ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αρχιτεκτονική ενεργειακά αποδοτικών διανυσματικών επεξεργαστών

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Βασίλειος Τιτόπουλος, ΑΕΜ: 70421

Επιβλέπων Καθηγητής: Γεώργιος Δημητρακόπουλος, Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δ.Π.Θ.

Ξάνθη, 2026

ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αρχιτεκτονική ενεργειακά αποδοτικών διανυσματικών επεξεργαστών

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Βασίλειος Τιτόπουλος, ΑΕΜ: 70421

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Μέλη της Συμβουλευτικής Επιτροπής:

Επιβλέπων Καθηγητής: Γεώργιος Δημητρακόπουλος, Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δημοκρίτειο Πανεπιστήμιο Θράκης

Μέλος 2: Γεώργιος Συρακούλης, Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δημοκρίτειο Πανεπιστήμιο Θράκης

Μέλος 3: Διονύσιος Πνευματικάτος, Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Εθνικό Μετσόβιο Πολυτεχνείο

Μέλη της Εξεταστικής Επιτροπής:

Μέλος 4: Χρυσόστομος Νικόπουλος, Αν. Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Κύπρου

Μέλος 5: Γεώργιος Κεραμίδας, Αν. Καθηγητής, Τμήμα Πληροφορικής, Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

Μέλος 6: Ιωάννης Βούρκας, Αν. Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δημοκρίτειο Πανεπιστήμιο Θράκης

Μέλος 7: Λάζαρος Παπαδόπουλος, Επικ. Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Δημοκρίτειο Πανεπιστήμιο Θράκης

Η παρούσα διδακτορική διατριβή υποβλήθηκε στο Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Δημοκρίτειου Πανεπιστημίου Θράκης για την απόκτηση του Διδακτορικού Τίτλου Σπουδών στις Επιστήμες του Μηχανικού.

Ξάνθη, 2026

DEMOCRITUS UNIVERSITY OF THRACE
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Architecture of Energy Efficient Vector Processors

DOCTORAL THESIS

Vasileios Titopoulos, Registration Number: 70421

COMMITTEE OF EXAMINERS

Members of the Advisory Committee:

Supervisor: Giorgos Dimitrakopoulos, Professor, Department of Electrical and Computer Engineering, Democritus University of Thrace

Member 2: Georgios Sirakoulis, Professor, Department of Electrical and Computer Engineering, Democritus University of Thrace

Member 3: Dionisios Pnevmatikatos, Professor, Department of Electrical and Computer Engineering, National Technical University of Athens

Members of the Committee of Examiners:

Member 4: Chrysostomos Nicopoulos, Assoc. Professor, Department of Electrical and Computer Engineering, University of Cyprus

Member 5: Georgios Keramidas, Assoc. Professor, School of informatics, Aristotle University of Thessaloniki

Member 6: Ioannis Vourkas, Assoc. Professor, Department of Electrical and Computer Engineering, Democritus University of Thrace

Member 7: Lazaros Papadopoulos, Assist. Professor, Department of Electrical and Computer Engineering, Democritus University of Thrace

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy (Ph.D.), Department of Electrical and Computer Engineering, Democritus University of Thrace.

Xanthi, 2026

ΑΝΑΦΟΡΑ ΣΤΗΝ ΤΗΡΗΣΗ ΤΩΝ ΑΚΑΔΗΜΑΪΚΩΝ ΑΡΧΩΝ ΔΕΟΝΤΟΛΟΓΙΑΣ

Η παρούσα εργασία με τίτλο Αρχιτεκτονική ενεργειακά αποδοτικών διανυσματικών επεξεργαστών είναι πρωτότυπη και πραγματοποιήθηκε από τον φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών με Αρ. Μητρώου ΑΕΜ 70421 στο Εργαστήριο Ολοκληρωμένων Κυκλωμάτων του τομέα Ηλεκτρονικής και Τεχνολογίας Συστημάτων Πληροφορικής, υπό την επίβλεψη του καθηγητή Γεώργιου Δημητρακόπουλου. Η συγγραφή της εργασίας/διατριβής πραγματοποιήθηκε εξολοκλήρου από τον φοιτητή Βασίλειο Τιτόπουλο, υπό την καθοδήγηση και τις υποδείξεις του επιβλέποντά του.

