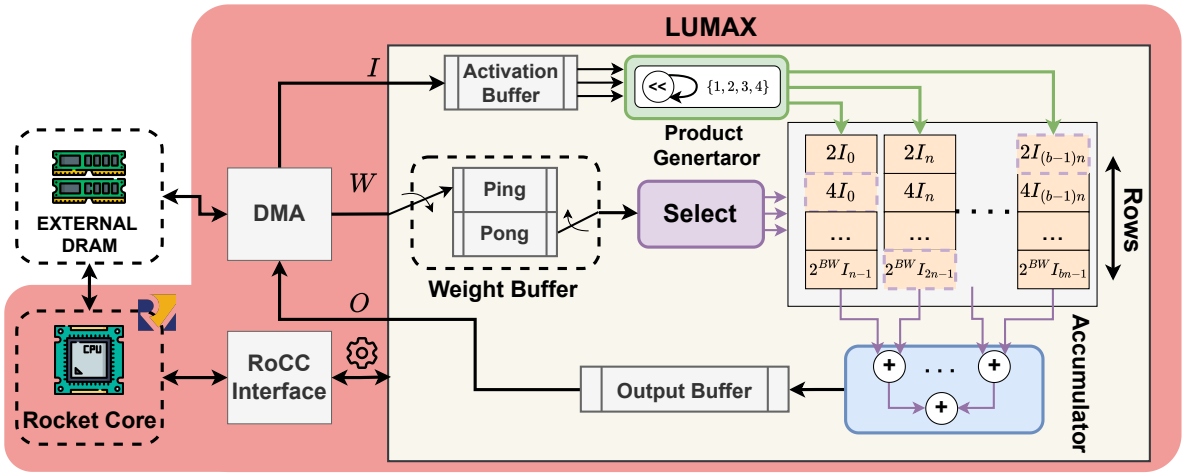


Figure 5.1.1: The Proposed LUMAX Architecture

archy, combined with DMA support and TileLink connectivity, ensures efficient and scalable communication for the execution of quantized GEMM workloads

This work proposes an accelerator capable of performing matrix multiplication while supporting variable precision for activations and weights, which is defined at runtime. To achieve this, it uses LUT-based techniques to compute and store possible products in advance, then retrieves them through appropriate indexing, thus replacing conventional multipliers.

The system consists of one **Rocket Core**, one Direct Memory Access (**DMA**) module with proper communication hardware description based on the TileLink protocol and the **RoCC accelerator.**

To store the batches of data for activation's ,weights and output we use buffers, described below.

- **Input Buffer** to temporarily store the activation values of the currently loaded window.
- **Weight Buffer**, which actually to one weight buffer, storing the corresponding weights of a window of activation's from one column.
- **Output Buffer** to store some of the output elements that accumulate values, so there is no need to send the products back to the Rocket Core constantly.

Figure 5.1.1 summarizes the general accelerator build and communication principles.We use **Ping-Pong Buffers** for weight elements so that we can write values from main memory to one buffer while reading values from the other buffer for indexing purposes. Additionally, we have a **Product Generator** that generates products through addition, stores them, and writes them to block memories. Each physical memory block is n a synchronous-read memory that enables one read/write operation per cycle and has a one-cycle delay for reading a value. We also have a **Select Module** that is responsible for finding, for each physical (synchronous-read) memory block, the index of the weight that must be read from this memory block, and for determining the row and offset where the product is located according to the weight. Furthermore, an **Accumulator Block** accumulates, using reduction, all the products selected from weights in the same column that correspond to the same input row, storing them into the correct output element. Finally, we have a **Scale Factor** module that can be enabled to scale one output row by a fixed-point number, using the same scale factor if uniform scaling per vector is applied. In later sections, we will describe each of these modules with more precision. However, it is important to mention now that the Rocket Core communicates with the accelerator via custom RoCC instructions. These instructions handle control signals, preload addresses of matrices, and manage other control tasks, such as when to start fetching weights or when to initiate the computation flow.

Along with our hardware stack, starting in the software implementation, a C API uis sed to interface with our custom ROCC hardware accelerator. The API includes three main functions:

- A *preload* function to configure runtime parameters and initiate weight fetching.
- A *start* function to issue the control signal for processing.
- A *fence* function implementing a busy-wait to ensure the accelerator completes its computation before software execution continues.

The `preload_hw` function is responsible for computing in advance, on the software side, all the necessary control parameters that govern the iteration bounds and repetition counts of each dataflow stage within the hardware accelerator. These parameters depend on the generator-defined micro-architectural factors, the runtime matrix dimensions, and the current activation and weight precisions. By performing these calculations in software, we avoid costly hardware computations such as divisions or modulo operations during execution. This approach minimizes control-path complexity and ensures that the hardware focuses solely on efficient data processing with ready-to-use parameters, without the need for dynamic reconfiguration logic.

5.2 Hardware Dataflow

This accelerator employs LUT-based techniques, where certain products results are precomputed and later retrieved through indexing to avoid costly runtime multiplications. However, precomputing all possible products for every potential bitwidth combination is impractical, as it would lead to excessive resource usage, especially when activations require higher precision (e.g., 16 or 32 bits). This would cause a combinatorial explosion in the number of possible products, significantly increasing the logic and memory footprint.

To mitigate this, the accelerator loads a window of activation elements into its internal memory. Based on the bitwidth of the weights, it precomputes and stores a limited set of partial products for this specific batch of activations. Once this precomputation is complete, the corresponding weights — fetched column-wise from the weight matrix — are used to select the appropriate precomputed products for the current activation batch. Specifically, each weight selects (depending on the number of activation rows) one or more values from the precomputed set. These selected values are then accumulated into the corresponding output elements, which are located at the intersections between the current weight column and the activation rows.

Throughout this work, the accelerator's operation is modeled on a matrix multiplication framework. The inputs (activations) are represented as a matrix of size $R_{\text{in}} \times C_{\text{in}}$, which is multiplied by a weight matrix of size $C_{\text{in}} \times C_{\text{out}}$. This multiplication produces the output matrix, leveraging the parallel processing windows defined by $\mathbf{X_S}$ and $\mathbf{Y_S}$ to compute and select partial results efficiently.

Before proceeding, it is important to clarify the notation and symbols that will be used throughout this work. Specifically, $\mathbf{In_{max}}$ and $\mathbf{W_{max}}$ denote the maximum bitwidths of the activations and weights, respectively. Conversely, **in_bits** and **w_bits** refer to the current bitwidth of the activations and weights during a given operation or configuration.

The parameters **x_slice** (**XS**) and **y_slice** (**YS**) define the processing window dimensions. In particular, **XS** does not directly represent the number of activation elements loaded, but acts as a parameter influencing how many activation elements are loaded per activation vector. This quantity may vary depending on the bitwidth of the activations, although the value of **XS** itself remains fixed across bitwidths. The parameter **YS** determines how many activation rows are processed concurrently — in other words, it defines the number of rows from which **N** elements are simultaneously taken. Together, these two parameters describe the processing window, that is, the subset of activation elements processed in parallel.

As described in Figure 5.2.1, the operation of the accelerator is based on a well-defined sequence of steps that can be divided into five distinct states. Although in the implementation, many of these steps can overlap in time (pipeline or parallel execution), here they are described as separate stages to clearly present the basic data and computation flow. The five stages are as follows:

1. **Load Activations (Load X)**
   In this stage, the elements of the input matrix/vector (activations) are transferred from the processor's external memory to the accelerator via direct memory access (DMA), following the communication protocol provided and recommended by the Chipyard environment (TileLink protocol). The transfer occurs in (often) 64-bit blocks, transferring a total of **N** input elements per row, repeated for `y_slice` rows. Essentially, depending on the bitwidth of both the activation and weight elements, a different number of activation elements are loaded from memory each time, specifically as many as can be simultaneously stored along with their possible products that arise when (conceptually) multiplied by different weights. Depending on the input bitwidth (e.g.16, 8, 4 bits), the same number of bits may correspond to different numbers of elements. Loading the required number of input elements often requires multiple memory requests, so critical parameters for this stage are how many requests can be active simultaneously (`Dma_Ids`) and the data width supported per request channel (eg. 64 bits). Obviously, this process repeats as needed until a total of `Cin` elements are loaded for each input vector.

2. **Product Generation (Generate BRAMs)**
   Once the input activation elements are loaded, the accelerator generates, for each input element, all possible product values that could result from multiplication with every even possible weight value within the supported bitwidth range (`w_bits`). These products are stored in dedicated on chip memory , so they can be reused later. How these values are placed and generated is critical and will be further analyzed. This process is implemented using add and store operation only, completely eliminating the need for multiplier units. However, this requires additional cycles to generate these values, depending on the available hardware resources dedicated for this purpose.

3. **Load Weights (Load W)**
   In this stage, the accelerator loads the weight elements corresponding to the already loaded activation elements so the weigth can uses as index to read the appropriate products fron memory blocks. Since these products are precomputed from the previous stage, it suffices to load from memory only the weights associated with the input elements whose products have been generated. Specifically, N weights are required each time (same as actvations window per row), regardless of the number of rows (`y_slice`) of the activation elements, as each weight corresponds to `y_slice` different input/activation elements. The weights are transferred in segments by column of the weight matrix and only the necessary elements corresponding to active input values are loaded. This achieves efficient bandwidth use and limits unnecessary transfers. This stage repeats `Cout` times for each new load of activation's window, so efficient execution is important to avoid bottlenecks caused by weight transfers.

4. **Select & Accumulate**
   This is the core operation of the accelerator. For each weight element, the accelerator uses its value as an index to select the corresponding precomputed product value from the on chip memory. The selected product is then added to the corresponding output element. If multiple rows (input vectors) are processed simultaneously (indicated by the `y_slice` parameter), the same weight element is used to select multiple products, each contributing to a different output element (as they belong to different output neurons), and these are then summed in the final result. This enables reuse of the weight matrix elements, as weights directly select their products without recomputing the same product multiple times. This stage also repeats `Cout` times for each new segment of input elements, and as will be shown later, there is a producer-consumer relationship between this stage and the weight loading stage.

5. **Store Outputs (Store O)**
   The final stage concerns transferring the computed output values back to main memory via direct memory access (DMA). Writing occurs only after `y_slice` rows of output have been fully computed. The data is temporarily held in output buffer and transferred only when computation is complete (Output Stationary), so communication and data transfer are not burdened by many small write operations from the processor to main memory.

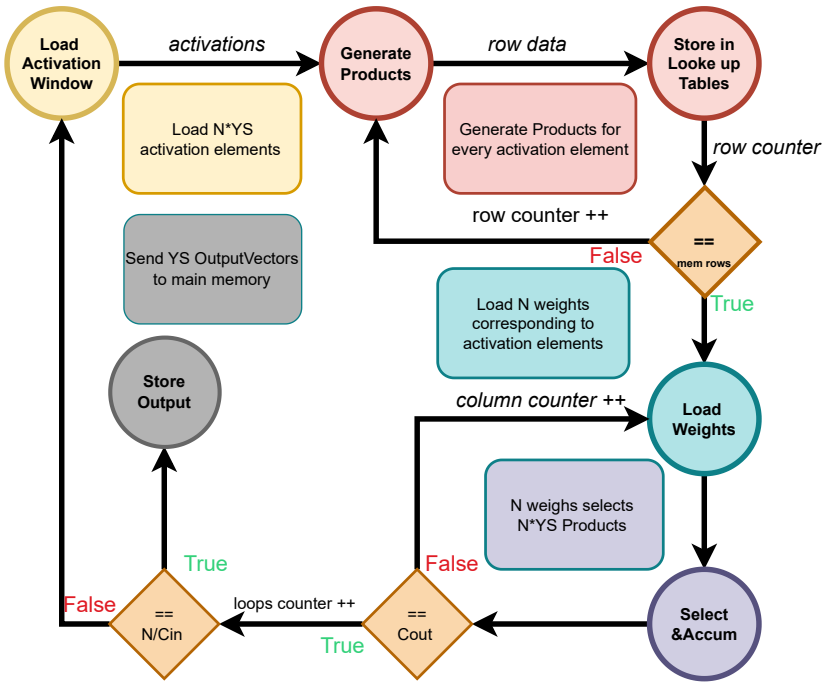


Figure 5.2.1: Accelerator States Diagram

Next, we present a simplified example to illustrate the concept. In this example, we make the same assumptions as before but omit any hardware logic details for clarity. We consider memories with 64 rows each, where each row can store exactly one product — an assumption that will be justified later. For simplicity, we ignore bitwidth considerations. Here, `X_slice` (XS) and `Y_slice` (YS) are both set to 2, meaning we process a window of 4 activation elements — 2 elements per row. Additionally, we assume that all possible products for a single activation require one dedicated memory each. Consequently, for one activation, all its generated products occupy $XS \times YS$ physical memory units.

For the following example, we assume a weight bitwidth of 8 bits and, for reasons to be discussed later, we only generate absolute values and only use even weights (2, 4, 8, ..., 128). As we will see later we handling the logic the products when weights are odd, so for now is like every logical block containing all products of one activation element is assumed to have the same size as one physical memory.

Let us now explain more specifically the hardware dataflow according to the simplifications previously assumed and using the 5.2.2 as a reference.

At the beginning, we read a 2×2 window of activation elements. For each one of these elements, we generate and store in memories (blocks) all the possible products that could result from multiplying an `in_bit` activation with a weight of `w_bit` precision. It is clear that for lower weight precision, there are fewer products to generate.

Then, we load all the corresponding weights column by column so that they can select (read) the products they need, depending on which activation element (from those we have already brought in) is routed to the appropriate memory block, and then acts as an index to choose the correct product.

We must note that in this example we load only 2 weights, for every weight column, there are 2 selection operations from two different blocks, with each block corresponding to one activation element.

As shown in part (1) of the figure, we start by bringing in 2 weights from the first column of the weight matrix, and because we are at the start, these are the first 2 elements of that column. Then, as we see in step (2) of the example, the next weights corresponding to the activations we have already loaded

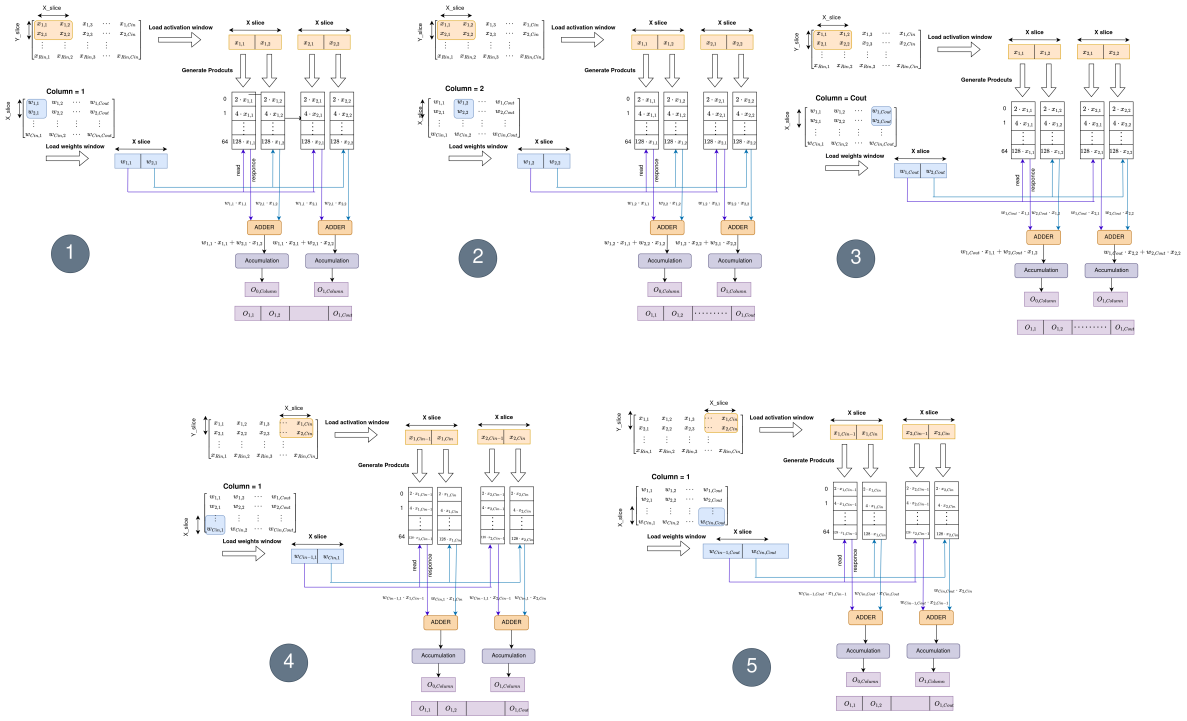


Figure 5.2.2: Simplified Example of the Hardware Dataflow

and whose already products have been generated, are located in the next column of weight matrix, specifically on the same rows as before.

Thus, we move column-wise, reading repeatedly from the same rows — in our case 2 rows — until we have read all columns, as shown in step (3) of the example. Up to this point, all the output elements of the first two output rows have received and accumulated some values, but are not yet fully computed.

Then, since there are no further weights associated with these activations, we bring in the next activation window and repeat the same procedure.

When the activation window reaches the last elements of the first two rows of the activation matrix, the hardware reads the final two weights of each column. These weights are then used in the last read-and-accumulate operations, producing the complete first two rows of the output matrix.

Once this step is completed — as shown in part (5) of the example — the computation for these rows is finalized. If the input matrix contains additional rows, the same procedure is repeated with the new activation elements, while the weight elements remain unchanged.

To generalize, we extend the hardware dataflow to a more flexible form. Each memory block can hold the partial products of N activations, thus accommodating $N \times$ `Y_slice` elements in total. The parameter `X_slice` is directly related to N; their connection will be examined later.

In the general case, for every $N \times$ `Y_slice` activation block (spanning columns i through j), the system loads exactly N weight values per column of the weight matrix (`Cout` columns in total), located from row i to j. This guarantees that all required weights are available, allowing the precomputed products for the activations to be properly selected and accumulated.

Algorithm 1 Accelerator Dataflow

```
Activations[Rin, Cin]
Weights[Cin, Cout]
Output[Rin, Cout]

elemers_per_row ← activation_elements / YS
    // === Outer loop over activation rows ===
for row = 0 to Rin/YS - 1 do
    start ← 0
    // — Loop over activation column blocks —
    for columns = 0 to Cin/elemers_per_row - 1 do
        // — Loop over YS rows within activation block —
        for ys = 1 to YS do // — Loop over elements per row —
            for elems = 1 to elemers_per_row do
                Load Activations[row × YS + ys, start + elems]
            end for// — End elements per row loop —
        end for// — End YS rows loop —

        # Generate products for every activation
        Generate_Products()
        // — Loop over weight columns —
        for columns_w = 1 to Cout do // — Loop over elements per row for weights —
            for elems = 1 to elemers_per_row do
                Load Weights[start + elems, columns_w]
            end for// — End elements per row loop —

            # Select products and accumulate them to outputs
            Select_and_Accum()
        end for// — End weight columns loop —
        start ← start + elemers_per_row
    end for// — End activation column blocks loop —
    // — Loop over YS output rows —
    for ys = 1 to YS do // — Loop over output channels —
        for c = 1 to Cout do
            Write Output[row × YS + ys, c]
        end for// — End output channels loop —
    end for// — End YS output rows loop —
end for// === End outer loop ===
```

Table 5.1: Repetition counts of accelerator dataflow stages

Stage	**Repetitions**
Load Activations and Generate Products	$\frac{R_{in}}{Y_S} \times \frac{C_{in}}{\texttt{elements\_per\_row}}$
Load Weights and Select&Accumulate	$\left(\frac{R_{in}}{Y_S} \times \frac{C_{in}}{\texttt{elements\_per\_row}}\right) \times C_{out}$
Store Output	$\frac{R_{in}}{Y_S}$

Generator Parameters

The Design of the accelerator have some parameters that can change some critical hardware modules dimensions or even change other aspects so a more specific and more hardware friendly design wil be generated using less resources or even to see how different component parameters of the system change the performance.

First, it is important to present the design parameters as shown in Table 5.2 that determine the dimensions and characteristics of the accelerator. These parameters serve as the foundation for conducting **Design Space Exploration (DSE)**.