Βεβαιώνεται ότι, κατά την εκπόνηση και τη συγγραφή της εργασίας/διατριβής του, ο φοιτητής τήρησε τα προβλεπόμενα από τον νόμο και τον αντίστοιχο εσωτερικό κανονισμό του Τμήματος, σεβάστηκε πλήρως τις Αρχές της Ακαδημαϊκής Ηθικής και του Κώδικα Δεοντολογίας, οι οποίες απαγορεύουν την παραποίηση των ερευνητικών/πειραματικών αποτελεσμάτων, την αναφορά ψευδών στοιχείων, την κατάχρηση της διανοητικής ιδιοκτησίας τρίτων και τη λογοκλοπή και ότι έγινε με σεβασμό στις αρχές.

Περίληψη

Η διανυσματική επεξεργασία έχει επανέλθει ως μια ιδιαίτερα αποτελεσματική προσέγγιση για την επίτευξη υψηλών επιδόσεων σε εφαρμογές με παραλληλία στα δεδομένα και αυστηρούς περιορισμούς κατανάλωσης ενέργειας. Οι διανυσματικές μονάδες επεξεργασίας καταλαμβάνουν μια σημαντική ενδιάμεση θέση μεταξύ επεξεργαστών γενικού σκοπού και εξειδικευμένων επιταχυντών. Απο τη μιά, διατηρούν ένα οικείο μοντέλο προγραμματισμού ενώ από την άλλη μπορούν να προσεγγίσουν την αποδοτικότητα εξειδικευμένων επιταχυντών. Ωστόσο, η ενσωμάτωση διανυσματικής εκτέλεσης σε συστήματα με περιορισμένους πόρους παραμένει απαιτητική καθώς η περιορισμένη επιφάνεια πυριτίου και ο ρυθμός μεταφοράς δεδομένων από και προς τις μνήμες περιορίζουν το βαθμό διανυσματικής παραλληλίας που μπορεί να αξιοποιηθεί αποτελεσματικά.

Η παρούσα διατριβή αντιμετωπίζει τις παραπάνω προκλήσεις προτείνοντας νέες αρχιτεκτονικές αποδοτικών διανυσματικών επεξεργαστών RISCV οι οποίοι παρουσιάζουν αυξημένη τοπικότητα δεδομένων, μειωμένη πολυπλοκότητα στα κυκλώματα υπολογισμού και αναδιάταξης δεδομένων και βελτιστοποιημένη διανυσματική εκτέλεση κρίσιμων υπολογιστικών πυρήνων. Οι συνεισφορές της διατριβής αναπτύσσονται σε δύο συμπληρωματικές κατευθύνσεις: (μικρο)αρχιτεκτονικές καινοτομίες για αποδοτικότερη αξιοποίηση των πόρων και συσχεδίαση αλγορίθμων και υλικού για σύγχρονες εφαρμογές μηχανικής μάθησης.

Μία βασική πρόκληση στους διανυσματικούς επεξεργαστές χαμηλού κόστους είναι η μείωση του κόστους του διανυσματικού αρχείου καταχωρητών (Vector Register File – VRF), το οποίο αποτελεί διαχρονικά σημαντικό καταναλωτή εμβαδού του τσιπ και ενέργειας. Η διατριβή παρατηρεί ότι πολλοί πυρήνες μηχανικής μάθησης παρουσιάζουν μικρά και σταθερά σύνολα δεδομένων εργασίας για μεγάλα χρονικά διαστήματα και αξιοποιεί αυτή την ιδιότητα μέσω του προτεινόμενου μηχανισμού διάχυσης των καταχωρητών (Register Dispersion). Ο προτεινόμενος μηχανισμός αντικαθιστά το μεγάλο συμβατικό VRF με μια αρχιτεκτονική δομή που λειτουργεί ως κρυφή μνήμη για τους συχνά προσπελαυνόμενους διανυσματικούς καταχωρητές, ενώ οι λιγότερο συχνά προσπελαυνόμενοι καταχωρητές μεταφέρονται στην υπάρχουσα ιεραρχία κρυφής μνήμης. Με τον τρόπο αυτό μειώνονται σημαντικά η επιφάνεια και η κατανάλωση ενέργειας του VRF, διατηρώντας παράλληλα την πλήρη συμβατότητα με το σύνολο εντολών.

Επίσης, η αποδοτική διανυσματική εκτέλεση εξαρτάται σε μεγάλο βαθμό και από την αποτελεσματική αναδιάταξη των δεδομένων μέσα στα διανύσματα. Πρακτικοί υπολογιστικοί πυρήνες απαιτούν συχνά λειτουργίες αναδιάταξεις των στοιχείων των διανυσμάτων. Η εκτέλεση αυτών των λειτουργιών στο υλικό πραγματοποιείται είτε από σύνθετα εξειδικιμένα κατά περίπτωση κυκλώματα ή οδηγεί στη σπατάλη πολλαπλών κύκλων εκτέλεσης. Για την αντιμετώπιση αυτών των προβλημάτων, η διατριβή προτείνει

μια ενοποιημένη μονάδα αναδιάταξης διανυσματικών δεδομένων, η οποία υποστηρίζει αποδοτικά όλες τις εντολές αναδιάταξης του συνόλου εντολών RISCV. Η προτεινόμενη σχεδίαση παρέχει προβλέψιμη λειτουργία σταθερής καθυστέρησης για μικρά διανύσματα, ενώ υποστηρίζει υψηλού ρυθμού διοχέτευση για μεγάλα διανύσματα.

Πέρα από τους βασικούς αυτούς αρχιτεκτονικούς μηχανισμούς, η διατριβή εξετάζει πώς η στοχευμένη προσαρμογή του διανυσματικού συνόλου εντολών μπορεί να βελτιώσει την εκτέλεση εφαρμογών μηχανικής μάθησης με αραιά δεδομένα. Ο πολλαπλασιασμός αραιού με πυκνό πίνακα (Sparse-dense Matrix Multiplication - SpMM) αποτελεί βασικό υπολογιστικό πυρήνα σε κλαδεμένα CNNs και σε άλλα μοντέλα που αξιοποιούν δομημένη αραιότητα. Η εκτέλεση του SpMM οδηγεί σε ακανόνιστες προσπελάσεις δεδομένων και η διαχείριση των δεικτών μπορούν να περιορίσουν την αποτελεσματικότητα της συμβατικής διανυσματικής εκτέλεσης. Για την αντιμετώπιση αυτού του προβλήματος, η διατριβή αξιοποιεί συνήθη πρότυπα αραιότητας N:M, όπως 1:2 και 2:4, και μία στοχευμένη επέκταση του συνόλου εντολών που επιτρέπει την έμμεση προσπέλαση του διανυσματικού αρχείου καταχωρητών. Η στοχευμένη αυτή προσαρμογή του διανυσματικού συνόλου εντολών επιταχύνει σημαντικά τον εσωτερικό βρόχο του SpMM με αμελητέο κόστος υλικού, καταδεικνύοντας πώς ακόμη και μικρές επεκτάσεις του συνόλου εντολών μπορούν να γεφυρώσουν αποτελεσματικά το χάσμα μεταξύ γενικού διανυσματικού υπολογισμού και των ειδικών απαιτήσεων αραιών εφαρμογών μηχανικής μάθησης.

Η διανυσματική επεξεργασία μπορεί επίσης να συμβάλει σημαντικά στην εκτέλεση μεγάλων γλωσσικών μοντέλων σε συστήματα με περιορισμένους υπολογιστικούς και ενεργειακούς πόρους. Στο πλαίσιο αυτό, η διατριβή αναπτύσσει μια αποτελεσματική διανυσματική υλοποίηση του αλγοριθμου προσοχής FlashAttention-2, χρησιμοποιώντας μια οργάνωση των δεδομένων ανεξάρτητη που επιτρέπει την αποδοτική αντιστοίχιση του υπολογισμού της προσοχής στο διανυσματικό υλικό και τη μείωση του κόστους του ελέγχου. Μια απλή προσέγγιση της εκθετικής συνάρτησης, υλοποιήσιμη αποκλειστικά με τυπικές διανυσματικές πράξεις κινητής υποδιαστολής, μειώνει περαιτέρω το κόστος του υπολογισμού της συνάρτησης softmax που εμπεριέχεται στο μηχανισμό προσοχής. Η προσπάθεια αυτή επιφέρει σημαντική βελτίωση της απόδοσης στο υπολογισμό του μηχανισμού προσοχής, καταδεικνύοντας ότι χαμηλού κόστους διανυσματικοί επεξεργαστές μπορούν να υποστηρίξουν αποτελεσματικά σύγχρονα γλωσσικά μοντέλα, χωρίς την ανάγκη ειδικά σχεδιασμένου υλικού.

Συνολικά, η διατριβή προτείνει νέες τεχνικές για τον σχεδιασμό αποδοτικών και χαμηλού κόστους διανυσματικών επεξεργαστών. Η διάχυση των καταχωρητών μειώνει το κόστος του διανυσματικού αρχείου καταχωρητών, η ενοποιημένη μονάδα αναδιάταξης δεδομένων βελτιώνει την αποδοτικότητα της μετακίνησης δεδομένων, ενώ η συσχεδίαση αλγορίθμων μηχανικής μάθησης και αρχιτεκτονικής του υλικού επιτρέπει την αποδοτική εκτέλεση εφαρμογών μηχανικής μάθησης με αραιά δεδομένα. Οι τεχνικές αυτές μπορούν να εφαρμοστούν σε σύγχρονους διανυσματικούς επεξεργαστές RISC-V με αυστηρούς περιορισμούς σε επιφάνεια, ενέργεια και διαθέσιμη μνήμη.