Parameter	Description
`In_max`	Maximum supported bitwidth for input activations
`W_max`	Maximum supported bitwidth for weights
`OutBitWidth`	Bitwidth of the accumulated output values
`In_min`	Minimum supported bitwidth for activations
`W_min`	Minimum supported bitwidth for weights
`x_slice`	Specifies the minimum number of elements that can be processed in parallel per activation row (horizontal parallelism), guaranteed even under the maximum supported bitwidth for activations and weights.
`y_slice`	Number of input rows loaded and proccesing simultaneously (vertical parallelism)
`bankMergeFactor`	Factor that reduces the number of memory banks by merging them, increasing the row count per bank to maintain total capacity
`Row Factor`	Factor that increases the number of rows per memory bank, expanding total memory capacity without changing the number of banks
`Dma_Ids`	Maximum number of in-flight DMA requests supported
`Dma_buswidth`	Maximum number of bytes transferred per DMA request
`SCALE`	Flag to enable applying a fixed-point scaling factor on the output
`Simple cache`	Flag to Enable pre-computed memory blocks for very low activation precision

Table 5.2: Generator Parameters defining the design space of the accelerator

Variable Precision Support

The parameters `In_max` and `W_max` define the maximum bit width that the accelerator can process for activations and weights, respectively, while `In_min` and `W_min` define the minimum supported precision as illustrated in Figure 5.2.3. For example, if `In_max = 8` and `In_min = 4`, then each activation register can store either one 8-bit element or two 4-bit elements, thereby fully utilizing its capacity. In this design, an input (activation) register has a bit width of `In_max`, while a weight register has a bit width of `W_max`.

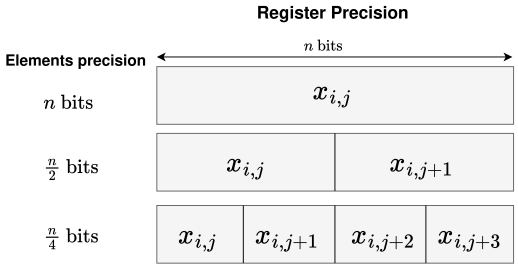


Figure 5.2.3: Elements packing for different precision

The design supports run time activation's and weight of different precision, allowing the accelerator to adapt not only during memory reads and register storage, but also during product generation, on chip Memory distribution, and parallelism degree. When the data has lower precision, the number of parallel operations increases, leading to higher throughput and better hardware utilization.

Parallelism and Memory Blocks Usage
The parameters `x_slice` and `y_slice` define the number of activation elements for which products can be generated and processed in parallel, and indirectly determine the amount of memory Blocks required (`x_slice` × `y_slice`). The `x_slice` parameter relates to the number of Memory Blocks per row of matrix , while `y_slice` sets how many rows of matrix are processed simultaneously. Each memory block (in default configuration) can stores all possible products for the worst-case precision scenario (`In_max`, `W_max`). When data of lower precision is used, the memory blocks can be partitioned (by rows or columns) to maximize capacity utilization. Therefore, with `x_slice` and `y_slice`, and assuming enough storage in the memoy blocks, it is possible to store products for `x_slice` activation's with bit width `In_max` and weights of bit width `W_max`. As can be understood, this is only one scenario and actually the worst and slowest case, since if either the activation bits or the weight bits are reduced, it becomes possible to store products for even more activation's within each memory Block. This allows the processing of more elements in parallel and effectively scales up the size of the activation window that is read in each step.

Memory Bank Structure Scaling Factors The parameters `bankMergeFactor` and `Row_Factor` are scaling factors that modify the organization of the on-chip memory banks. The `bankMergeFactor` reduces the number of physical memory banks by merging them, proportionally increasing the number of rows per bank in order to preserve the total memory capacity. In contrast, the `Row_Factor` increases the number of rows in each memory bank without reducing the number of banks, effectively expanding the total memory capacity. These two parameters can also be used simultaneously — for instance, each memory block can be both deeper (more rows) and merged (fewer total banks) depending on the selected configuration.

By default, each memory block is provisioned with a number of rows sufficient to store all the intermediate products generated by one activation when multiplied with all weights, assuming the highest supported precision for both activations and weights. This baseline configuration ensures compatibility with all lower-precision modes. Tuning the memory scaling factors allows exploration of different memory hierarchies, offering trade-offs between parallelism, memory reuse, and hardware area.

DMA Requests and Data Flow
The accelerator implements a DMA mechanism to move data from the main memory to the accelerator's local memory (registers or scratchpad), as well as to transfer accumulated results back to the main memory. The size of these DMA transactions is determined by the DMA parameters. These DMA parameters are tightly coupled to the Rocket Chip SoC system; in particular, the `DMA_buswidth` parameter is associated with the `SystemBusKey.beatBytes` configuration.

The parameter `Dma_Ids` defines how many outstanding requests the DMA can issue without waiting for responses. The higher this value, the greater the overlap between read/write operations and processing, leading to faster execution. Maximum performance is observed when all required low-precision data can be transferred simultaneously.

Cache for low activation's precision
The `Cache` flag determines whether a cache logic based on LUTs is used for low-bitwidth activation's and weights. Since the design is optimized for low weight bit width and higher activation bit width, performance may degrade when activation's also have low precision . For example, with 4-bit activation's and 8-bit weights, it is not efficient to generate products for each 4-bit activation independently, since 8-bit weights require storing at least 128 products. A better approach is to use a precomputed LUT that contains all 4-bit × 8-bit products (about 5–6 KB) as we can see in Figure 5.2.4 , allowing both the activation and the corresponding weight to act as indices for selecting products. Then, using the same accumulation flow as the general design, these products are summed into the output elements. Because in this scheme there is no need to store a window of activation elements in memory blocks

(since the products are already precomputed in the cache logic memory), the activation window can be maximized, as activation's are only used for indexing. This can outperform the non-cache approach, provided there is sufficient data bandwidth to support efficient loading of the larger window. Of course, this technique is not efficient for activation's with 8 or 16 bits, since the required number of products would be too large, increasing storage demands and adding indexing complexity.

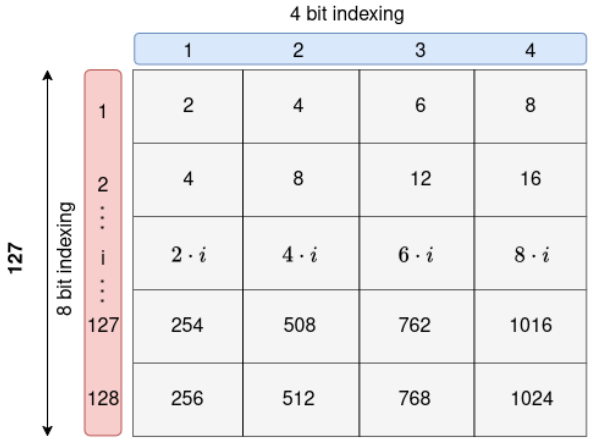


Figure 5.2.4: Cache 4 -8 precision

Output Scaling (Dequantization)
The `SCALE` flag determines whether *dequantization* is performed at the output. When enabled, the results are converted to fixed-point format with an appropriate scale factor (e.g., 32-bit) per output vector. When disabled, the output is stored as integer (INT) values in memory.

5.3 Major Components

This section describes the most important hardware components of the accelerator and how they interconnect. Furthermore, it explains how different generator parameters modify the RTL of the design, enabling flexible customization according to these parameters. In this way, developers can experiment with and evaluate various design trade-offs easily.

We will discuss the following key components of the accelerator architecture:

- memory Blocks (On-chip memories)
- Direct Memory Access (DMA) module
- Product generator component
- Select and accumulate component

These elements form the core of the hardware structure and determine how data is stored, transferred, computed, and accumulated within the accelerator.

5.3.1 Memory Blocks

In our design, a very important aspect is how and where to store the partial products in a way that is both performance-efficient and hardware-friendly. We have chosen to implement a synchronous read - write memory with single-cycle read and write latency, featuring a single read/write port that allows one data word to be accessed per cycle. These memories are implemented using Chisel's `SyncReadMem` primitive, which is a synthesizable, ASIC- and FPGA-friendly synchronous memory structure. These memories are referred to as *Physical Memory* or *Physical Memory Blocks* within our architecture. Of

course, it is necessary to organize and partition each Physical Memory Block into different logical sections as illustrated in Figure 5.3.1 in order to fully exploit the available storage resources and maximize utilization among the different precision in activation's and weights .

Now that the accelerator parameters have been introduced, we proceed to describe in detail the data flow and how synchronous memory usage changes across different bit width configurations.This includes the parallel processing of activation and weight elements, and how the activation window adapts to the current precision. Furthermore, we analyze how the size of the synchronous memory affects the placement of product values within the dedicated memory blocks.

Assume `bankMergeFactor` = 1 and `Row_Factor` = 1. Under this configuration, there are exactly `x_slice` × `y_slice` synchronous read memories (physical memory blocks). Each block contains $2^{W_{max}-2}$ rows, where W_{max} denotes the maximum supported weight bit width. For instance, with $W_{max} = 8$, each physical memory block provides 64 rows. This capacity ensures that each block can store all positive products that may result from even-valued weights of a single activation in the worst-case precision scenario, where the weights use their maximum supported precision.

It is important to note that only absolute weight products are generated and stored, since signed representations are used and negative values are handled during run time. To further optimize storage, only products for even weights are stored; odd weights are reconstructed by adding the product of the closest even weight with the activation, as detailed later.

Row-wise partitioning of Synchronous Memory (Rows) : For every activation element, we must allocate a logical memory block containing its associated products (as dictated by the weight precision). These logical blocks are then packed into the available physical memories. The height in rows of a logical block, denoted as `block_rows`, is computed as

$$\text{block\_rows} = 2^{w_{\text{bits}}-2} \tag{5.3.1}$$

where `w_bits` is the active weight precision at run time.

With `bankMergeFactor` = 1, each physical memory contains $2^{W_{max}-2}$ rows, sufficient to accommodate the entire logical block of products for a single activation in the worst-case precision. At lower weight precision's, these logical blocks are smaller and multiple blocks can be packed sequentially within the same physical memory.

The `bankMergeFactor` parameter scales the row capacity of each physical memory block. For example, if `bankMergeFactor` = 2, each physical memory will contain $2^{W_{max}-2} \times 2$ rows, allowing the same product capacity to be supported with fewer physical memories.This provides a flexible trade-off between memory size and the number of memories required.

Also Row_Factor just scales the numbers of rows per psycial memory block so can increase actvations products per memory with out dealing with extra indexing logic what every memory need.

In summary, the number of physical memories and their characteristics are given by:

$$\text{Mems} = \frac{x_{\text{slice}}}{\text{bankMergeFactor}} \times y_{\text{slice}} \tag{5.3.2}$$

$$\text{Mem\_Rows} = 2^{W_{\text{max}}-2} \times \text{bankMergeFactor} \times \text{Row\_Factor} \tag{5.3.3}$$

This strategy enables flexible and efficient placement of logical product blocks within physical memories depending on run time bit widths and configuration parameters.

Horizontal Partitioning of Synchronous Memory (Width)
Another important optimization is horizontal partitioning of the memory. Each row of a synchronous memory has a fixed bit width, sized to hold the widest product possible under maximum precision.

However, in practice the active bit widths (`in_bits` and `w_bits`) can be typically lower, reducing the actual product width. This allows multiple logical product blocks to be packed side-by-side within the same memory row, improving resource utilization. Each product block is stored as a concatenated bit vector, and masking with offset selection allows retrieval of the appropriate product segment during execution. This technique supports increased parallelism for higher activation precision's combined with moderate or low weight precision's.

Hence the physical memory width is:

$$\text{Mem\_Width} = (In_{\min} + W_{\max}) \times \frac{In_{\max}}{In_{\min}}. \tag{5.3.4}$$

while the width of each logical block is:

$$\texttt{block\_width} = \texttt{in\_bits} + \texttt{w\_bits}. \tag{5.3.5}$$

By doing this we can have full memory blocks utilization and increase activation window parallelism, memory efficiency, and area as is we clear can see in Figure 5.3.2.

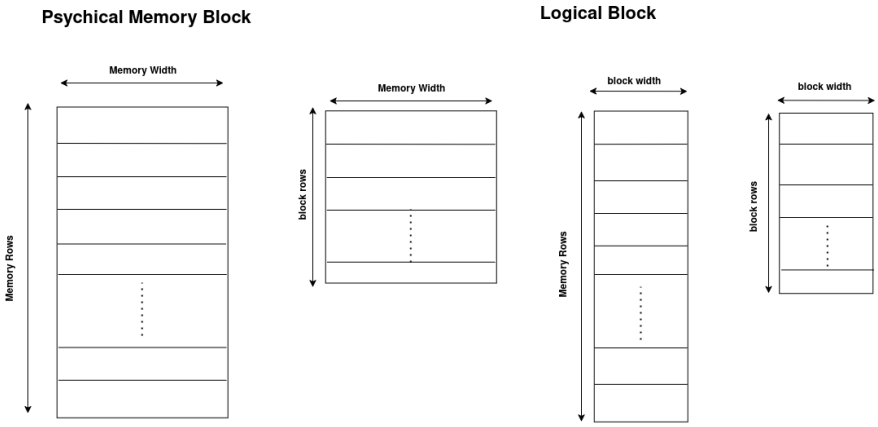


Figure 5.3.1: Memory Block dimension and logical block for different actvations and weights precision's

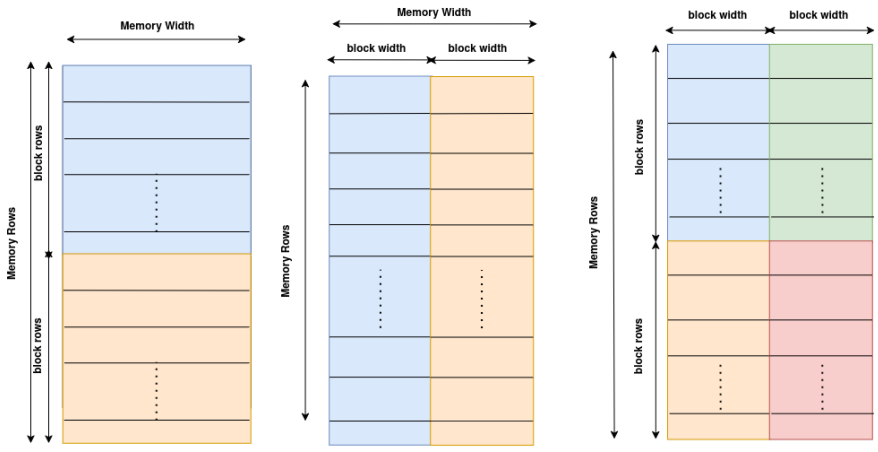


Figure 5.3.2: Mapping of activation blocks onto physical memory blocks, where each color corresponds to a different activation block.

In the 5.3.1 as a reference, we observe that a *synchronous memory* is essentially one physical Memory Block with a fixed number of rows and a fixed width, effectively forming a one-dimensional array. This dimension can be configured in the generator using the parameters described earlier, but it remains fixed for each generated design. On the other hand, the *logical blocks* depend on the precision of activations and weights. Since the design supports variable bitwidths at runtime, the logical blocks can vary in the number of rows and their width. Recall that a logical block essentially stores the products for a single activation element. As a result, a logical block may occupy fewer rows than a physical block, a smaller width than a physical block, or both fewer rows and a smaller width. We have ensured that logical blocks fit neatly within a single physical memory block. This way, no logical block needs to be split across multiple physical memories, which would otherwise increase the complexity of later selecting the correct elements. Finally, in 5.3.2, we illustrate how logical blocks are mapped within a physical memory block when, for example, the `block_width` equals half of the physical memory width, and the `block_rows` equals half of the physical memory rows.

Now that we have described how the products for each activation element are stored in logical blocks, and how these blocks are placed within synchronous memories (physical on-chip memories), we can examine how, for different precision levels of both activations and weights, the degree of parallelism achieved during processing varies. The maximum number of activation elements (activations window elements) the accelerator can handle is given by

$$\texttt{N} = \texttt{Mems} \times \frac{\texttt{Mem\_Rows}}{\texttt{block\_rows}} \times \left(\frac{\texttt{Mem\_Width}}{\texttt{block\_width}} \right). \tag{5.3.6}$$

This window of activation elements is distributed across Y_S activation rows, resulting in

$$\texttt{elems\_per\_row} = \frac{\texttt{N}}{\texttt{YS}} \tag{5.3.7}$$

Logical Blocks mapping to Synchronous Memories

In this section, we complete the discussion of the structure of the synchronous on-chip memories (Sync Memories), by explaining how the logical blocks containing the partial products for each activation element are organized and mapped. In particular, it is necessary to clarify how each logical block is associated with a specific activation element within the loaded activation window, where exactly it is placed in memory, and how it is indexed so that the corresponding weights can later access it correctly.

As previously described, the activation window of `N` activation elements is loaded into a register vector, with each register having a bitwidth equal to the maximum supported activation precision, denoted $In_{\max}$. This guarantees that the accelerator can handle any possible bitwidth configuration. A key question is: how many registers are required to handle every possible bitwidth? The answer is determined by considering the worst-case scenario, namely when activations use their maximum precision while weights use their minimum precision. This configuration allows the largest number of activations to be processed in parallel, and thus requires the maximum number of registers.

Accordingly, the maximum number of activation registers per actvation vector can be expressed as:

$$\texttt{max\_activation\_regs} = 2^{W_{\max} - W_{\min}} \times XS \tag{5.3.8}$$

Each register is $In_{\max}$ bits wide, and there are such registers for each of the Y_S activation rows. Therefore, the total register file capacity is $\texttt{max\_activation\_regs} \times Y_S$.

$$\text{max\_weights\_regs} = \text{max\_activation\_regs} \times \left(\frac{In_{\max}}{In_{\min}} \right) \tag{5.3.9}$$

Each register is $W_{\max}$ bits wide.

Within each register, the number of activation elements that can be stored depends on their bitwidth. Specifically, each activation register can store $In_{\max}/in_{\text{bits}}$ activation elements.

Because the block width is smaller than the product width, it is possible for multiple logical blocks to share the same row in a Sync Memory, each located at a different offset along that row. Therefore, the position of each activation element within its register determines the offset of its corresponding logical block inside the memory. Our accelerator, as previously mentioned, can load up to N activation elements per window, corresponding to $\frac{N}{YS}$ elements per activation vector, as discussed in equations 5.3.6 and 5.3.7.

We now examine how many registers are required, and how each register is associated with a register block. As mentioned earlier, a register block contains $\frac{In_{\max}}{in\_bits}$ logical blocks. This implies that each block either holds the partial products of one activation element or multiple elements. For example, if $In_{\max} = 16$ and $in\_bits = 8$, then each register block holds data for two activation elements. These logical registers are concatenated row-wise.

In theory, we can load up to $\frac{N}{YS}$ elements. However, if the input dimension Cin is smaller, the actual number of elements loaded is limited to:

$$N_{\text{real}} = \min\left(\frac{N}{YS}, Cin\right) \tag{5.3.10}$$

Thus, per vector row (YS), the required number of activation and weight registers becomes:

$$\text{Activation\_regs} = \frac{N_{\text{real}}}{In_{\max}/in\_bits}, \quad \text{Weight\_regs} = \frac{N_{\text{real}}}{W_{\max}/w\_bits} \tag{5.3.11}$$

Next, we must determine how to distribute register blocks across the available memory banks. To maximize performance, it is essential to balance the usage across all available memories. For instance, if we have 2 memory banks, each capable of holding 4 register blocks, and we require 4 total blocks, then it is preferable to distribute them evenly (e.g., 2 per memory) rather than fully utilizing one and leaving the other idle. This ensures we exploit the hardware's capability of one read per cycle per memory.

To compute how many register blocks each memory will manage at runtime, we group them as follows:

$$G = \frac{\text{Activation\_regs}}{\text{Mems}/YS} \tag{5.3.12}$$

where G is the number of register blocks per memory.

The first Sync Memory holds the Register blocks and their associated registers for the activation row $ys = 0$, covering block indices from 0 to $G - 1$, then the subsequent group from G to $2G - 1$, and so forth. This systematic allocation defines exactly which logical blocks are assigned to each physical memory and how they are mapped to their corresponding activation registers.

After filling the first X_S registers, the dataflow moves to the next activation register row, incrementing ys until $ys = Y_S - 1$. Within each of these activation rows, the block assignments repeat in the same ranges 0 to $G - 1$, G to $2G - 1$, etc., ensuring a consistent and predictable layout of blocks across the physical memories. This can become clear by seen the Figure 5.3.3

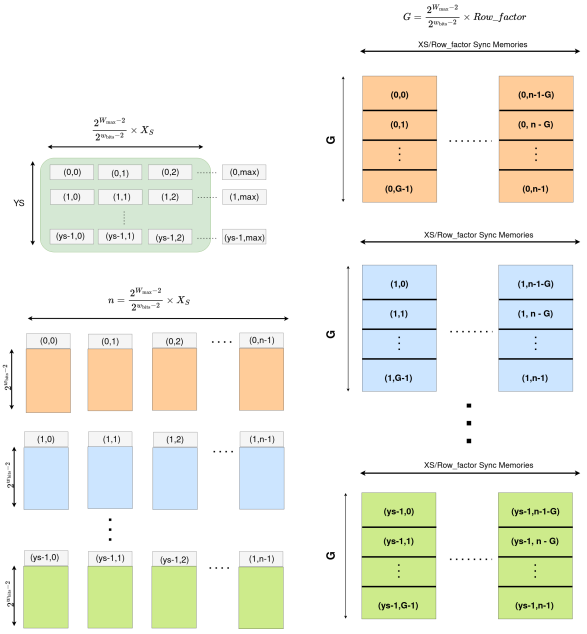


Figure 5.3.3: Block indexing across registers and physical memories.

Having established how many activation registers each synchronous memory needs, we now consider the vertical arrangement of these memories — namely, how they can be partitioned along the vertical axis. This partitioning is possible because each logical block has $\frac{In_{\max}}{in_{\text{bits}}}$ distinct offsets. As described earlier, the partial products are generated and stored contiguously within each memory row, with the offsets encoding different activation elements. Thus, by reading an entire row of physical memory and applying a specific offset mask, the design can retrieve the partial products associated with any desired activation. This strategy enables vertical partitioning of the physical memories while still supporting efficient and flexible access to partial products.

As previously discussed, the architecture manages two levels of logical blocks: **register logical blocks**, which correspond to each allocated register, and **element logical blocks**, which represent each activation element stored within a register.

Each register logical block internally holds multiple element logical blocks. The number of elements it can hold depends on the register packing factor, which is given by $In_{\max}/in_{\text{bits}}$. In other words, a single register can store multiple activation elements depending on their bit-width. These element blocks are logically independent, although they are physically concatenated within the same register logical block.

Register logical block mapping We define a register location by its coordinates $(x_{\text{th}}, y_{\text{th}})$, where

$$x_{\text{th}} \in [0, \texttt{max\_activation\_regs} - 1], \quad y_{\text{th}} \in [0, Y_S - 1].$$

Each such register corresponds to a dedicated register logical block.

The total number of physical synchronous memories (Sync Memories) in the system is denoted `Mems`, with each containing `Mem_Rows` rows of storage. Note that, for each activation row y_{th}, there are $\frac{\texttt{Mems}}{Y_S}$ physical memories available.

Furthermore, as calculated earlier, each Sync Memory can store up to G register logical blocks, indexed from 0 to $G - 1$.

Given this structure, the mapping of each register block is determined as follows:

1. **Physical Memory ID** where the register block is placed:

$$\texttt{memory\_id} = y_{\text{th}} \times \frac{\texttt{Mems}}{Y_S} + \left\lfloor \frac{x_{\text{th}}}{G} \right\rfloor \tag{5.3.13}$$

2. **Index of the register block inside the assigned Sync Memory**:

$$\texttt{register\_block\_id\_in\_mem} = x_{\text{th}} \bmod G \tag{5.3.14}$$

3. **Starting row inside the Sync Memory** for this register block:

$$\texttt{block\_row\_start} = \texttt{register\_block\_id\_in\_mem} \times \texttt{block\_rows} \tag{5.3.15}$$

This mapping scheme provides a systematic placement of register logical blocks across Sync Memories, ensuring predictable addressing and efficient memory partitioning.

Finally, each register logical block contains $\frac{In_{\max}}{in_{\text{bits}}}$ element logical blocks, each corresponding to a different offset within the same physical memory row. These offsets allow the accelerator to selectively retrieve partial products for each activation element while supporting flexible precision.

5.3.2 Communication Architecture

The design supports two types of communication: between the RISC-V processor and the accelerator, and between the accelerator and external memory.

- **RISC-V ↔ Accelerator Communication:**
  The processor and the accelerator communicate through custom instructions over the RoCC (Rocket Custom Coprocessor) interface. Specifically, the Rocket core:
  - sends the addresses of the matrices,
  - provides the dimensions of these matrices,
  - specifies the bitwidths of activations and weights,
  - issues a start-processing signal to trigger computation,
  - and receives busy/ready signals from the accelerator, so that the accelerator can notify the processor when the computation has completed.
- **Accelerator ↔ Memory Communication:**
  Beyond the RoCC interface, the accelerator exchanges data with the main memory using DMA (Direct Memory Access) over the TileLink protocol. This mechanism allows the accelerator to autonomously read activation and weight matrices from memory and to write back the output matrix after processing is complete.

RoCC Custom Instruction

The accelerator communicates with the processor via the standard RoCC interface, which provides two 64-bit source registers (`rs1`, `rs2`) to send data to the accelerator, and one 64-bit destination register (`rd`) to receive data from the accelerator. Each custom instruction is uniquely identified by its function code, which the accelerator uses to interpret the operation and process the transferred data accordingly.

We define the following control instructions for setting up the accelerator:

- `set_activations(activation_address,` R_{in}`,` C_{in}`)`
- `set_weights(weights_address,` C_{out}`)`
- `set_output(output_address)`

- `set_bitwidths(activation_bits, weight_bits, output_bits)`

In addition, we provide execution controls:

- `start_calculation()` — sends a signal to begin computation.
- `wait_until_not_busy()` — a blocking instruction that stalls further code execution until the accelerator finishes its matrix multiplication.

DMA Communication

Since using the RoCC interface to directly transfer large matrix data would be extremely inefficient, the design instead implements a classic Direct Memory Access (DMA) interface. Here, the accelerator operates as the bus master, issuing read and write requests directly to main memory. This is far more efficient for high-volume data.

The DMA interface has two key configuration parameters:

- **DMA_Id**: the maximum number of concurrent memory requests (in flight) that the accelerator can issue. If this limit is reached, the accelerator must wait for one or more responses before issuing new requests.
- **DMA_bytes**: the data width of each memory request (commonly 64 or 128 bits).

For each matrix, the accelerator maintains two small internal counters to track the current position:

- (i_x, j_x): coordinates for the activation matrix
- (i_w, j_w): coordinates for the weight matrix
- (i_o, j_o): coordinates for the output matrix

Additionally, there is a `start_counter` variable to track how many elements of the current row for activations or column for weights have already been read and processed in previous activation windows.

The DMA using the simple TileLink Uncached Lightweight (TL-UL) protocol that have two channels the A channel that master (Accelerator) send a read/write request and and D channel where slave send a responce data/acknowledgement.

Activation Data Transfer Protocol

This section describes in detail how the accelerator loads activation data from memory using the DMA interface. The process is divided into two phases: the *request* mechanism and the *response* mechanism.

(1) Request Mechanism

To load an activation window starting at coordinates (i_x, j_x), the linear base address is calculated as

$$\texttt{linear\_addr} = \texttt{activation\_addr} + (i_x \times C_{in} + j_x) \times \texttt{in\_bits}.$$

For each window, the accelerator needs to load `elements_per_row` activations across Y_S rows. Therefore, the total amount of bits per row group is

$$\texttt{activation\_data} = \texttt{elements\_per\_row} \times \texttt{in\_bits}.$$

Given that each DMA transaction transfers `DMA_bytes` bytes, the number of memory requests required is

$$\texttt{packets\_in} = \left\lceil \frac{\texttt{activation\_data}}{\texttt{DMA\_bytes} \times 8} \right\rceil.$$

Hence, the accelerator issues `packets_in` DMA requests to fetch the activation window. If the number of required requests exceeds `DMA_id`, then the DMA signals `busy`, and the accelerator waits until a

source ID becomes available before issuing the next request. After each request, the coordinates for the activation window are updated as

$$i_x \leftarrow i_x + \texttt{elements\_per\_row}, \quad j_x \text{ unchanged.}$$

When the current row is fully processed, the counters are reset to

$$i_x \leftarrow \texttt{start\_counter}, \quad j_x \leftarrow j_x + 1$$

to continue with the next row of the activation matrix. Once all Y_S rows of activations have been loaded, the coordinates (i_x, j_x) remain unchanged to indicate the next address for the following activation window.

(2) Response Mechanism

While receiving responses, a register index counter $\texttt{register\_idx} = 0$ tracks how many register vectors have already been filled. Each DMA response contains a data packet with up to `activation_data` bits, which may include multiple packed activation elements. This data is subdivided into chunks of

$$\frac{\texttt{DMA\_bytes} \times 8}{\texttt{In\_max}}$$

so that each chunk fits exactly into a single activation register. The accelerator stores each chunk in

$$\texttt{Activation}\,(\texttt{register\_idx} + i) = \texttt{Chunk}(i)$$

for all i corresponding to the chunks in that packet. The register index pointer is then incremented by

$$\texttt{register\_idx} \leftarrow \texttt{register\_idx} + \frac{\texttt{DMA\_bytes} \times 8}{\texttt{In\_max}} \tag{5.3.16}$$

Once an entire activation window is completely loaded, the register index counter is reset to zero, so the next window begins loading from the first register.

Both the request and response mechanisms employ counters to ensure proper synchronization. The request counter tracks how many DMA requests have been sent for the current activation or weight window. Once all requests for the window have been issued, the accelerator halts further requests until all corresponding responses have been received.

Similarly, the response counter monitors the number of DMA responses received. When all responses for the current window are returned, the accelerator confirms that all necessary data is available and proceeds to the subsequent computation stage. This coordination guarantees data integrity and efficient pipelining of the accelerator's workload.

Weights Data Transfer Protocol

The weights loading mechanism is largely analogous to the activation loading process described previously, with the following important distinctions:

(a) The weight matrix is accessed column-wise (transposed relative to activations), which affects the linear address calculation.

(b) The linear indexing for weights is given by:

$$\texttt{linear\_addr} = \texttt{weight\_addr} + (j_w \times C_{in} + i_w) \times \texttt{w\_bits},$$

where i_w and j_w are the row and column indices within the weight matrix, respectively, and `w_bits` denotes the bitwidth per weight element.

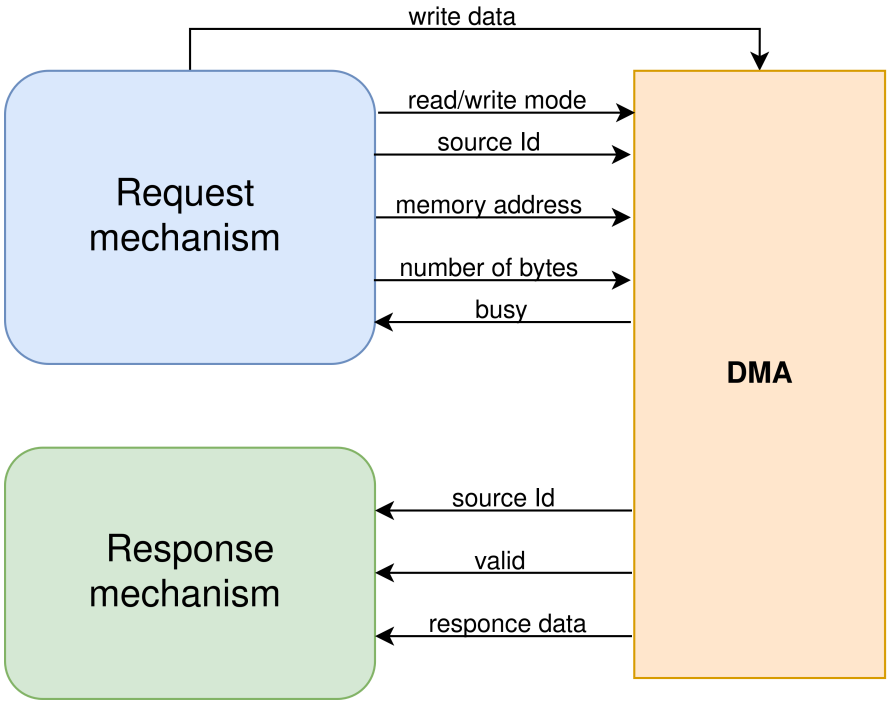


Figure 5.3.4: Request Responce DMA System

Aside from these differences, the number of DMA packets sent, the updating of indices, and the response handling follow the same principles outlined for activation loading.

In in Fig. 5.3.4,, we illustrate the signals exchanged between the accelerator (acting as the DMA master) and the DMA module. First, the accelerator sends a control signal to select whether it will write to memory or read from memory.

Read from memory: The accelerator sends a *source ID*, identifying which DMA slot (among the available `DMA_id` slots) is being used, along with the *start memory address* from which to read in main memory and the *number of bytes* to read. The *busy* signal indicates that all available DMA source IDs are currently occupied with in-flight requests, so the accelerator must wait until at least one response returns and frees a source ID before issuing a new request. On the other hand, the accelerator monitors the *valid response* signal from the DMA, which notifies when the requested data has been retrieved from main memory. Upon receiving a valid response, the accelerator can then process the returned data accordingly.

Write to memory: Similar to the read operation, the accelerator specifies the starting address for the write as well as the number of bytes to store in memory. It then performs a *concatenation* of the data into a single unsigned data packet and transmits it. A mask is also provided in case certain intermediate data elements should not be modified. Subsequently, the accelerator begins issuing write requests to memory until all data has been sent or until no available source ID exists. In that case, it simply waits for a response message, confirming that specific data have been written to memory, before proceeding with the next write request.

Request Mechanism (Read Mode). The request mechanism in read mode is divided into two parts: a *control part* and a *computation part*.

- **Control Part:** This logic determines when to issue the next request. It uses a counter, `dma_counter`, which tracks how many requests (for a specific window transaction) have already been successfully sent to the DMA. This is compared with `packets`, a register storing the total

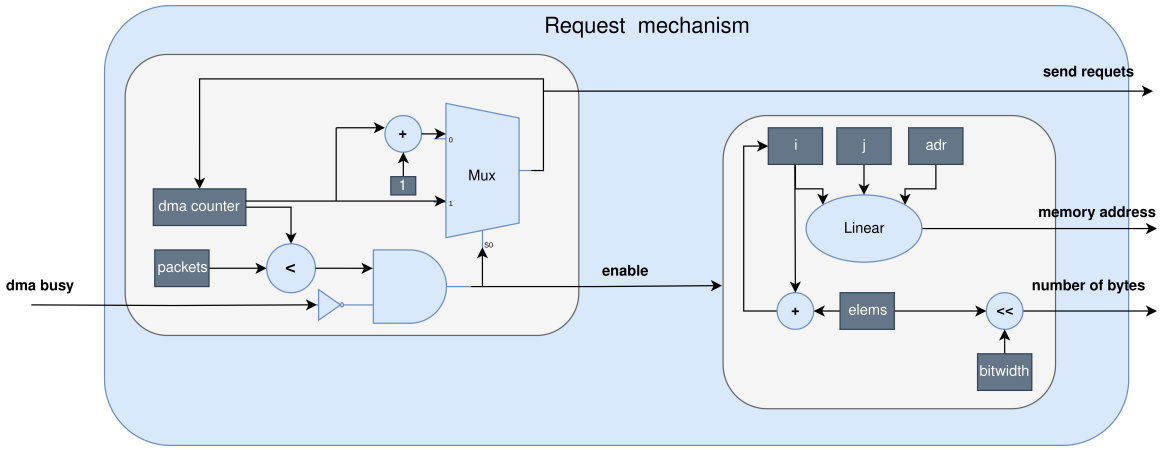


Figure 5.3.5: Request Logic for Read mode

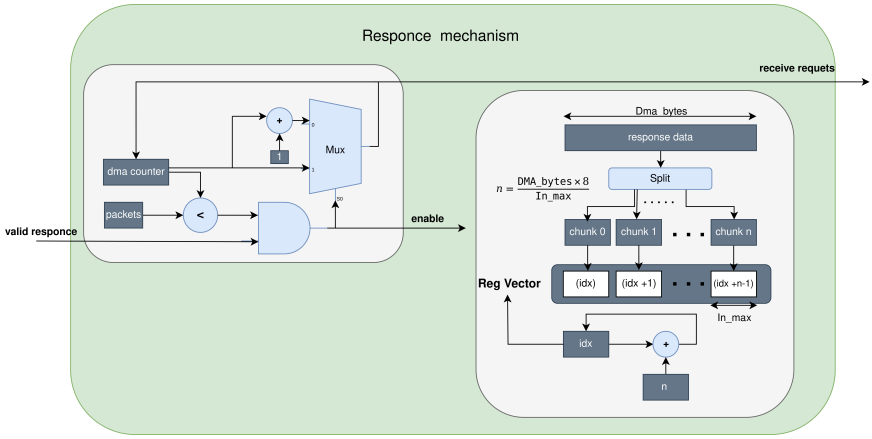


Figure 5.3.6: Response Logic for Read mode

number of requests needed to fully read the desired activation or weight elements. In addition, a new request can only be issued if at least one source ID is available, indicated by the `dma_busy` signal being low. In summary, a request is allowed to proceed if:

$$\texttt{dma\_counter} < \texttt{packets} \quad \text{and} \quad \texttt{dma\_busy} = 0.$$

When these conditions are met, the request is issued, `dma_counter` is incremented, and an enable signal is asserted to trigger the computation logic.

- **Computation Part:** This logic calculates the memory address from which to read, based on the current (i, j) coordinates and a base address register. It also determines how many bytes should be read for the current request, according to the number of elements requested and their bit-width. After calculating these parameters, the module updates the window counters:
  - for activations, typically the i coordinate is incremented
  - for weights (due to their transposed storage), typically the j coordinate is incremented.

In this way, the hardware module as illustrated in Figure 5.3.5 handles reading data efficiently by interleaving control logic (for managing DMA slots and in-flight requests) with address computation (for correctly generating memory accesses).

Response Mechanism (Read Mode). The response mechanism in read mode is similarly divided into two parts: a *control part* and a *data storage part*, as can be seen in Figure Figure 5.3.6 .

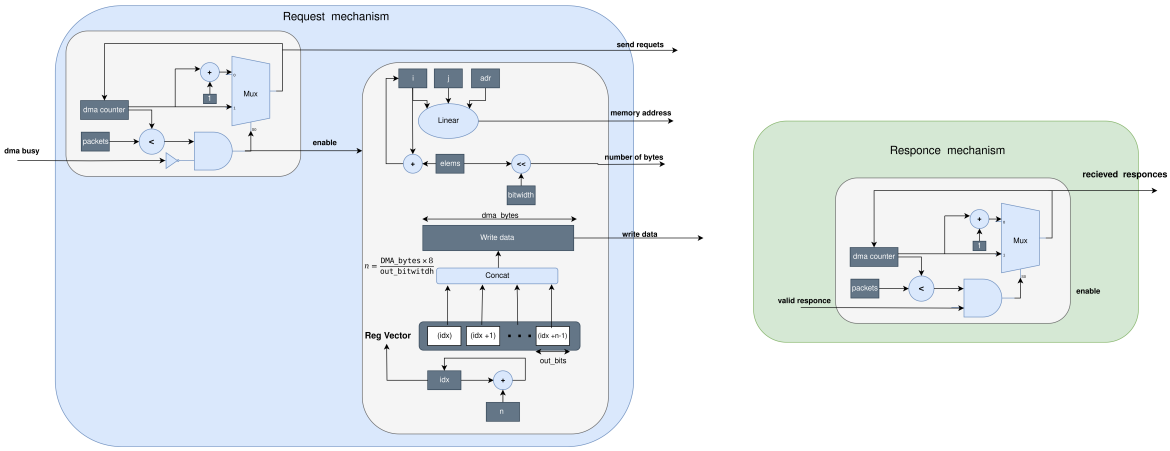


Figure 5.3.7: Request - Responce mechanism for Write mode

- **Control Part:** This section uses a counter, `dma_counter`, which counts how many valid responses have been received from the DMA for the current transaction. Whenever the DMA asserts a valid response signal, the counter is incremented, and an enable signal is asserted to allow the data to be stored. The transaction is considered complete when:

$$\texttt{response\_valid} = 1.$$

  At that point, the module transitions to the next stage of processing.

- **Data Storage Part:** When a valid response arrives from the DMA, it contains `DMA_bytes` worth of data, which is then split into chunks of `In_max` bits (for activations) or `W_max` bits (for weights), depending on the data type. These n chunks

$$n = \frac{\texttt{DMA\_bytes} \times 8}{\texttt{In\_max}}$$

  are sequentially stored into n activation registers. To avoid overwriting existing data, a register index pointer, `register_idx`, is maintained, and is incremented by n each time a response is stored so that the next group of data will be placed in the next set of free registers. After all packets of the current window are received and processed, this index can be reset to zero for the next transaction.

Write Mode Request and Response Mechanism In write mode, the request and response mechanisms operate similarly to the read mode as illustrated in Figure 5.3.7. However, in the request phase, there is an additional step in which the accelerator concatenates data from multiple output registers into a single packet before sending it to main memory. This packet includes all the data to be written along with an optional write mask, in case some data within the packet should not be modified. On the response side, the accelerator simply increments a received request counter whenever a valid response signal is returned, confirming that the data has been successfully written to memory.