Λέξεις-κλειδιά: Διανυσματικοί επεξεργαστές, Συσχεδίαση υλικού-λογισμικού, Επιταχυντές μηχανικής μάθησης, Ενεργειακή αποδοτικότητα, Ενσωματωμένα συστήματα, Αρχιτεκτονική υπολογιστών

Abstract

Vector processing has re-emerged as a highly effective approach to achieving high performance on data-parallel workloads under tight energy constraints. Vector engines occupy a valuable middle ground between general-purpose CPUs and domain-specific accelerators: they preserve a familiar programming model and ISA-level portability while approaching accelerator-like efficiency when workloads expose regular data parallelism. However, integrating vector capabilities into power- and cost-constrained systems remains challenging. Energy efficiency must be balanced against programmability, while silicon area, memory bandwidth, and limited on-chip storage constrain the amount of vector parallelism that can be effectively exploited.

This thesis addresses these challenges by proposing novel vector processor architectures that preserve the programmability and portability of standards-based vector ISAs, with particular emphasis on the RISC-V Vector extension (RVV), while systematically improving efficiency through increased data locality, reduced datapath complexity, and optimized execution strategies for critical workloads. The contributions span two complementary directions: (micro)architectural innovations for resource efficiency, and algorithm–hardware co-design for modern machine-learning workloads.

A fundamental challenge in low-cost vector processors is to reduce the cost of the Vector Register File (VRF) that is historically recognized as a major contributor to both area and energy. This thesis observes that many machine-learning kernels exhibit small and stable working sets over extended execution phases and exploits this property through Register Dispersion. The proposed mechanism replaces the large conventional VRF with a compact physical structure that acts as a demand-driven cache for frequently accessed (“hot”) vector registers, while less frequently accessed (“cold”) architectural registers are transparently dispersed into the existing cache/memory hierarchy. This substantially reduces VRF area and energy while preserving strict ISA compliance and dynamically accommodating different vector-length multiplication (LMUL) configurations and memory-traffic demands.

Beyond register storage, efficient vector execution also depends critically on data relocation within vectors. Practical kernels frequently require shuffles, transposes, and pack/unpack operations that can otherwise introduce complex dedicated datapaths or multi-cycle stalls. To address this bottleneck, the thesis proposes a Unified Vector Permutation Unit that efficiently supports all RVV permutation classes within a unified microarchitecture. The proposed design provides predictable fixed-latency operation for short vectors while supporting high-throughput pipelining for long vectors, turning vector permutations into a low-overhead and reusable execution primitive.

Beyond these core architectural mechanisms, the thesis investigates how targeted vector ISA customization can improve the execution of sparse machine-learning workloads. Sparse-dense matrix mul-

tiplication (SpMM) is a key computational primitive in pruned CNNs and other models that exploit structured sparsity, yet its irregular data accesses and index handling can limit the effectiveness of conventional vector execution. To address this, the thesis exploits common N:M sparsity patterns, such as 1:2 and 2:4, through novel row-based vector kernels. By strategically placing non-zero values and their indices across scalar and vector register files, the proposed kernels reduce data dependencies and enable deeper loop unrolling. While the pure-RVV implementation already captures significant efficiency, the thesis further introduces IndexMAC, a minimal fused-instruction extension that enables low-cost, index-driven VRF reads to improve dense-operand reuse. This targeted ISA customization substantially accelerates the inner loop of SpMM with negligible hardware cost, demonstrating how small extensions to a standard vector ISA can effectively bridge the gap between general-purpose vector processing and the specific execution requirements of sparse ML workloads.

Vector processing can also play a key role in bringing Transformer models to resource-constrained edge platforms, where the computational cost of attention remains a major bottleneck. This thesis develops a fully RVV-compliant implementation of FlashAttention-2, using vector-length-agnostic tiling to efficiently map the attention computation onto vector hardware while minimizing scalar control overhead. A lightweight exponential approximation, implemented directly using standard vector floating-point operations, further reduces the cost of the softmax computation. By efficiently vectorizing the attention kernel and reducing its computational overhead, the proposed approach delivers significant speedups for attention layers, demonstrating that low-cost vector processors can effectively support Transformer and other modern language models at the edge without relying on dedicated AI accelerators or custom hardware.