During the design of the DMA architecture, **alignment-related issues** may arise, especially when the data width varies (2, 4, 8, 16, or 32 bits) as can been seen in Figure 5.3.8 . If the starting address of the transfer is not aligned according to the word size supported by the bus (e.g., a 32-bit word on a non-4-byte-aligned address), the TileLink UL protocol may either split the transfer across multiple cycles—reducing throughput—or trigger an error in slaves that do not support unaligned accesses. To mitigate such issues, an aligned transfer strategy is implemented: the DMA initiates the read from an address aligned before the desired data, covering a sufficient length to fully include the requested segment.

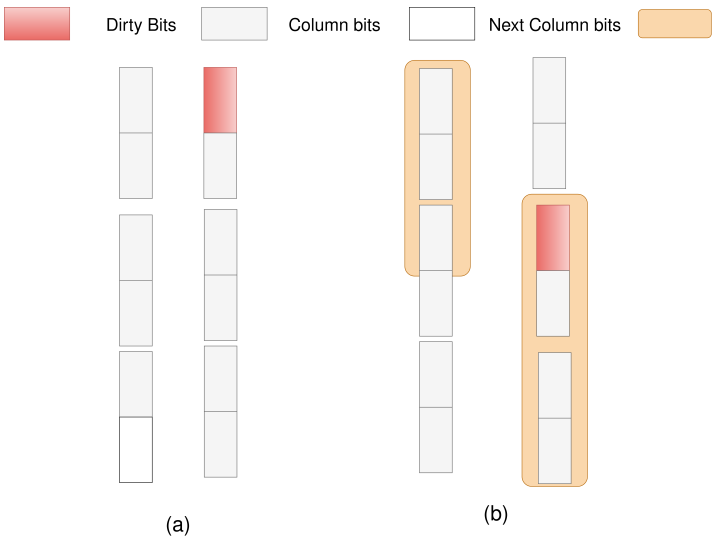


Figure 5.3.8: Illustration of misaligned data fetching across columns. In (a), a larger activation window reads almost an entire column, but in order to stay aligned with subsequent columns, it must redundantly fetch the last element again, marked in red as a "dirty bit." In (b), a smaller activation window reads only a partial column at a time; as it slides down the column, it overlaps with previously read elements, again introducing redundant or dirty bits to maintain proper data alignment.

Then, with appropriate offsetting and masking, the unwanted data preceding the target location is ignored, using dedicated "dirty bits" registers to track valid segments. In this way, full compatibility with the TileLink bus is ensured, while maintaining optimal performance regardless of the element width of the transferred matrix. So we use dirty bits as a mask , we already know what element we have read so if we must read the same element we now how many bits is dirty so we read them to read align the data via dma but we dont use them at all.

One important aspect to consider is that the request–response mechanism introduces logic that must calculate the target address for read or write operations. This calculation may significantly increase the critical path of the design. To address this challenge, we propose a simple yet effective solution based on precomputing the request data one cycle ahead. This approach does not modify the data itself; rather, one cycle before a transition stage (such as load activation or load weight), the target address is precomputed so that it is immediately available in the following cycle. Consequently, the DMA can receive the request without delay, while the next request is already being calculated. This reduces the critical path associated with address generation. Similarly, for the response path, waiting to calculate the register index in order to store the response can also increase the critical path. Therefore, we precompute the register index one cycle earlier, so that as soon as a valid response arrives, the data can be stored immediately while, in parallel, preparing the register index for the next valid response. This approach improves timing closure and minimizes the critical path in both the request and response flows.

5.3.3 Generate Products

Another important component is the *Product Generator*, which is responsible for producing the possible values of each activation within the activation window. We have already discussed how these values are stored in the synchronous read memories (Sync Read Mems); now we will describe how these values are generated. As previously mentioned, the elements are represented using signed integer values. To generate all possible products for a specific activation, it is necessary to precompute all the potential results that may arise when this activation is multiplied by any possible weight value, according to the

bit width of the weights. The following optimizations are applied:

1. **Absolute Products:** Only the absolute values of the products are generated, meaning that the absolute value of each activation is multiplied by the absolute values of all possible weights. Since signed products are symmetric around zero (for example, 8×2 and $8 \times (-2)$ have the same absolute magnitude), storing only the absolute results reduces the number of precomputed products by half. Later, dedicated sign logic determines whether the final product should be positive or negative depending on the sign of the activation and the weight.
2. **Even-Weight Products:** Only the products resulting from multiplying each activation value by even-valued weights are precomputed. This means that not all possible activation-weight products are generated, but only those that can arise if the weights are restricted to even values. This approach further halves the amount of product data that needs to be precomputed. Later, dedicated logic can handle any remaining products involving odd-valued weights, effectively completing the full product space on demand.

As a result, for each activation, there are 2^{w_bits-2} different products to generate. Each register in the memory stores a concatenation of In_max/in_bits different activation elements.

Because each synchronous memory (Sync Mem) has only one write port, the logic is replicated and unrolled so that one write can be performed on every Sync Mem in parallel. As a result, there is a separate product-generation logic instance for each Sync Mem. The system begins to compute the products and enables one write for each Sync Mem per cycle, proceeding row by row from the first row of each memory to the last row.

At the start of the product generation process, the same activation register mapping described previously is reused. Specifically, according to the Sync Mem index and the current row, the system can identify the corresponding activation register for each memory block. A row counter is maintained for all Sync Mems, indicating which row is currently being written. Based on this row, the correct activation register block can be easily selected according to the Sync Mem mapping.

Given: First, we define:

$$mems\_per\_Ys = \frac{Mems}{ys}$$

This value indicates how many physical Sync Memories are dedicated to each $ys \in [0, YS-1]$, that is, to each row of the activation window.

Given:

- **Constants (static):**
  - `mem_idx`: Index of the physical memory block.
  - `mems_per_ys`: Number of physical memories allocated per input vector (per activation window row).
- **Runtime reconfigurable parameters (depend on weight bitwidth):**
  - `G`: Number of logical register blocks that put in one physical memory.
  - `block_rows`: Number of rows in each logical register block.
- **Indexing counters:**
  - `block_counter`: Current logical register block within each physical memory, ranges from 0 to `blocks_per_mem` - 1.
  - `block_row_counter`: Current row inside the current logical register block, ranges from 0 to `block_rows` - 1.

the coordinates of the corresponding activation register can then be calculated as:

$$\texttt{total\_blocks\_per\_y} = \left(\frac{\texttt{Mem}}{YS}\right) \times G \tag{5.3.17}$$

$$\text{global\_block\_idx} = \text{mem\_idx} \times \text{G} + \text{block\_counter} \tag{5.3.18}$$

$$x_{\text{th}} = \text{global\_block\_idx mod total\_blocks\_per\_y} \tag{5.3.19}$$

$$y_{\text{th}} = \left\lfloor \frac{\text{global\_block\_idx}}{\text{total\_blocks\_per\_y}} \right\rfloor \tag{5.3.20}$$

This selects the correct activation element corresponding to each block. For each block, an accumulator register, called *Product*, is maintained, with the following sequence:

$$Product(sync\_mem\_idx) = 0 \quad \text{(initialization)}$$
$$Product(sync\_mem\_idx) = Product(sync\_mem\_idx) + Activation\_reg(y_{th})(x_{th}) \times 2$$

This accumulation continues until the `row_in_block_counter` reaches the total number of block rows, indicating that for this activation, all required products have been generated and stored. At that point, the *Product* register is re-initialized to zero, and the `row_in_block_counter` is reset. However, the `row_counter` is incremented, preparing the system to process the next activation block in the pipeline.

Another important aspect is the handling of multiple activation elements concatenated within the same register. To manage this, both the activation register and the product accumulator register are logically split into chunks, each corresponding to the individual concatenated elements. Accumulation is then performed independently for each chunk, and the results are concatenated back together before being written to memory.

The coordinate calculation for each activation chunk uses the same formulas for x_{th} and y_{th} as described previously. However, the registers are now divided as follows:

$$Activation\_reg(x_{th})(y_{th}) = [chunk_1, \ldots, chunk_n]$$
$$Product(sync\_mem\_idx) = [chunk_1, \ldots, chunk_n]$$

The product update for each chunk i is computed independently as:

$$Product(sync\_mem\_idx)_{chunk_i} = Product(sync\_mem\_idx)_{chunk_i} + Activation\_reg(x_{th})(y_{th})_{chunk_i} \times 2$$

Each Sync Memory block contains its own dedicated logic that performs the following steps: based on the current `block_counter` within the Sync Memory and using the predefined mapping, it identifies the corresponding activation register. According to the current activation bitwidth, it dynamically splits the register and the accumulation logic at runtime into $\frac{In\_max}{in\_bits}$ parts. It then performs accumulation to compute the new product value by adding and shifting accordingly. Finally, it writes the current product to the memory address corresponding to the previous `block_row_counter` (i.e., `block_row_counter` - 1).

To implement runtime splitting and accumulation for different activation precisions, each activation register contains concatenated activation elements with maximum precision In_{max}. Each individual activation element, however, has a runtime-configurable precision in_{bits}.

Given an activation register of width In_{max}, we create a bitmask

$$\texttt{mask\_in} = 2^{in_{bits}} - 1,$$

which isolates the lower in_{bits} bits of a value. To handle the worst-case scenario, where an activation register is split into multiple products, we define

$$\texttt{max\_parts} = \frac{In_{max}}{in_{bits}},$$

representing the number of activation parts within a register.

For each part index $part_{idx} \in [0, \texttt{max\_parts} - 1]$, the corresponding activation sub-value is extracted from the register $Activation\_Reg(y_{th})(x_{th})$ as

$$value\_part = |((Value \gg (part_{idx} \times in_{bits})) \,\&\, \texttt{mask\_in})|\,,$$

where the absolute value accounts for signed activations.

Similarly, the product accumulator register is split with a potentially different bit-width $product\_bits$ and mask `mask_product`:

$$product\_part = ((P\_value \gg (part_{idx} \times product\_bits)) \,\&\, \texttt{mask\_product})\,,$$

Accumulation is performed by

$$product\_part = product\_part + (value\_part \ll 1),$$

where the shift by 1 corresponds to multiplication by 2, as required by the design.

This process is unrolled across all $part_{idx}$, and finally, all partial products are concatenated to form the updated $Product$ register:

$$Product = \text{Concat}(product\_part_0, product\_part_1, \ldots, product\_part_{\texttt{max\_parts}-1}).$$

Finally, all the individual product chunks are concatenated to form a single product register, which is then written back to the corresponding Sync Mem at index $sync\_mem\_idx$.

Lastly, when writing to the Sync Memories, the write operation must only occur if the `block_row_counter` is not zero. This condition ensures that the first product (essentially the value multiplied by 2) has already been produced. The data is then written to the memory row addressed by `block_row_counter - 1`. The `block_row_counter` is a counter that increments every cycle during the "Generate Products" stage.

Formally:

$$\text{valid\_write} = (block\_row\_counter \neq 0)$$

$$\text{Write\_Address} = mem\_row\_counter - 1$$

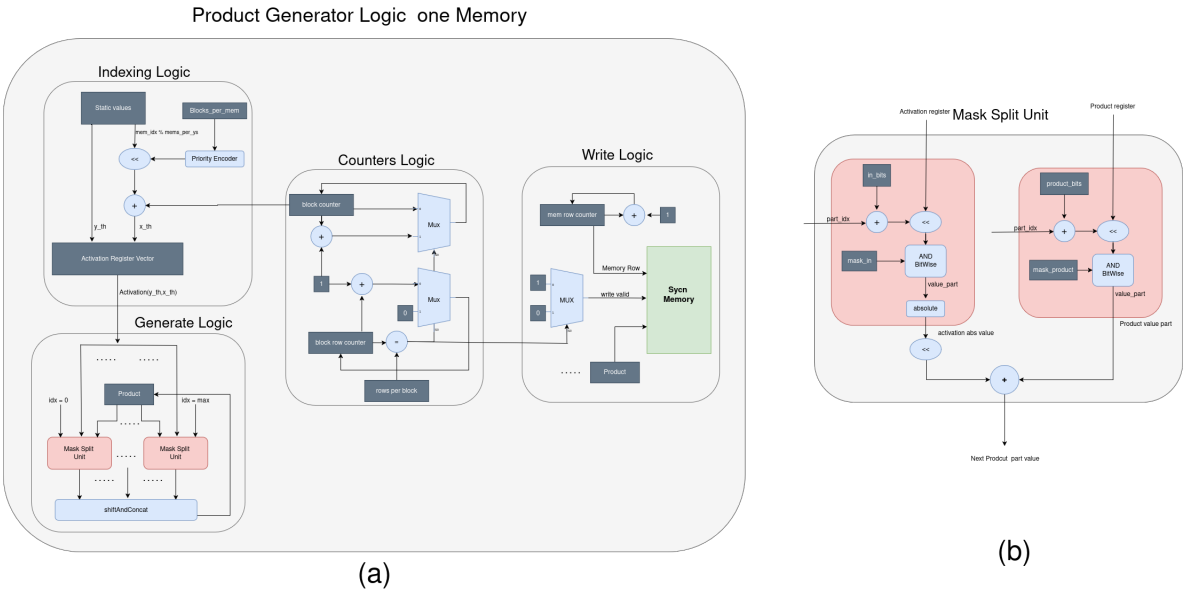


Figure 5.3.9: (a) Product Generator for one memory (b) Mask Split Unit

Poducts Generator Units for one Memory

In the above Figure 5.3.9, we see the logic needed to generate the products for a single Sync Memory. There is an *Indexer* module, whose role is to determine, according to the current block counter and some precomputed values (such as how many blocks fit into each memory and how many physical memories exist per input vector), the corresponding activation element within the activation register vector. Next, there is a *Generator* module, whose role is to fetch the activation register data and, via a *Mask Split Unit*, dynamically split it into the required parts, doing the same for the product register (each Sync Memory has its own product register, acting as an accumulator). The Mask Split Unit, as shown in part (b) of the figure, computes the mask based on the current activation precision and splits both the activation and product registers accordingly. The activation register also passes through a multiplexer to take the absolute value of the activation, since, as described earlier, only absolute products are written to memory.

It is important to note that the Mask Split Unit is replicated multiple times (unrolled) in case the register must be split into more than one part; each part is processed with add-and-shift, and then concatenated again. If the register is used fully, the same logic is effectively applied repeatedly, which is equivalent to performing an OR of the identical binary value, preserving the same result. If, however, the mask is smaller than the register width, the partial segments are ORed together to produce the final concatenated binary value. Finally, in the Generator module, the newly computed product is written back to the product register so that it can be stored to memory in the following cycle.

Additionally, there is a *Counter Logic* module responsible for updating all counters. In particular, there is the block counter indicating which block is currently processed, and the block row counter showing which row within the current block is active. When the block row counter reaches the last row of the current block, the block counter is incremented, moving to the next block, and the block row counter is reset to zero, since the next block starts from its first row.

Lastly, there is a *Write* module, which decides when and where to write to the Sync Memory. We also have a *memory row counter* that counts the memory rows as they are written sequentially. A valid write occurs only when the block row counter is not zero, because if it is zero, the new product value has not yet been calculated. The written value comes from the product register and is stored to the memory row indicated by the memory row counter.

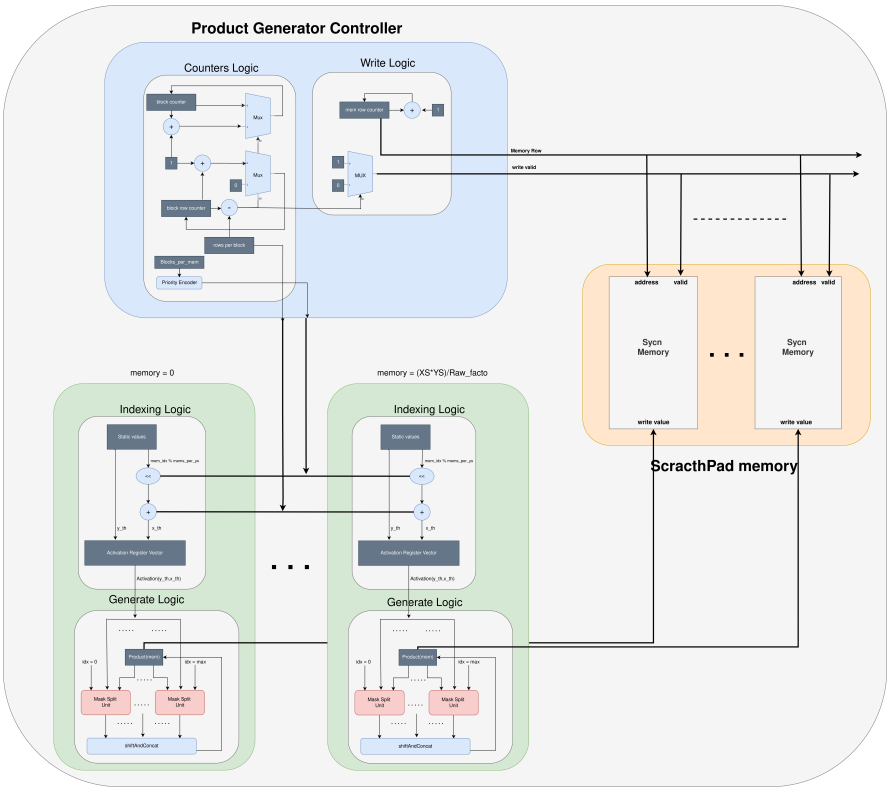


Figure 5.3.10: Product Generator for multiple Memories

Products Generator Units for Scratchpad Memory

In the Fig. 5.3.10 , we see the same product generation logic, but now scaled to handle multiple Sync Memories, which together form a scratchpad structure. The same principles apply as in the single-memory case, but in a more structured form, since some parts of the control logic are shared across all Sync Memories. Specifically, there is a *Controller* module, which incorporates both the Counter Logic and the Write Logic. This is because we traverse all memories row by row, meaning that in all memories we are at the same row (so we need a global memory row counter), and likewise we are at the same block across all memories (for example, block 0 of the first memory, block 0 of the second memory, etc.). Consequently, all the control signals and counter signals are shared across all Sync Memories.

On the other hand, the indexing logic is distinct for each Sync Memory, since each memory needs to select its own corresponding activation element. Although we are in the same block for all memories, the activation register stores first all activation elements of all blocks for the first memory, then for the second memory, and so forth. Therefore, an offset must be applied depending on which Sync Memory is addressed, to correctly index its elements.

Furthermore, each Sync Memory maintains its own product register (accumulator) and its own local logic for accumulation, mask splitting, and concatenation, because it manages the accumulation of its unique activation elements. Finally, the same control signals from the Controller (including the counters) are used to coordinate writes to memory, but each Sync Memory writes its own product value to its dedicated memory row. This means that, in the same cycle, all memories write to the same row index, but with different product data as illustrated in Figure 5.3.11 .

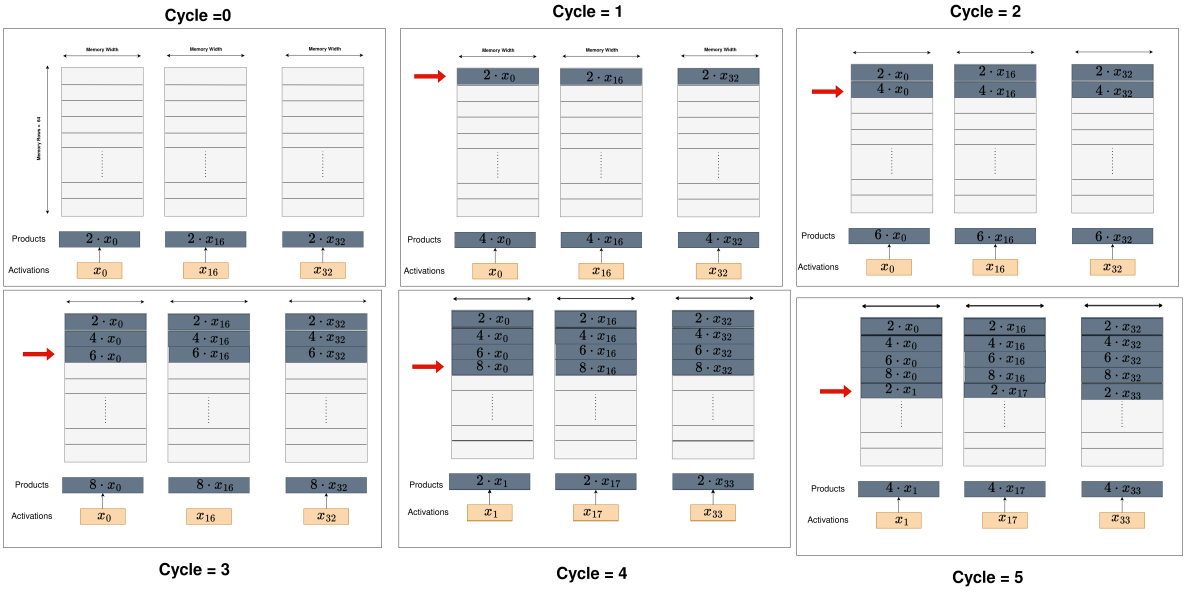


Figure 5.3.11: Example of product generation for 64 memory rows with 4-bit weights and register-matched activation bitwidth. The diagram shows the cycle-by-cycle filling of memory blocks, where each cycle writes one row into each block.

5.3.4 Select and Accumulate

The main processing module of the accelerator implements a *select-and-accumulate* procedure, where precomputed products are selected and accumulated toward the corresponding output elements. An essential part of this process is the indexing mechanism, which determines, for each activation (and thus for each logical block), the corresponding weight to use as an index for selecting the correct partial product. Each Sync Memory block has its own dedicated logic to retrieve the corresponding weight value and select the correct memory address (row) to read from.

To begin, consider the Weight Register Vector, which acts as an on-chip buffer holding all the weights for a current weight matrix column. This buffer stores `N/YS` elements, allowing every activation row to select its corresponding weight value. Since each Sync Memory supports one read per cycle with a one-cycle delay, the logic in each memory (Sync Memory) is responsible for fetching, cycle by cycle, the required weight element. Consequently, the total number of reads per Sync Memory needed to cover every activation element (logical block) is calculated as:

$$\texttt{reads\_per\_sync\_memory} = G \times \frac{In_{\max}}{in_{\text{bits}}} \tag{5.3.21}$$

If the system is configured to support the maximum activation elements, it will require exactly this number of reads (meaning this many cycles) to read all corresponding products. Afterward, the architecture proceeds to load the next weight column, repeating the same read sequence.

This reading process is managed with two counters: a block counter indicating which block is currently being processed (shared across all Sync Memories) and an offset counter indicating which portion of the activation register is targeted (according to its precision). This means the read process can be described as:

- for `offset` from 0 to $\frac{In\_max}{in\_bits}$
- for `block_counter` from 0 to `G`

In this way, the architecture traverses first all the offsets (i.e., all the columns) for a given `block_counter`, and then increments the `block_counter` (moving to the next row) while resetting the offset counter to zero. As a result, the logical blocks within each memory are read row by row.

There are two additional challenges to address:

1. Determining the sign of the product
2. Handling products for odd-valued weights

Since only absolute products for even-valued weights are stored in the scratchpad memories, a mixed LUT-computation technique is applied. This technique stores absolute partial products for odd weights, while dynamically managing the sign and handling even weights through additional logic. Although there could be an interesting trade-off to shift more functionality toward the LUT (in order to pre-compute and reduce sign and odd-weight handling logic), this trade-off is not further explored in this work.

To handle the sign and odd-weight challenges, we must identify the corresponding activation element for every entry in the activation register vector, as well as the corresponding weight in the same logical position. The architecture relies on a set of counters for indexing the synchronous memories:

- `block_counter` indicates the current logical block (row-wise)
- `offset_counter` indicates the current offset within a block (column-wise)

To efficiently determine the position of data within the processing memory blocks, several runtime-calculated parameters are defined.

First, the *start row register* stores the index of the starting row of the current logical block inside memory. This is computed based on the current block counter, which tracks the block under processing, multiplied by the number of rows that each block occupies. Formally, the calculation is expressed as

$$\texttt{start\_row} = \texttt{block\_counter} \times 2^{\log_2(\texttt{block\_rows})}. \tag{5.3.22}$$

This allows the hardware to jump directly to the beginning of the memory block for faster addressing.

This defines how many activation elements can be streamed along the y vector dimension.

A crucial aspect is the packing of activations and weights within their dedicated registers. The number of activation elements that can fit into a single activation register depends on the maximum width of the register, denoted $In_{\max}$, divided by the bit-width of each activation element in_{bits}. This yields

$$\texttt{activations\_per\_reg} = \frac{In_{\max}}{in_{\text{bits}}}. \tag{5.3.23}$$

Analogously, the number of weight elements that can be packed into a weight register is

$$\texttt{weights\_per\_reg} = \frac{W_{\max}}{w_{\text{bits}}}. \tag{5.3.24}$$

Since bit-level operations are necessary to extract specific values from the registers, bit masks are also defined. These masks isolate the relevant bits of the activation or weight, depending on the configured bit-width. The mask for activations is

$$\texttt{mask\_in} = 2^{in_{\text{bits}}} - 1, \tag{5.3.25}$$

while the mask for weights is

$$\texttt{mask\_w} = 2^{w_{\text{bits}}} - 1. \tag{5.3.26}$$

These masks are applied using bitwise operations during register extraction to guarantee only the valid data bits are propagated further in the pipeline.

Overall, these runtime-calculated quantities provide the essential indexing and packing parameters for efficient and correct register and memory operation.

For reference, note:

- The least significant bit (LSB) indicates parity:
  - LSB $= 0 \quad \Rightarrow$ even
  - LSB $= 1 \quad \Rightarrow$ odd
- The most significant bit (MSB) indicates the sign:
  - MSB $= 0 \quad \Rightarrow$ positive
  - MSB $= 1 \quad \Rightarrow$ negative

Now have logic for wil be needed for evey sync mem

First, to locate the corresponding activation element, we need to identify which memory we are currently accessing, how many blocks fit in each memory, and how many blocks have been processed so far. This allows us to find the global logical block position within the synchronized memories.

Since the activation register is two-dimensional, we calculate which register contains the activation element of interest based on the number of blocks available for each activation vector dimension Y_S.

Using the global block index, we compute its coordinates $(x_{\text{th}}, y_{\text{th}})$ within the register array using the same mapping scheme as Equations (5.3.17)–(5.3.20). Then, considering the offset counter and the bitmask determined by the input bit-width, we extract the relevant bits from the register to obtain the exact activation element. This approach takes into account that each register stores multiple activation elements, depending on the precision configuration.

The activation value is then extracted by right-shifting the activation register at coordinates $(y_{\text{th}}, x_{\text{th}})$ by the appropriate offset (multiplied by the input bit-width), and masking to keep only the relevant bits:

$$\text{activation\_value} = \big(\text{ActivationReg}[y_{\text{th}}][x_{\text{th}}] \gg (\text{offset\_counter} \times In_{\text{bits}})\big) \;\&\; \text{mask}_{\text{in}} \tag{5.3.27}$$

Finally, the sign bit of the activation is extracted by selecting the most significant bit according to the input bit-width:

$$\text{sign}_a = \text{activation\_value}[In_{\text{bits}} - 1] \tag{5.3.28}$$

Next, we need to find the corresponding weight element that matches the activation element previously extracted. Since weights and activations share the same element index, the n-th activation element corresponds to the n-th weight element.

However, weights and activations typically have different bit-widths, so the location of the weight element within the weight registers differs from that of the activation element. To find the exact weight register and the offset within it, we perform the following calculations:

- Compute the linear index w_{idx} of the weight element within the flattened weight buffer, accounting for the x_{th} coordinate, the number of activations per register, the offset counter, and any dirty (invalid) elements:

$$w_{\text{idx}} = x_{\text{th}} \times \text{activations\_per\_reg} + \text{offset\_counter} + \text{dirty\_elements}$$

- Determine the weight register index $w_{\text{reg\_idx}}$ by dividing w_{idx} by the number of weights per register:

$$w_{\text{reg\_idx}} = \left\lfloor \frac{w_{\text{idx}}}{\text{weights\_per\_reg}} \right\rfloor$$

- Calculate the offset w_{offset} within the register using the modulo operation:

$$w_{\text{offset}} = w_{\text{idx}} \bmod \text{weights\_per\_reg}$$

The weight value is then extracted by right-shifting the corresponding weight register by $w_{\text{offset}} \times W_{\text{bits}}$ bits and applying a bitwise AND with the weight mask:

$$\text{weight\_value} = \big(\text{WeightReg}[w_{\text{reg\_idx}}] \gg (w_{\text{offset}} \times W_{\text{bits}})\big) \,\&\, \text{mask}_w$$

Finally, the sign bit of the weight is obtained by selecting the most significant bit of the extracted value:

$$\text{sign}_w = \text{weight\_value}[W_{\text{bits}} - 1]$$

Finally, to read the desired product from Sync Memory, we must determine the exact memory row and the offset within that row where the data resides.

Given the current logical block index block_counter and knowing how many rows each block occupies (which depends on the weight bit-width W_{bits}), we first calculate the starting row (5.3.22) of the logical block.

Next, to identify the exact row inside this block, we use the weight value as an index. Specifically, the row inside the block is calculated differently depending on whether the weight value is even or odd. This is because products exist only for even weights, and odd weights require special handling.

$$\text{row\_in\_block} = \begin{cases} \left(\frac{\text{weight\_value}}{2}\right) - 1, & \text{if weight\_value}(0) = 0 \quad \text{(even)} \\ \left(\frac{\text{weight\_value}-1}{2}\right) - 1, & \text{if weight\_value}(0) = 1 \quad \text{(odd)} \end{cases}$$

In logic form, this can be expressed as:

$$\text{row\_in\_block} = (\text{weight\_value} \gg 1) - \big(1 - \text{weight\_value}(0)\big)$$

Finally, the absolute memory row is:

$$\text{mem\_row} = \text{start\_row} + \text{row\_in\_block}$$

The product is read from memory at $(\text{mem\_idx}, \text{mem\_row})$, shifted by the appropriate offset (depending on offset_counter and product bit-width), and masked to extract the relevant bits:

$$\text{Product} = \big(\text{Mem}(\text{mem\_idx}, \text{mem\_row}) \gg (\text{offset\_counter} \times \text{product\_bits})\big) \,\&\, \text{mask}_p$$

The sign of the product is determined by the XOR of the activation and weight signs:

$$\text{product\_sign} = \text{sign}_a \oplus \text{sign}_w$$

Since the products exist only for even weights, when the weight is odd, the product corresponding to weight_value $-$ 1 is selected, and the activation is added to correct the product:

$$\text{sum} = \begin{cases} 0, & \text{if weight\_value}(0) = 0 \\ \text{product\_sign} \times |\text{activation\_value}|, & \text{if weight\_value}(0) = 1 \end{cases}$$

Or equivalently, using logical operations:

$$\text{sum} = (\text{product\_sign} \times |\text{activation\_value}|) \;\&\; \big(-\text{weight\_value}(0)\big)$$

The final signed product value read from memory is:

$$\text{Product\_Value} = \text{product\_sign} \times \text{Product} + \text{sum}$$

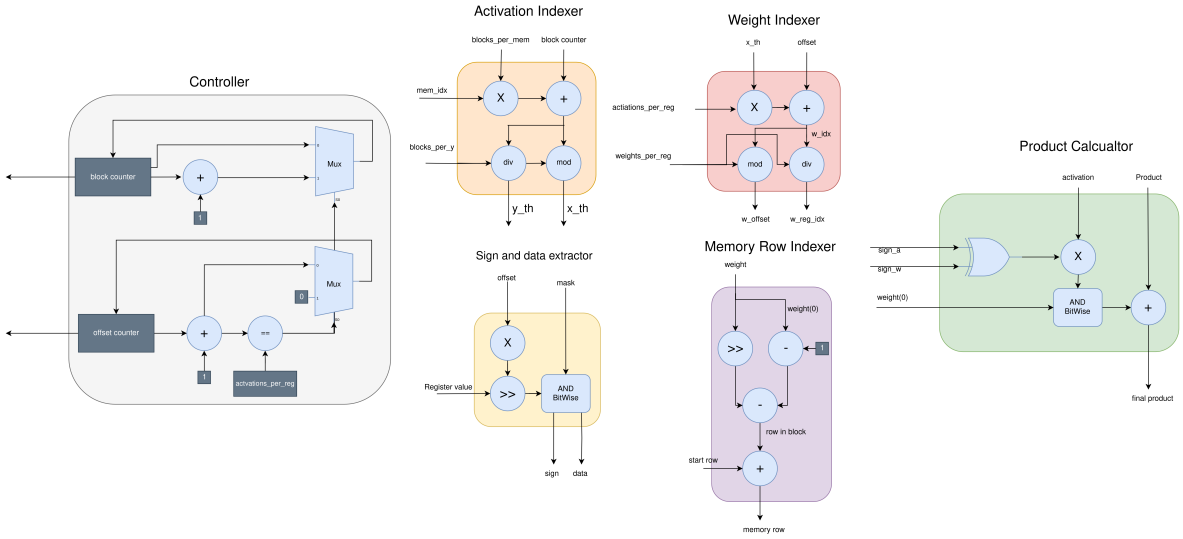


Figure 5.3.12: Select Stage Modules

In the Select stage, several cooperating modules as showed in in Figure 5.3.12 are responsible for correctly retrieving and preparing the data required for computation:

- **Controller Module:** This module is shared across all Sync Memories. Its main responsibility is to update the block and offset counters, which essentially traverse the memory row by row. It also updates the `start_row` register, which depends on the current block counter and determines the starting row of products for the selected activation block.
- **Activation Indexer Module:** Given the block and offset counters, together with the capacity of each block (how many activation elements can fit inside), this module calculates the position of the activation element of interest inside the activation register vector.
- **Weight Indexer Module:** This module guarantees that for every fetched activation element, the corresponding weight element is also retrieved. Since activations and weights may have different runtime bitwidths, the module computes both the register index inside the weight register vector and the offset within that register. Note that each weight register can store $\frac{W_{\max}}{w_{\text{bits}}}$ weight elements, where w_{bits} is dynamically reconfigurable at runtime.
- **Sign and Data Extractor Module:** This module receives a register, the bitwidth of its elements, and a precomputed mask, and extracts the relevant bits according to the current runtime bitwidth. It uses the offset counter to determine which slice of the register to select (for example, whether to extract the first 10 bits or the following 10 bits, and so on).
- **Memory Row Indexer:** This module computes which row of the Sync Memory contains the precomputed product needed, based on the weight value and the `start_row`, which indicates the starting row of products corresponding to the activation element in memory.

- **Product Calculator:** Finally, the Product Calculator takes the product retrieved from memory and determines its correct sign, based on the signs of the activation and the weight. Additionally, if the weight is odd, the module adjusts the product by adding the activation once more, because the memory stores products only for even weights; therefore, odd-weight products are computed by referencing the product for the preceding even weight plus one more activation term.

The overall calculation proceeds as follows: first, the *Activation Indexer Module* determines the corresponding activation register that contains the required activation element. Then, using the *Sign and Data Extractor* module, the system isolates both the actual activation value and its sign. This is also illustrated in Figure 5.3.13

Next, the *Weight Indexer Module* identifies the corresponding weight register and its offset within the register, again making use of the *Sign and Data Extractor* to extract the correct weight value and its sign.

Using the obtained weight element, the *Memory Row Indexer Module* calculates which row in the precomputed product memory must be accessed and issues a read request. Since the data from memory arrives with a one-cycle latency, it is necessary to delay and synchronize the activation sign, the weight sign, the least significant bit of the weight (to determine whether the weight is odd or even), the offset counter and the activation value. This ensures that when the product data arrives from memory, all required control signals are available in a consistent pipeline stage.

Finally, because each memory row may store multiple products depending on the runtime bitwidth, the system uses the offset counter together with the *Sign and Data Extractor* to isolate only the relevant portion of the memory row. The *Product Calculator Module* then finalizes the signed product, applying the correct sign and handling the special case when the weight is odd, by selecting the product corresponding to the previous even weight and adding once more the activation value.

To summarize for the select module implementation as seen from Figure 5.1.1, the select module is fed with weights and is responsible to assign the proper memory row positions to the MEMs that the desired product is stored. An analytical representation of the select module is also represented in Figure 5.3.14.

Every memory block (MEM) needs one select module to take as input the weight, corresponding to current activation block, and use it as an indexer to locate the product, a flow which can be broken down to three discrete parts.

First, the weight LSB was used to decide if the weight is odd because only the products for even weights were stored. If the weight is odd the nearest smaller even value is selected by decrementing the weight by one, else the weight is used as is. After that we map the selected weight to the correct storage index by executing right shift. Later, for odd weights the activations are accumulated to the selected product to produce the correct final value.

The second part is the row selection, and it is performed by select modules highlighted in purple as seen in Fig. 5.3.14. They determine the correct row inside the activation block according to the weight value. In order to find the row an activation block offset counter that points to the start row the the current activation block inside MEM is used. This counter is shared among all MEMs and icreases at every select stage cycle.

As a final step, the product sign is given by the XOR of the weight and activation MSBs, since only absolute values are stored. One critical observation of the select module is the hardware overhead over the addition of the adder in case of odd products, which is yet minimal when compared with the storage allocation optimization of storing half the generated products.

Once the target row of the MEM is determined, a request is sent to the synchronous memory, with the response available one cycle later. To account for this latency, delay registers are inserted to the corresponding control signals so that they remain synchronized with the returned data. This scheme is applied uniformly to every MEM, without adding extra pipeline latency since the design is fully pipelined.

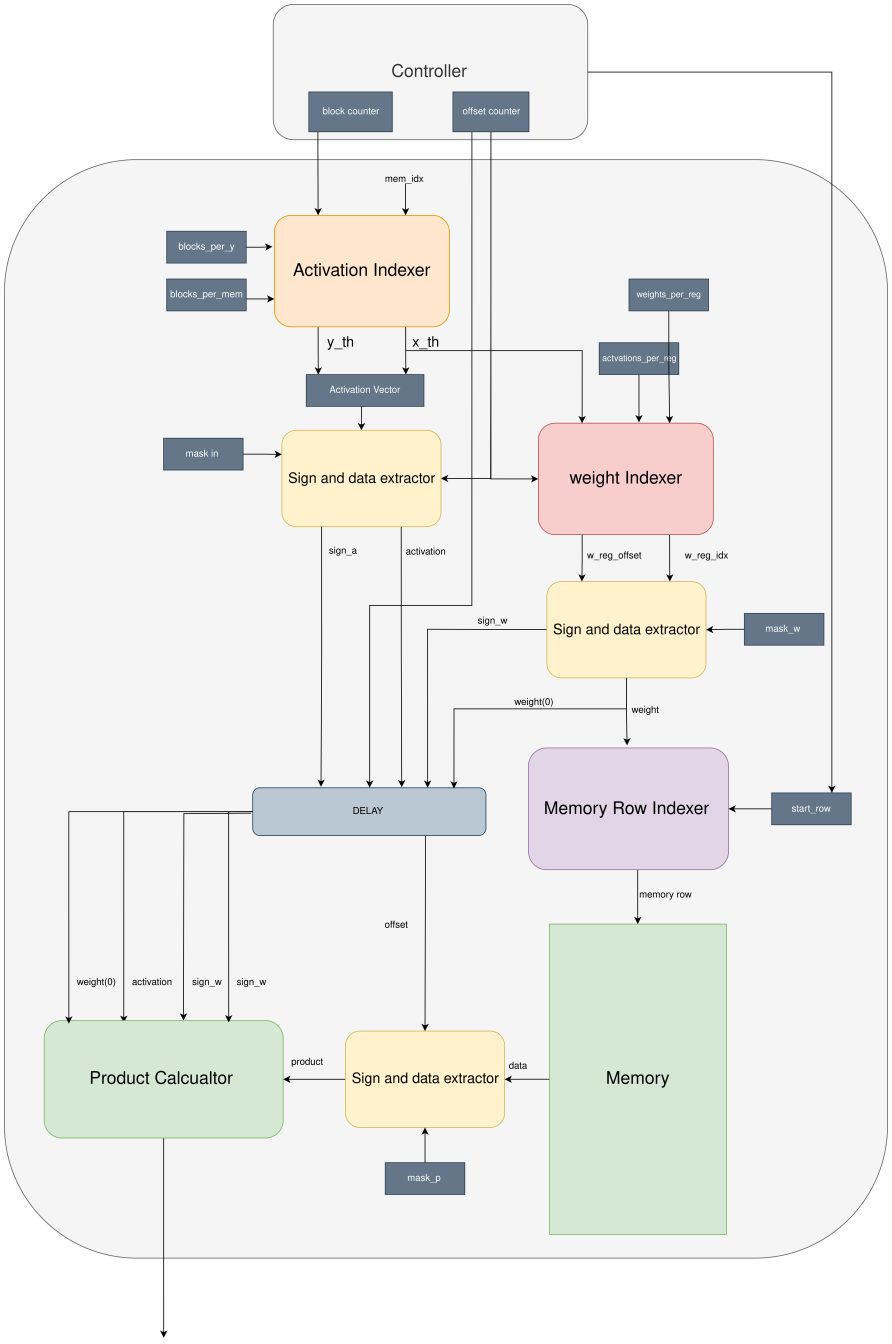


Figure 5.3.13: "Select stage" System for one Sync memory

After the generation of the selection and sign data, the pre-processed segments are propagated to the accumulator which uses an adder tree to produce the final output as seen highlighted in blue in Fig. 5.1.1 and Fig. 5.3.14.

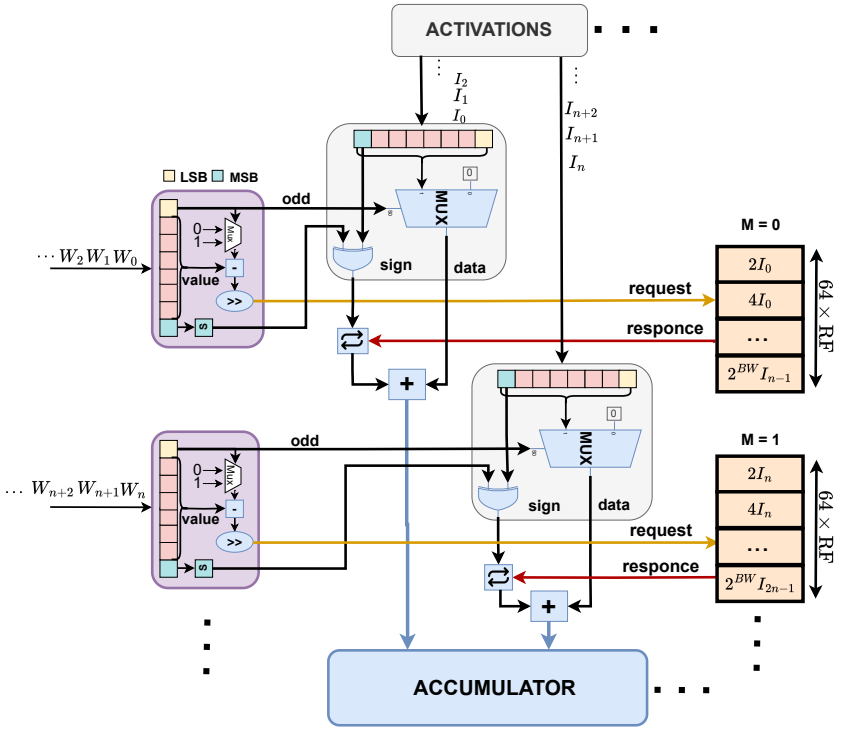


Figure 5.3.14: Detailed Representation of the Select Module for Y =1

Accumulator System Also, we include the *Accumulator Module*, whose role is to combine the signed products received from each memory. Once a signed product has been retrieved from each memory, it must be accumulated toward its corresponding output element, which is located in the same column as the weights that selected these products from memory.

If the parameter $Y_S > 1$, meaning multiple activation vectors are being processed in parallel, then all the signed products that belong to the same vector must first be combined. This is implemented through a reduction operation (an adder tree), which efficiently sums the intermediate products belonging to the same vector:

$$\text{Accumulator}(y_s) = \sum_{\text{mem}=0}^{\frac{\text{mems}}{Y_S}-1} \text{SignProduct}(y_s, \text{mem})$$

where the sum is taken over all products corresponding to the same activation vector y_s.

Finally, these partial sums are accumulated into the output elements as

$$O(y_s, \text{column}) = O(y_s, \text{column}) + \text{Accumulator}(y_s)$$

so that the contribution of each weight–activation pair is reflected in the correct output position. In this way, all products for a given activation vector are efficiently combined and stored.

In this design, we partition the total number of products, denoted by `mems`, among a set of `YS` accumulators. Each accumulator is assigned a fixed block of products of size `mems/YS`. Specifically, accumulator 0 sums the products in indices 0 to (`mems/YS` -1), accumulator 1 sums the products in indices `mems/YS` to ($2 \times$ `mems/YS` -1), and so forth, up to accumulator `YS` -1 as illustrated in Figure 5.3.15 . This static partitioning guarantees that each product is assigned to exactly one accumulator without any overlap or dynamic selection, resulting in a simplified and efficient implementation.

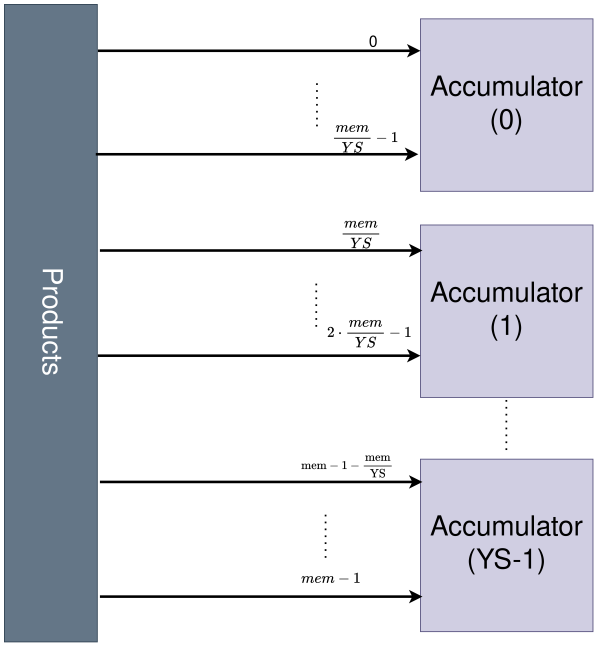


Figure 5.3.15: Accumulators Scheme

5.4 Design Optimizations

In this chapter, we further describe several optimizations used in our accelerator design to increase performance. In particular, we discuss:

- Select-and-Accumulate Pipeline
- Load Weights Ping-Pong Buffer Technique
- Increasing Memory Size

5.4.1 Select-and-Accumulate Pipeline

As we mentioned earlier, an important component of our design is the *Select-and-Accumulate* unit. This unit operates in two phases: the **Select** phase, which is responsible for drive weight elements to the physical memories to select (read) the correct products, and the **Accumulate** phase, which sums these products and stores the accumulated results in the output register.

Because we use synchronous memories, there is a one-cycle latency when reading from them. To avoid performance loss, we pipeline these two stages. In other words, instead of issuing a read request and then idling for one cycle before accumulating, we overlap the request and accumulation stages. After the initial latency penalty, the pipeline keeps both stages active on each subsequent cycle.

It is important to recall that after loading the weights corresponding to an entire column N/YS weight elements , we must perform Select-and-Accumulate operations for each of them. From equation (4.3.8), we know that each memory contains G (groups) of register blocks, and each register block includes

$\frac{In_{max}}{in_{bits}}$ logical blocks, with each logical block holding the products for one activation element. Therefore, in total, there are

$$G \times \frac{In_{max}}{in_{bits}}$$

Select-and-Accumulate operations for a weight buffer (i.e., a specific weight column) before moving to the next column.

By pipelining these two stages as we clear see in Figure 5.4.1 , we avoid repeatedly paying the one-cycle read delay. Instead, after the first cycle, the system continues to accumulate data while simultaneously issuing the next read request, thereby maximizing performance.

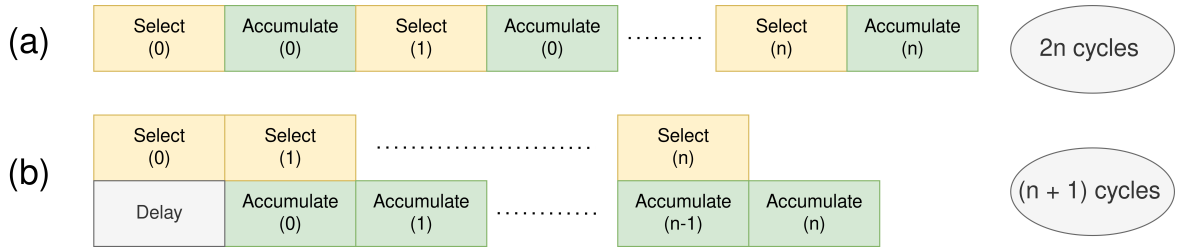


Figure 5.4.1: Select-and-Accumulate Phases: (a) Without Pipelining, (b) With Pipelining

It is important to emphasize that this pipelining optimization does **not** increase the required hardware resources, since in each cycle we still perform a single Select phase and a single Accumulate phase. The only change is adding control logic to overlap the two phases, effectively reusing the same hardware within the same cycle. As a result, if we need to read n elements per block, the total latency is reduced from approximately $2n$ cycles to only $n + 1$ cycles thanks to the pipelining.

5.4.2 Weight Ping Pong Buffers

To improve hardware performance, we use double buffering for weight storage, enabling data transfer and computation to overlap. For each activation window, weights from each column of the matrix are fetched to index the correct products, forming a producer–consumer relationship: the producer loads weights, while the consumer performs selection. Since selection usually takes fewer cycles than loading, a single buffer would force the consumer to finish before the next column loads, causing slowdowns. Double buffering allows one buffer to handle selection while the other loads the next column, then swaps roles.

It is easy to see that the basic operation of the accelerator is to load weights, use these weights for indexing, read the correct precomputed products, and accumulate them. We have described these two stages as the *Load Weights* and *Select & Accumulate* stages.

First, we need to load from main memory via DMA the $\frac{N}{Y_S}$ weights for one column of the weight matrix. This is not a one-cycle operation; it takes several cycles depending on N, Y_S, the current weight bitwidth, and the DMA parameters. Only after finishing this loading process can we use these elements, stored in the Weight Buffer, as indices in the Select & Accumulate phase.

It is important to note that the number of cycles needed to load these weights into the buffer and the number of cycles required to use all of these weights as indices form a classic producer-consumer problem, where the producer is the Load Weight stage and the consumer is the Select & Accumulate phase, with the Weight Buffer acting as their shared product store.

If we use only a single Weight Buffer, the Select & Accumulate stage must finish consuming its contents before the Load Weight stage can refill it with the next data. This leads to idle periods and underutilization of hardware resources. To solve this, we employ a **Ping-Pong Buffer** (also known as double buffering). In this technique, two buffers, $W\_buffer(0)$ and $W\_buffer(1)$, are used in an alternating fashion. While one buffer is in use for reading by the Select & Accumulate stage, the

other buffer can simultaneously be filled with new weights by the Load Weight stage as illustrated in Figure 5.4.2 .

Ping-pong buffering is a well-established technique in high-performance computing and signal processing systems, precisely because it hides data transfer latency and allows continuous data flow without stalling. It fits perfectly in our design, where reading weights from memory (which has variable latency) must be matched with a high-throughput Select & Accumulate pipeline.

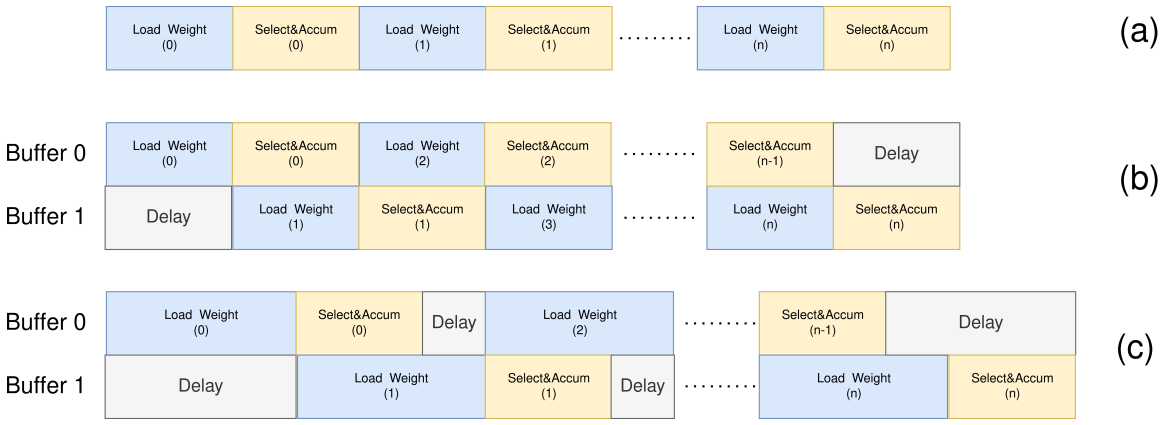


Figure 5.4.2: Load Weights and Select & Accumulate phases: (a) without double buffering, (b) with double buffering where Load Weights and Select & Accumulate require the same number of cycles, (c) with double buffering where the Load Weights stage needs more cycles than the Select & Accumulate stage.

From a performance perspective, it is important to understand that the cycles required for loading weights (c_1) and the cycles required for the Select & Accumulate phase (c_2) are generally different, and in fact depend both on generator parameters and on the runtime bitwidth configuration. If we denote by C_{out} the number of activation windows to process, then in a system without double buffering the total execution time will be

$$C_{\text{out}} \times (c_1 + c_2),$$

while with double buffering it is reduced to

$$C_{\text{out}} \times \max(c_1, c_2).$$

It is therefore crucial, as will be further analyzed, that the ratio c_1/c_2 remains as close as possible to one. When $c_1 > c_2$, the accelerator becomes essentially *memory-bound*, whereas when $c_2 > c_1$ it is instead *Select-bound* (which in practice means bounded by the synchronous memory reads). Hence, from a cycle-optimization point of view, it is very important to balance these two phases appropriately.

5.4.3 Balancing Main Memory Loads and Sync Memory Reads

The final aspect we must consider is how to enforce an appropriate exploitation of the Sync Read Memories so that the number of Select cycles (i.e., reading products from the Sync Memories) is balanced against the cycles required to load the weights from main memory. This balancing strongly depends on the bitwidth configuration. For example, when using weights with high precision (large bitwidth), each Sync Memory can store only a limited number of activation products, since each product occupies more memory. As a result, fewer products are stored, requiring fewer reads, and the Select stage completes in a minimal number of cycles, essentially stalling until the next weights arrive from main memory.

Conversely, with very low precision weights, products occupy less space, allowing us to store products for many more activations. This means the Select stage will require significantly more reads and

therefore more cycles. To achieve a balanced behavior, it is necessary to identify a switching point for each bitwidth, which can then be enforced. For instance, we could design the Sync Read Memories to be sufficiently large and leave them partially unused for small bitwidths, fully populating them only when larger bitwidths justify it. In this way, the accelerator is forced to maintain higher parallelism (more Select cycles) only when the bitwidth is high enough to benefit.

Similarly, we can artificially constrain the number of activation windows loaded into memory for small bitwidths, preventing an excessive number of Select reads and cycles. Although this approach does not fully exploit the theoretical maximum parallelism of the accelerator or the entire Sync Memory capacity, it achieves a better balance between the latency of loading weights from the main memory and the throughput of the Select stage. Consequently, this balanced approach leads to more consistent and optimal performance without increasing the accelerator size (i.e., without adding more Sync Memories). In other words, across all supported bitwidths, it is critical to match the bandwidth of the main memory with the throughput of the Sync Memory reads to achieve the highest performance.

5.5 Cycle-Accurate Performance Model

According to the generator parameters, we succeeded in developing a cycle-accurate model. This model is essentially based on a set of equations that describe how many cycles each stage requires, depending on the design parameters, as well as how many times each stage must be repeated. In this way, we obtain a performance model capable of accurately simulating the execution cycles for different inputs and varying runtime bitwidths.

We define the following generator design parameters:

- `XS`, `YS` : activation vector tiling factors
- `DMA_ID`, `DMA_bytes` : DMA engine configuration
- `In_max`, `W_max` : maximum input/weight bitwidths
- `bankMergeFactor`, `Row_Factor` : Memory Blocks architectural parameters

and the following runtime-configurable parameters:

- R_{in}, C_{in}, C_{out} : input/output activation dimensions
- `in_bits`, `w_bits` : runtime bitwidths

First of all, according to these parameters, we determine how many elements per activation vector will be consumed and processed simultaneously:

$$N' = \min(N, C_{\text{in}}) \times \min(\text{YS}, R_{\text{in}}) \tag{5.5.1}$$

where N is the ativation window as defined in Equation (5.3.6).

$$N_x = \frac{N'}{\texttt{YS}} \tag{5.5.2}$$

where N_x represents the number of elements per `YS` activation row.

Activation Loading Stage First, we must load the activation window. The total data in bits is

$$\texttt{activation\_data} = N_x \times \texttt{in\_bits} \tag{5.5.3}$$

which requires

$$\texttt{Packets\_in} = \left\lceil \frac{\texttt{activation\_data}}{\texttt{DMA\_bytes} \times 8} \right\rceil \tag{5.5.4}$$

DMA packets. Then, the number of cycles to load one activation vector is

$$\texttt{cycles}_1 = (\texttt{DMA\_ID} + HS) \times \left\lceil \frac{\texttt{Packets\_in}}{\texttt{DMA\_ID}} \right\rceil \times \texttt{YS}, \tag{5.5.5}$$

and this process repeats

$$\frac{C_{in}}{N_x} \times \frac{R_{in}}{\texttt{YS}} \tag{5.5.6}$$

times to cover the entire activation window.

In order to load an activation window of elements, we must first calculate, according to the runtime activation bitwidths, how many bits need to be transferred from main memory to the accelerator. Since each DMA request transfers `DMA_bytes`, a total of `Packets_in` requests must be issued for each activation row of size `YS`. After sending a request, there is a handshake latency, for example around 6 cycles in a TileLink UL protocol, before the response returns. If the accelerator is capable of sending up to `DMA_ID` requests in flight, then the responses arrive one after another, requiring approximately `DMA_ID` plus handshake latency cycles to receive these responses, where each response delivers `DMA_bytes` of data. If more data is needed than can fit within one group of requests, the accelerator must wait for a previous response before issuing additional requests. This procedure repeats for every activation row of size `YS` in order to load the N_x activation elements. The same mechanism continues each time a new activation window must be loaded, until all activation elements across the input channels, namely C_{in} divided by N_x, and across the matrix rows, namely R_{in} divided by `YS`, have been completely loaded.

Generate Products After loading activations, the next step is to generate the precomputed products. The products are stored row-by-row in Sync Memories. The cycles required are

$$\texttt{cycles\_2} = G \times \texttt{block\_rows} \tag{5.5.7}$$

In this equation, G represents the number of register blocks that are grouped together into a single physical memory, as defined in Equation (5.3.12). The term `block_rows` refers to the number of rows contained in each register block, as defined in Equation (5.3.1).

and this is repeated for every activation window.

$$\frac{C_{in}}{N_x} \times \frac{R_{in}}{\texttt{YS}}. \tag{5.5.8}$$

Load Weights and Select & Accumulate

(i) Load Weights

Next, we load into the weight buffer N_x elements for the current column:

The total data is

$$\texttt{weight\_data} = N_x \times w_{\text{bits}} \tag{5.5.9}$$

requiring

$$\texttt{Packets\_w} = \left\lceil \frac{\texttt{weight\_data}}{\texttt{DMA\_bytes} \times 8} \right\rceil \tag{5.5.10}$$

packets. Hence the cycles to load one buffer are

$$\texttt{cycles}_{3\_1} = (\texttt{DMA\_ID} + HS) \times \left\lceil \frac{\texttt{Packets\_w}}{\texttt{DMA\_ID}} \right\rceil. \tag{5.5.11}$$

Here, we load N_x weight elements for one column, each corresponding to Y_S activation elements, using the same mechanism as for the activation loading, in order to store them in the weight buffer.

(ii) Select & Accumulate

The next step processes the logical blocks (products) from the Sync Memories, accumulating them:

$$\texttt{cycles\_3\_2} = G \times \frac{\texttt{In\_max}}{\texttt{in\_bits}} \tag{5.5.12}$$

Since these two stages (loading weights and Select & Accumulate) are pipelined via a ping-pong buffering scheme, their effective latency is determined by the slower of the two. After loading N_x weight elements into the buffer, each of these elements must be used to select and accumulate the products stored in the corresponding logical blocks. The number of logical blocks depends on the size of the physical Sync Memory as well as the activation and, most critically, the weight bitwidths. For simplicity, we can approximate that if we have N_x weights, then we need to read N_x products from memory. Since each physical Sync Memory block can serve one read per cycle, and there are `Mems` parallel Sync Memory blocks available, the total cycles required are (5.5.12) , assuming one read per cycle per physical memory.

For a weight buffer with

$$\texttt{cycles}_3 = \max\left(\texttt{cycles}_{3\_1}, \texttt{cycles}_{3\_2}\right). \tag{5.5.13}$$

This combined stage repeats for every new activation window and for every output column, so it is performed

$$\frac{C_{in}}{N_x} \times \frac{R_{in}}{\texttt{YS}} \times C_{out} \tag{5.5.14}$$

times. This means it is the stage executed most frequently and therefore consumes the largest portion of total accelerator cycles.

Because we previously discussed loading weights into a buffer and then using this buffer to read out the values, we employ a ping-pong buffering mechanism with two alternating buffers in order to support overlapped read and write operations. These two stages — weight loading and product selection/accumulation — are therefore overlapped from a cycle perspective, so the total latency is determined by the slower of the two stages. As a result, the effective latency for this combined operation is taken as the maximum of these two stages, since this represents the performance bottleneck.

Output Store Finally, the accelerator stores the output results. The total number of output elements is

$$\texttt{output\_elements} = C_{out} \times \texttt{YS} \tag{5.5.15}$$

with data size

$$\texttt{output\_data} = \texttt{output\_elements} \times \texttt{out\_bits} \tag{5.5.16}$$

and required packets

$$\texttt{Packets\_out} = \left\lceil \frac{\texttt{output\_data}}{\texttt{DMA\_bytes} \times 8} \right\rceil \tag{5.5.17}$$

leading to cycles

$$\texttt{cycles}_4 = (\texttt{DMA\_ID} + HS) \times \left\lceil \frac{\texttt{Packets\_out}}{\texttt{DMA\_ID}} \right\rceil. \tag{5.5.18}$$

This repeats every

$$\frac{R_{in}}{\texttt{YS}} \tag{5.5.19}$$

times, since results are collected per activation window row.

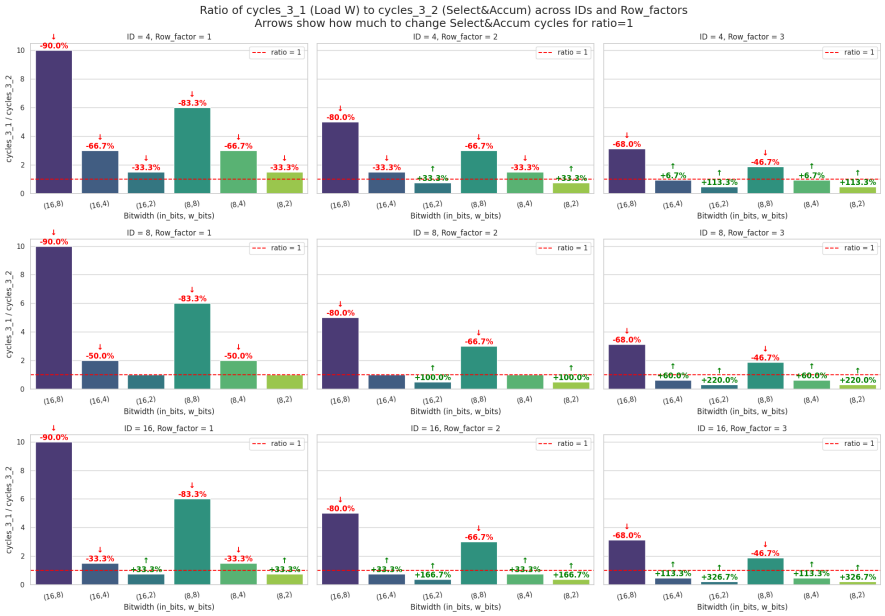


Figure 5.5.1: Ratio between weight load cycles and select & accumulate (S&A) read cycles from synchronous memories, for different `ID` and `Row_factor` parameters across various supported runtime bitwidths.

Total Cycle Estimate Hence, the total estimated execution cycles of the accelerator are

$$\boxed{\texttt{cycles}_{\text{total}} = \left(\frac{C_{in}}{N_x}\right)(\texttt{cycles}_1 + \texttt{cycles}_2) + \left(\frac{C_{in}}{N_x}\frac{R_{in}}{\texttt{YS}}C_{out}\right)\texttt{cycles}_3 + \left(\frac{R_{in}}{\texttt{YS}}\right)\texttt{cycles}_4} \tag{5.5.20}$$

It is evident from the above equation that the stage which repeats the most is the *Load Weights and Select & Accumulate* stage. As discussed in the *Optimizations* section, it is important to align the cycles used to read weights from memory—which are then written to the weight buffer—as closely as possible with the cycles used to read the products from the synchronous read memory blocks.

The main challenge, however, lies in achieving balanced performance across different bitwidth configurations. Changing the bitwidth directly affects both the weight-loading cycles into the Weight Buffer and the read cycles from the synchronous read memories. This happens because different bitwidths effectively change the activation window depending on how many activation products can fit inside the synchronous read memories.

For smaller bitwidths, more elements can fit in the same physical memory, since each element occupies fewer bits. As a result, the data transfer required for loading weights from external memory is not significantly affected becuase of the lower bitwidth. However, the number of distinct elements stored in each physical memory increases substantially, which causes the number of synchronous read cycles to rise markedly compared to the weight-loading cycles.

As we can see in the in Figure 5.5.1 , we analyze the ratio of the cycles spent on loading weights to the cycles spent on select & accumulate (S&A) per weight buffer (meaning per weight column). We examine how this ratio changes for different design parameters: the `DMA_ID` (which controls how many requests can be sent on-the-fly) and different `Row_factor` values (where increasing `Row_factor` reduces the number of synchronous-read memories, but increases the number of rows per memory).

The plots illustrate these ratios for various bitwidth pairs (`in_bits`, `w_bits`) and across different configurations of `ID` and `Row_factor`.

Chapter 6

Experimental Setup and Evaluation

As described above, the LUMAX accelerator was designed as a RocketChip Co-processor (RoCC). In this configuration, the accelerator is integrated within a System-on-Chip (SoC) architecture, with RocketChip serving as the main processor and communication taking place over the TileLink proprietary protocol.

The complete SoC, together with the LUMAX accelerator, is emulated on the ZCU106 Zynq-based FPGA platform [45]. Within this setup, the DMA engine of the LUMAX accelerator leverages the external DRAM of the ZCU106 to fetch input activations and weights, as well as to store the produced outputs.

Hardware synthesis and power analysis were performed using the Vivado 2022.1 tool flow. The evaluation of the design explores different architectural configurations, including the number of memory blocks (MEM) allocated per activation vector where MEM =XS*YS ($MEM \in \{4, 8, 16, 32\}$), as well as the number of rows per memory block, referred to as the Row Factor ($RF \in \{1, 8\}$).

Those two parameters are critical for the operation of our design, because activation vector size depends on both MEM and RF-MEM represents the number of separate MEMs,while RF is the size of each MEM. The total size of all MEMs determines the available space for products and, together with the current weight bit-width, how many activations can fit,while MEM also dictates how many elements can be selected and accumulated per cycle.

6.1 Experimental Setup

For our test case to evaluate the performance of the accelerator, we utilize the open-source LLaMA2 [32] implementation written in pure C. The availability of its source code in plain C enables easy integration into the Chipyard environment and straightforward conversion to RISC-V binaries. This allows execution with Verilator and later on FPGA with our RISC-V processor enhanced with the ROCC accelerator.

We have also modified the code to support scaling per vector , with per group scaling as the default. This approach allows us to clearly isolate and evaluate the accelerator's performance without the influence of output scaling, specifically to analyze how integer matrix multiplication operates in our implementation. Additionally, scaling per output vector is more hardware-friendly, enabling us to split the multiplication into integer matrix operations (accelerated by our hardware) followed by dequantization in pure C.

The LLaMA2 [32] implementation performs vector-matrix multiplication, where an activation vector is multiplied by a weight matrix to produce an output vector. For this use case, we set $Y_S = 1$, as multiple rows are not processed simultaneously. While the code might be adaptable for matrix-matrix multiplication, this is outside the scope of the current work.

The measured cycles correspond to the execution of all the GEMM operations (vector–matrix multiplication in our case) on our high-level implementation of LLaMA2 [32] using the TinyStories-15M dataset [34].

The integer GEMM operations and their dimenions based on the TinyStories dataset are summarized in Table 6.1. Note that $R_{in} = 1$ and the channel relationship (vector-matrix) $C_{in} \times C_{out}$ describe the input row factor and channel dimensions and int .

Table 6.1: Summary of GEMM operations in the TinyStories 15M parameters model.

Step	Operation	Matrix Dimensions	Repetitions
1	Q, K, V projections	288×288	$3\times$
2	Output projection (attention)	288×288	$1\times$
3	Feed-Forward Network (FFN)	288×768	$2\times$
4	Output projection (FFN)	768×288	$1\times$
5	Classifier	32000×288	$1\times$

With a total of approximately 15 million parameters, the model requires 180 forward passes across 6 layers. The total number of cycles is calculated as:

total cycles = forward passes $\times$ num layers $\times$ cycles for steps 1–4 + forward passes $\times$ cycles for step 5.

The number of forward passes ultimately depends on the number of input tokens.

To measure execution cycles before the design is deployed on the the ZCU106 [31], a good approach is to use Verilator, as it is easily integrated within the Chipyard environment.

6.2 Sensitivity Analysis on Generator Parameters

In this section, we perform a sensitivity analysis to evaluate how different generator parameters affect both execution cycles and hardware utilization. The focus is specifically on the integer matrix multiplication operations within one layer of LLaMA 2, based on the multiplications described in the previous section.

We evaluate the accelerator using the following **configuration**: activation bitwidths of 8 and 16 bits, weight bitwidths of 2, 4, and 8 bits, an input slice size (**XS**) of 16, and a row slice size (**YS**) fixed at 1. Since **YS** is set to 1, each memory block (MEM) effectively corresponds to a single input slice (**XS**).

We will evaluate the impact of the following parameters:

Table 6.2: Explored design parameters of the LUMAX accelerator.

Parameter	Description
XS	Input slicing factor
Row Factor	Number of rows mapped per memory block
DMA IDs	Number of DMA channels/identifiers used
Cache Mode	Cache mode enabled/disabled

The DMA interface has two parameters one that denotes how many bytes can be sent via the DMA bus per data response and another that defines how many concurrent requests can be active at a time. These two parameters are configurable in our generator. In our implementation the DMA bus is configured to handle 64-bit transfers, and the maximum number of concurrent requests is set to 4. Additionally, response FIFO has been added between the DMA inteface and the accelerator, ensuring that responses are received in order. This analysis will help us understand which architectural parameters most significantly influence performance and area, and guide optimal configurations for

LLaMA 2 with 15M parameters during inference. In terms of execution cycles, we focus exclusively on the integer vector-matrix multiplication, as the remaining operations do not vary with respect to our accelerator and remain implemented in standard C code.

6.2.1 Gemmini as Baseline

We set as baseline for our comparative study, the state-of- the-art work of the Gemmini [15] accelerator. More specifically, for fair comparisons, we implement Gemmini configurations in an iso-resource fashion w.r.t. LUMAX. The Gemmini implementations also came along with LUMAX equivalent interface characteristics, i.e. data bandwidth and DMA controllers.

However, Gemmini does not support **mixed-precision execution**, such as using different bitwidths for activations and weights, nor does it allow for **custom multipliers** within processing elements to efficiently handle low-precision data (e.g., 4-bit or 2-bit operands).

In this evaluation, we focus only on the number of cycles required for the integer vector-matrix multiplication in a single layer, based on the LLaMA2 15M TinyStory dataset workload.

Parameter(s)	**Description**
`tileRows`, `tileColumns`	Dimensions of a tile (combinational systolic unit).
`meshRows`, `meshColumns`	Dimensions of the full mesh (with pipelining between tiles).
`dataflow`	Data movement pattern: output-stationary (OS), weight-stationary (WS), or both.
`sp_banks`, `sp_capacity`, `acc_capacity`	Number of banks and capacities (in KiB) for scratchpad and accumulator memories.
`inputType`, `outputType`, `accType`	Data types for input, output, and accumulation (e.g., `SInt(8.W)`, `Float(8,24)`).
`pe_latency`	Latency of the processing element (MAC unit), important for floating-point ops.
`ld_queue_length`, `st_queue_length`, `ex_queue_length`, `rob_entries`	Queue lengths for load, store, execute operations, and reorder buffer entries.
`dma_maxbytes`, `dma_buswidth`, `mem_pipeline`	DMA configuration: max transfer size, bus width (bits), and memory pipeline depth.
`mvin_scale_args`, `mvin_scale_acc_args`, `mvin_scale_shared`	Input/accumulator scaling configuration for move-in operations; optionally share scaling hardware.

Table 6.3: Grouped Summary of Key Gemmini Generator Parameters

This is the dafault chosen parameters : In this configuration, the scratchpad and accumulator capacities are set to 256 KiB and 64 KiB respectively, with 4 banks allocated to the scratchpad and 2 banks to the accumulator. The lengths of the load, store, and execute instruction queues are configured to 8, 2, and 8 entries respectively. Additionally, the Translation Lookaside Buffer (TLB) is set with 4 entries. Scaling functionality on move-in operations is explicitly disabled, both for standard inputs and accumulators, by setting `mvin_scale_args` and `mvin_scale_acc_args` to `None`.

We configure the dataflow to support both *weight-stationary* and *output-stationary* modes, allowing us to experiment with and evaluate both execution styles. For the data types, we select a 32-bit signed integer for the output type, matching the precision of the expected final results in our accelerator. For the input types (both activations and weights), we evaluate 8-bit and 16-bit signed integers; unfortunately, 4-bit inputs are not supported in this setup.

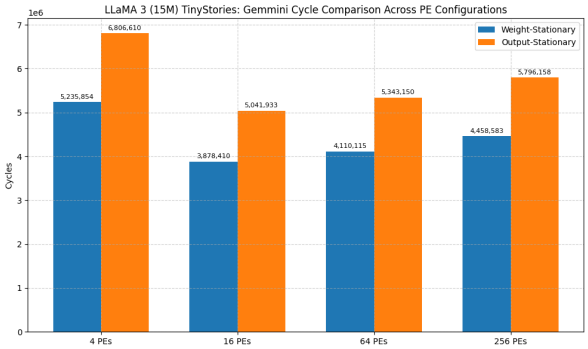


Figure 6.2.1: Gemmini int vector-matrix cycles for different PEs configurations

From the Fig. 6.2.1 above, it can be concluded that it is better to compare our design using the *weight stationary* mode, as it requires fewer cycles than the *output stationary* mode. Additionally, we will choose 16 PEs as the optimal solution for this dataset and test cases.

The cycle counts were obtained by running the testbench implemented in C from the Gemmini testbench in bare-metal mode, using the `tiled_ws_matmul` and `tiled_os_matmul` test cases. In the following plots showing cycles for our design, a **horizontal grey line** indicates the cycles of Gemmini for the specific configuration so 3,878,410 cycles.

6.2.2 Results and Discussion

6.2.3 Cycles Performace

In Table 6.4 we present the evaluation of LUMAX under different memory configurations, where the number of XS (MEM) and the Row Factor (RF) determine the effective size of the scratchpad buffers. The results are compared against the default Gemmini configuration with a 4×4 PE systolic array support INT8 activation and weights, while keeping the communication parameters and DMA configuration constant for all designs. As shown, power consumption increases with both the number of memories and the Row Factor, as a larger number of memories requires additional logic and enables more parallel operations, while a higher Row Factor demands greater storage capacity in the scratchpad. Nevertheless, even in the largest tested configurations, the power consumption remains consistently lower than Gemmini, ranging from about 0.3× to 0.6×. This highlights the energy efficiency of LUMAX across a wide design space.

The performance of the LUMAX accelerator is heavily influenced by the weight bitwidth, the Row Factor (RF), and the number of memory blocks (MEM).

Table 6.4: Power gains and Speedup comparison to Gemmini 4x4 PEs for different bitwidths

Config XS-RF	Relative Power	Cycles Speedup Per Config (I_{width}, W_{width})					
		(16,8)	(16,4)	(16,2)	(8,8)	(8,4)	(8,2)
4-1	0.291	0.2	1.7	1.73	0.4	1.7	1.82
8-1	0.317	0.4	2.35	3.56	0.73	2.29	3.6
16-1	0.329	0.72	2.36	4.56	1.17	2.4	4.63
32-1	0.402	1.21	2.38	4.56	1.21	2.4	4.64
4-8	0.329	1.05	1.79	1.83	1.13	1.80	1.81
8-8	0.382	1.12	2.35	3.55	1.17	2.38	3.6
16-8	0.462	1.15	2.37	4.56	1.21	2.4	4.63
32-8	0.683	1.2	2.37	4.56	1.22	2.4	4.73

Impact of weight bitwidth: Higher weight bitwidths (e.g., 8 bits) generally degrade performance. This is due to several factors: more cycles are required to generate the increased number of possible products, the activation window is significantly reduced, limiting parallel computation, and additional data transfers over DMA are needed. Conversely, reducing the weight bitwidth (e.g., 2 bits) provides substantial speedups, as the activation window is larger, fewer possible products are generated, and data transfer cycles decrease because fewer elements must be moved per activation. In some configurations, this results in speedups of up to 4.73×.

Effect of increasing the Row Factor (RF) at fixed XS: For configurations with a fixed number of memory blocks, increasing the Row Factor improves speedup by enlarging the activation vector. This is achieved by allowing more rows per memory block to store partial products. The benefit of a larger RF is most evident for high-precision weights, where the activation window would otherwise be small; for example, in a configuration with 16×8 weights, increasing RF from 1 to 8 (with MEM = 4) improves speedup from 0.2 to 1.05. In contrast, for lower weight bitwidths, such as 8×2, the activation window is already sufficiently large, and increasing RF yields minimal improvement (1.81–1.82). These observations indicate that further performance gains at low precision require either more memory blocks or faster weight fetching from DMA.

Effect of increasing XS at fixed RF: When the Row Factor is held constant, increasing the number of memory blocks consistently improves speedup across all weight bitwidths. This is because multiple elements can be selected per cycle, reducing the need to wait cycle by cycle for all activation blocks to be processed. However, the benefit diminishes for low-precision weights or very large RF (e.g., RF = 8), where the activation window is already large, leaving memory bandwidth or weight fetching as the limiting factor. For example, increasing MEM from 16 to 32 provides minimal additional speedup under these conditions.

Memory-bound scenarios: When further increases in MEM or RF do not yield additional performance gains, the design becomes memory-bound. In such cases, the selection and accumulation stages consume weights faster than the DMA can supply them. Although the activation window is large and multiple memory blocks are available, the system stalls waiting for additional weights. Essentially, a very large activation window can saturate the pipeline, and further improvements require either faster DMA transfers or additional parallel weight consumption.

Visualization of scaling limits: Table 6.4 highlights these effects using color coding. Green cells indicate configurations where increasing RF alone provides additional speedup due to larger activation windows, particularly for 8-bit weights. Orange cells indicate scenarios where increasing RF alone is insufficient, and additional XS are needed to enable more selects per cycle in the select-and-accumulate stage. Blue cells correspond to memory-bound designs, where further speedup is limited by DMA communication, despite having a large activation window and multiple memory blocks available.

Overall, the analysis shows that performance is determined by a careful balance of weight bitwidth, activation window size (RF), and the number of memory blocks (MEM). Optimizing one parameter in isolation can provide improvements only up to a point, after which the design becomes memory-bound and further enhancements require increased memory bandwidth or faster data movement.

6.2.4 Utilization Report

In terms of resource efficiency, Figure 6.2.2 shows the FPGA resource utilization of our LUMAX design across different configurations on the ZCU106, compared with Gemmini implementation. Initially, LUMAX is largely DSP-free, using only 8 DSPs regardless of the configuration, in contrast to Gemmini, which consumes 197 DSPs even for a relatively small systolic array. Therefore, the emphasis in our design is on LUTs and BRAMs, which are the primary resources utilized. The first four bars of Figure 6.2.2 correspond to configurations with Row Factor = 1, where each memory has only 64 rows (64×1). As a result, the memory can be implemented entirely using LUTs rather than BRAMs. In contrast, the next four bars correspond to Row factor = 8, where each memory has 512 rows (64×8). Here, BRAMs are used to store both products as well as activation and weight buffers, as the maximum activation window increases and more on-chip storage is needed for computing a single activation window.

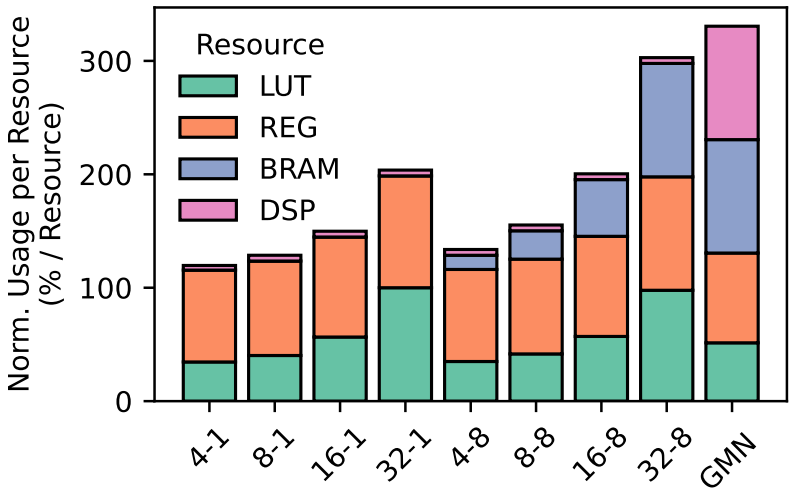


Figure 6.2.2: Normalized utilization of resources on the Xilinx ZCU106 FPGA for different XS-RF configurations of LUMAX and the Gemmini 4×4 (GMN) baseline. Each resource type is normalized with respect to its maximum across all designs, enabling a fair comparison of resource usage distribution rather than absolute counts.

In both cases, increasing the number of MEMs leads to higher LUT usage, as replicate logic is required for each MEM to implement both the product generator and the select-and-accumulate stage for parallel indexing and retrieval of products. Registers also increase proportionally, storing temporary values and maintaining synchronization across multiple MEMs. Overall, as the number of MEM increases, LUT and register usages approach the levels of Gemmini's other resources (except DSPs), but our design achieves this with improved performance and lower power consumption.

6.2.5 Different Dma parameters

Different Dma IDs

Now we fix the configuration parameters (Row Factor = 1) and observe how the number of cycles changes across different in-flight ID requests for all supported bitwidths. We evaluate two cases, with **XS = 16** and **XS = 32**. Increasing **XS** enlarges the activation window, which requires more data transfers, but it also increases the number of products read per cycle in the select stage. In this way, we can demonstrate how memory-bound problems can be mitigated simply by increasing DMA throughput.

In this subsection, we fix the design parameters to `XS` = 16 (16 memory blocks) and a DMA bus width of 8 bytes (64 bits per DMA request), and analyze the impact of increasing `ID`, which defines how many requests can be issued in flight. Memory-bound cases occur when the weights corresponding to the current activation window are transferred column-wise. Therefore, it is important to understand the amount of data loaded into the weight buffer under different scenarios.

We estimate the required data bandwidth per column of weights for different activation and weight precision combinations:

- **Activation: 16 bits, Weight: 8 bits**
  With `XS` = 16, each column of weights requires transferring $16 \times 8 = 128$ bits. Since each DMA transfer is 64 bits, an `ID` = 4 (i.e., $4 \times 64 = 256$ bits) is sufficient to transfer all required data in a single cycle. Increasing `ID` beyond this does not improve performance.
- **Activation: 8 bits, Weight: 8 bits**
  Here, each column requires $16 \times 2 \times 8 = 256$ bits. Again, `ID` = 4 provides enough bandwidth.

- **Activation: 16 bits, Weight: 4 bits**
  Now, we require $16 \times 16 \times 4 = 1024$ bits per column. Thus, the ideal `ID` would be $\frac{1024}{64} = 16$.
- **Activation: 8 bits, Weight: 4 bits**
  The requirement increases to $16 \times 16 \times 2 \times 4 = 2048$ bits, implying that an ideal `ID` would be 32.
- **Activation: 16 bits, Weight: 2 bits**
  The required bandwidth grows even more: $16 \times 64 \times 2 = 2048$ bits, leading again to an ideal `ID` = 32.

The same logic applies when increasing `XS` (e.g., from 16 to 32), which directly increases the data needed per column and thus raises the ideal `ID` accordingly.

However, in practice, we observe in Figure 6.2.3 that increasing `ID` beyond a certain threshold (e.g., above 16 or 32) no performance improvements. This is because the system's bottleneck shifts from weight loading (handled via DMA) to the select phase, where products are read from Sync Memories. Once the DMA bandwidth is sufficient to fully feed the pipeline, further increases only accelerate a non-critical path.

In other words, since the pipeline stalls waiting for slower operations (such as SyncMem read/select), optimizing the already-fast weight-loading phase has no effect. Thus, it is not meaningful to keep increasing `ID` beyond what is required to saturate the memory bus for the given precision configuration.

Increasing the DMA `ID` does not directly change the hardware resource utilization of the accelerator itself, since the DMA interface primarily handles communication between the Rocket core and the accelerator.

However, increasing `ID` increases the amount of data transferred per cycle, which can lead to higher power consumption and increased activity on the Rocket core's buses, as they must handle the larger data throughput in real-time.

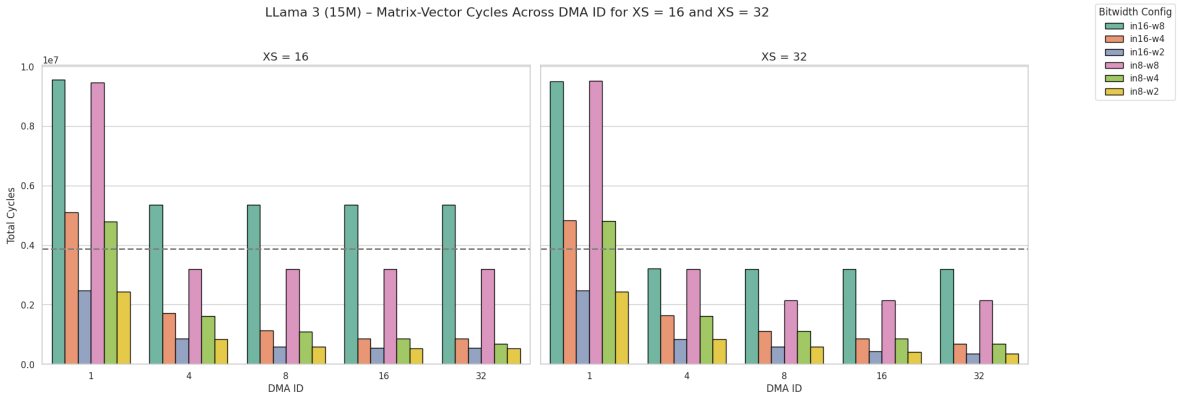


Figure 6.2.3: LUMAX Cycles for different DMA IDs and XS for activation's 4bits and weights 8 bits

6.2.6 Cache mode

In this configuration (where Row Factor = 1 ,DMA ID is 4 and DMA bytes is 8), activation's are represented with lower precision (4 bits) compared to weights (8 bits). This enables the use of a caching mechanism (`cache_4_8`), where activation's and weights are used as indices to a precomputed lookup table, avoiding the need to compute new products during execution. As a result, the activation window can reach its maximum possible size under given hardware limits, since more activation values fit into the same data footprint.

As shown in Figure 6.2.4 when the horizontal size of the compute array (`XS`) is small, the number of available synchronous read memories is also limited. In such cases, the cache mechanism is particularly

effective, allowing efficient reuse of activation-weight pairs without requiring additional logic or memory ports. As long as the DMA channel ID count is sufficiently large to sustain high throughput, caching leads to a significant reduction in total cycles and enhances overall performance.

Conversely, when `XS` is large, the hardware naturally supports a wider activation window due to increased memory access parallelism. Therefore, the performance with or without the cache becomes comparable. In this case, the benefit of caching is diminished, but the system still functions efficiently.

Overall, the cache strategy provides a low-cost and practical optimization, particularly beneficial for small `XS` values. It improves performance by reusing data in memory rather than increasing hardware utilization, making it an ideal solution in resource-constrained environments.

In utilization, enabling the cache uses just a little more resources — about 4% more LUTs and around 1% more FFs.All other resources remain essentially the same.

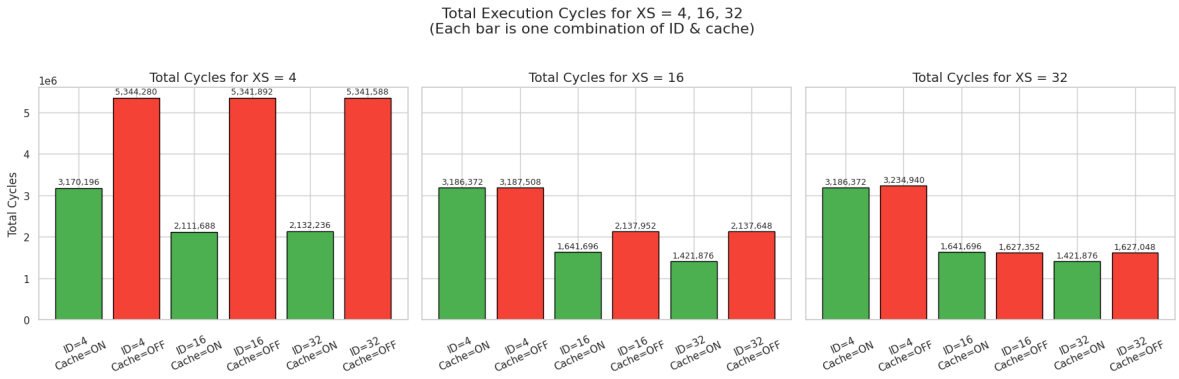


Figure 6.2.4: Cache enable vs Disable Cycles

Chapter 7

Conclusions - Future Work

This dissertation thoroughly investigated and developed LUMAX, a novel LUT-based mixed-precision accelerator specifically tailored for the efficient edge inference of large language models (LLMs). Motivated by the rising demand for deploying powerful yet resource-constrained AI applications at the edge, LUMAX addresses critical challenges in achieving high performance, energy efficiency, and flexibility for varying bit precisions.

At the core of LUMAX lies a reconfigurable quarter-size LUT architecture that significantly reduces the memory footprint and computational resource consumption compared to existing systolic array designs. By decomposing the general matrix multiplication into LUT-based product generation and leveraging a hybrid storage strategy—memorizing only even weight products and compensating odd weights via activation offsetting—LUMAX achieves notable scalability for quantized LLM workloads without compromising throughput or accuracy.

Integration with the RocketChip RISC-V SoC ecosystem enables tight hardware-software co-design and seamless deployment, while the fully pipelined, double-buffered hardware design maximizes utilization and minimizes stalls even under demanding LLM inference workloads. Compared against the state-of-the-art Gemmini systolic array accelerator, LUMAX demonstrates up to 4.7× speedup on realistic benchmarks including the LLaMA2 model and reduces LUT and DSP consumption by up to 33% and 96%, respectively. These gains translate into an approximately 70% improvement in overall energy efficiency, which is critical for deployment in energy-sensitive edge environments.

The LUMAX accelerator structure consists of three major components: the Product Generation Unit, which precomputes and stores the partial products of activations and weights using the optimized LUT scheme; the Memory Blocks (MEMs), which efficiently store these precomputed products with synchronous single-cycle read and write operations; and the Select and Accumulate Unit, responsible for indexing the correct partial products from the MEMs according to the weights and performing the final accumulation through an efficient adder tree. These modules are coordinated through buffers for activations and weights, implemented with dedicated Block RAMs in a ping-pong configuration to support double buffering and continuous data flow. This architectural decomposition supports scalability and high throughput while maintaining flexibility for various mixed-precision configurations, enabling real-time, energy-efficient LLM inference on edge devices.

This work not only advances the state of LUT-based accelerators by overcoming prior limitations related to weight bitwidth scaling and hardware/software integration, but also proposes practical design optimizations such as activation quantization alignment, flexible size configuration, and product generation techniques. The analytical modeling and comprehensive evaluation of design space trade-offs further provide valuable guidelines for future accelerator design tailored to emerging LLM workloads.

In conclusion, LUMAX offers a compelling combination of architectural innovation, practical hardware implementation, and demonstrated application-level benefits. It sets a new benchmark in mixed-

precision accelerators for large language models by providing a highly efficient, flexible, and scalable solution particularly suited for edge deployment. This work paves the way for future research to further optimize and extend LUT-based designs toward next-generation AI acceleration in resource-constrained environments.

Future extensions of the LUMAX accelerator should focus on expanding activation support to include FP16 and FP8 floating-point formats, complementing the current support for INT16 and INT8. This enhancement would enable broader compatibility with modern mixed-precision LLM workloads, many of which increasingly leverage reduced precision floating-point arithmetic to balance accuracy and efficiency.

Additionally, it is imperative to evaluate LUMAX across a wider range of LLM applications that can fully exploit low bitwidths and heterogeneous precision configurations between activations and weights. Layer-wise experiments should be conducted to understand the precise impact of mixed bitwidths on the accelerator's performance and energy efficiency.

A complete end-to-end mapping of LUMAX onto FPGA platforms is also needed to obtain actual inference latency and throughput metrics. This real hardware deployment would provide insight into the true acceleration benefits beyond simulation and enable exploration of architectural optimizations, including support for select nonlinear operations common in LLMs, which may currently pose bottlenecks.

Finally, a systematic and fair comparison against other state-of-the-art LUT-based accelerators that utilize bit-serial architectures is necessary. Such benchmarking as a feature work will yield a clearer understanding of LUMAX's relative advantages and guide future improvements in scalability, flexibility, and energy efficiency.

Bibliography

[1] J. Wu, M. Song, J. Zhao, Y. Gao, J. Li, and H. K.-H. So, "TATAA: Programmable Mixed-Precision Transformer Acceleration with a Transformable Arithmetic Architecture," *arXiv preprint*, 2024.

[2] A. Genc, C. Schmidt, L. Amarnath, H. Mao, T. Zhao, *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE/ACM, 2020.

[3] M. A. Maleki, M. Kamal, and A. Afzali-Kusha, "Heterogeneous multi-core array-based dnn accelerator," 2022.

[4] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," vol. 44, 06 2016.

[5] C. Silvano, D. Ielmini, F. Ferrandi, L. Fiorin, S. Curzel, L. Benini, F. Conti, A. Garofalo, C. Zambelli, E. Calore, S. Schifano, M. Palesi, G. Ascia, D. Patti, N. Petra, D. De Caro, L. Lavagno, T. Urso, V. Cardellini, G. C. Cardarilli, R. Birke, and S. Perri, "A survey on deep learning hardware accelerators for heterogeneous hpc platforms," *ACM Comput. Surv.*, vol. 57, June 2025.

[6] M. Loukadakis, J. Cano, and M. O'Boyle, "Accelerating deep neural networks on low power heterogeneous architectures," 01 2018.

[7] W. Wang, W. Chen, Y. Luo, Y. Long, Z. Lin, L. Zhang, B. Lin, D. Cai, and X. He, "Model compression and efficient inference for large language models: A survey," *arXiv preprint arXiv:2306.07658*, 2023.

[8] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, pp. 485–532, Apr. 2020.

[9] Y. Liu, H. He, T. Han, X. Zhang, M. Liu, J. Tian, Y. Zhang, J. Wang, X. Gao, T. Zhong, Y. Pan, S. Xu, Z. Wu, Z. Liu, X. Zhang, S. Zhang, X. Hu, T. Zhang, N. Qiang, T. Liu, and B. Ge, "Understanding llms: A comprehensive overview from training to inference," *arXiv preprint arXiv:2401.02038*, 2024.

[10] E. Frantar, Z. Fang, S. Xu, and D. Alistarh, "The art and science of quantizing large-scale models: A comprehensive review," *arXiv preprint arXiv:2210.17323*, 2022.

[11] N. Li, S. Guo, T. Zhang, M. Li, Z. Hong, Q. Zhou, X. Yuan, and H. Zhang, "The moe-empowered edge llms deployment: Architecture, challenges, and opportunities," 2025.

[12] Y. Zheng, Y. Chen, B. Qian, X. Shi, Y. Shu, and J. Chen, "A review on edge large language models: Design, execution, and applications," 2025.

[13] S. V. Kandala, P. Medaranga, and A. Varshney, "Tinyllm: A framework for training and deploying language models at the edge computers," 2024.

[14] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, "Tensorflow lite micro: Embedded machine learning on tinyml systems," 2021.

[15] P. Yu, A. Levisse, M. Gupta, T. Evenblij, G. Ansaloni, F. Catthoor, and D. Atienza, "An Energy Efficient Soft SIMD Microarchitecture and Its Application on Quantized CNNs," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 6, pp. 2200–2212, 2023.

[16] B. Zhao, Y. Wang, H. Zhang, J. Zhang, Y. Chen, and Y. Yang, "4-bit cnn quantization method with compact lut-based multiplier implementation on fpga," *IEEE Transactions on Instrumentation and Measurement*, vol. PP, pp. 1–1, 01 2023.

[17] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," 2021. .

[18] X. Tang, Y. Wang, T. Cao, L. Zhang, and Q. Chen, "Lut-nn: Empower efficient neural network inference with centroid learning and table lookup," in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pp. 741–753, ACM, 2023.

[19] G. Park, H. Kwon, J. Kim, J. Bae, B. Park, D. Lee, and Y. Lee, "FIGLUT: An energy-efficient accelerator design for FP-INT GEMM using look-up tables," in *Proceedings of the 2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2025.

[20] G. Park, B. Park, M. Kim, S. Lee, and J.-J. Kim, "Lut-gemm: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models," *arXiv preprint arXiv:2206.09557*, 2022.

[21] Z. Mo, L. Wang, J. Wei, Z. Zeng, S. Cao, L. Ma, N. Jing, T. Cao, J. Xue, F. Yang, and M. Yang, "Lut tensor core: Lookup table enables efficient low-bit llm inference acceleration," 08 2024.

[22] A. Jahanshahi, M. Wong, and D. Brooks, "Carat: Unlocking value-level parallelism for multiplier-free gemms," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*, ACM, 2024.

[23] L. Wang, L. Ma, S. Cao, Q. Zhang, J. Xue, Y. Shi, N. Zheng, Z. Miao, F. Yang, T. Cao, Y. Yang, and M. Yang, "Ladder: Enabling efficient Low-Precision deep learning computing through hardware-aware tensor transformation," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, (Santa Clara, CA), pp. 307–323, USENIX Association, July 2024.

[24] Y. Chen, A. F. AbouElhamayed, X. Dai, Y. Wang, M. Andronic, G. A. Constantinides, and M. S. Abdelfattah, "Bitmod: Bit-serial mixture-of-datatype llm acceleration," 2025.

[25] C. Guo, C. Zhang, J. Leng, Z. Liu, F. Yang, Y. Liu, M. Guo, and Y. Zhu, "Ant: Exploiting adaptive numerical data type for low-bit deep neural network quantization," 2022.

[26] A. H. Zadeh, M. Mahmoud, A. Abdelhadi, and A. Moshovos, "Mokey: enabling narrow fixed-point inference for out-of-the-box floating-point transformer models," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, p. 888–901, ACM, June 2022.

[27] J. Jang, Y. Kim, J. Lee, and J.-J. Kim, "Figna: Integer unit-based accelerator design for fp-int gemm preserving numerical accuracy," pp. 760–773, 03 2024.

[28] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. Yoo, "Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE Journal of Solid-State Circuits*, vol. 54, pp. 173–185, Jan. 2019. Publisher Copyright: © 1966-2012 IEEE.

[29] Y. Ding, B. Hou, X. Zhang, A. Lin, T. Chen, C. Y. Hao, Y. Wang, and G. Pekhimenko, "Tilus: A tile-level gpgpu programming language for low-precision computation," 2025.

[30] UC Berkeley Architecture Research Group, "Chipyard documentation." . Accessed: 2025-06-21.

[31] AMD Xilinx, "Zcu106 evaluation kit." . Accessed: 2025-06-21.

[32] A. Karpathy, "Inference llama 2 in one file of pure c," 2023. Accessed: 2025-09-01.

[33] E. Choukroun, E. Kravchik, F. Yang, and P. Kisilev, "Low-bit quantization of neural networks for efficient inference," in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, pp. 3009–3018, Oct. 2019.

[34] R. Eldan and Y. Li, "Tinystories: How small can language models be and still generate coherent text?," in *International Conference on Learning Representations (ICLR)*, 2024.

[35] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr. 2016. Accessed: 2025-06-21.

[36] I. SiFive, "Tilelink specification 1.7 draft," 2017. Accessed: 2025-06-21.

[37] "Improved-moesi cache coherence protocol," *Arabian Journal for Science and Engineering*, 2013.

[38] "Design and implementation of cache coherence protocol for high-speed multiprocessor system," in *2018 International Conference on Computing, Power and Communication Technologies (GUCON)*, 2018.

[39] "High-speed data transfer using axi protocol for image detection," *Arabian Journal for Science and Engineering*, 2023.

[40] "A novel design and implementation of imaging chip using axi protocol for mpsoc on fpga," in *Proceedings of the International Conference on Computer and Communication Engineering*, 2018.

[41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[42] G. Li, Y. Xi, J. Ding, D. Wang, Z. Luo, R. Zhang, B. Liu, C. Fan, X. Mao, and Z. Zhao, "Easy and efficient transformer: Scalable inference solution for large nlp model," *arXiv preprint arXiv:2305.19130*, 2023.

[43] T. Markidis and E. Laure, "Nvidia tensor core programmability, performance & precision," *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 522–531, 2018.

[44] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gupta, D. J. Harp, S. A. Hines, R. Jain, J. B. Jang, A. J. Jones, L. Joseph, R. K. Kanter, D. Killebrew, K. R. Koch, N. Kumar, S. Sengupta, P. M. Leong, T. Leung, Z. H. Mai, B. Patil, A. Ramakrishnan, R. Rangan, A. S. Ranganathan, M. M. Ross, A. Salek, K. Samadiani, J. Severn, G. S. Shalf, M. Shalf, S. M. Shalf, J. Smith, and D. A. Smith, "In-datacenter performance analysis of a tensor processing unit," *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

[45] OpenAI, "Gpt-4 technical report," 2023.

[46] Mistral AI, "Mistral 7b: A competitive open-weight language model," 2023.