Taken together, these contributions establish a design approach for efficient, low-cost vector engines that preserves the programmability of standard vector ISAs while addressing their key hardware and workload-level limitations. Register Dispersion reduces the cost of vector state, the Unified Vector Permutation Unit addresses data-movement overheads, and targeted ISA and algorithm–hardware co-design enables efficient execution of both sparse and dense machine-learning workloads. The resulting techniques provide generalizable and scalable solutions for contemporary RISC-V vector processors and, more broadly, for vector architectures in which area, energy, and memory efficiency are first-class design constraints.

Keywords: Vector processors, Hardware-Software codesign, Machine learning accelerators, Energy efficiency, Embedded Systems, Computer Architecture.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Giorgos Dimitrakopoulos. His guidance, encouragement, and constant support have been invaluable to me, not only during my Ph.D. studies, but even before I embarked on this journey. His dedication, depth of knowledge, and passion for research have inspired me, and I feel truly fortunate to have had the chance to work under his supervision. Beyond research, his mentorship has helped me grow both as an engineer and as a person, something for which I will always remain grateful.

I would also like to thank the members of my advisory and defense committee, Professors Georgios Sirakoulis, Chrysostomos Nicopoulos, Dionisios Pnevmatikatos, Georgios Keramidas, Ioannis Vourkas, and Lazaros Papadopoulos, for taking the time to review my work and for offering constructive comments that helped improve this thesis. I owe special thanks to Professor Chrysostomos Nicopoulos for our close collaboration and for his meaningful contribution to my published research.

My heartfelt thanks also go to my labmates and friends, Dionysis, Christodoulos, Kosmas, Giorgos, and Dimitris, for all the discussions, the laughs, and the moments we shared inside and outside the lab. I am especially thankful to Kosmas and Giorgos for our excellent collaborations and the papers we worked on together. I also want to thank my friends Giannis and Kostas, whose support and companionship helped me maintain a healthy balance through the demands of research life. I am equally grateful to my fellow Ph.D. friends, Christodoulos, Kostas, Angelos, Manolis, and Karolos, whose company during quick coffee breaks and long evenings out made the challenges of research life much easier to face.

I would also like to express my deepest gratitude to my parents, Dimitris and Pagona, and to my sister, Eleni, for their love, encouragement, and unwavering support throughout all these years. Their belief in me has always given me strength, and I owe them more than words can express.

Finally, I would like to thank Codasip for funding this research and supporting the work carried out during my Ph.D. studies.

Contents

1 Introduction **1**

1.1 Modern Data-Parallel Workloads . . . 1
1.2 The Accelerator Design Space . . . 9
1.3 The Role of Vector Processing in Heterogeneous Acceleration . . . 18
1.4 Thesis Contributions . . . 19
1.5 Thesis organization . . . 21

2 Configurable Vector Microarchitecture **23**

2.1 RISC-V Vector Instruction Set Extension . . . 23
2.2 Organization of the Vector Core . . . 26
2.3 Issue Logic and Hazard Management . . . 29
2.4 Supporting Larger LMUL Configurations . . . 30
2.5 Execution Datapaths . . . 33
2.5.1 Floating-Point Execution Unit . . . 33
2.5.2 Multiplier Unit . . . 35
2.5.3 Permutation Unit . . . 36
2.6 Writeback Arbitration and Result Commitment . . . 44
2.7 Memory Subsystem . . . 45
2.7.1 Support for Vector Memory Operations . . . 46
2.7.2 Scalar–Vector Memory Ordering . . . 46
2.7.3 Design Trade-offs . . . 47
2.8 Layout of the Proposed Vector Microarchitecture . . . 48
2.9 Conclusion . . . 49

3 Structured Sparsity Acceleration on Vector Processors **51**

3.1 Introduction . . . 51
3.1.1 Unstructured vs. Structured Sparsity . . . 51
3.1.2 Vectorizing Sparse-Dense Matrix Multiplication . . . 52
3.1.3 Vector ISA customization . . . 53
3.2 Design Space Exploration of Vector Sparse×Dense Matrix Multiplication . . . 53
3.2.1 Exploring data placement in vector row-wise matrix multiplication . . . 54
3.2.2 Vector Row-wise matrix multiplication for structured-sparse data . . . 55

3.2.3 Loop unrolling row-wise vector sparse×dense matrix multiplication 57
3.3 Speeding up sparse-dense matrix multiplication with a custom instruction 58
3.3.1 The proposed vector index-multiply-accumulate **vindexmac** instruction 59
3.3.2 Sparse-Dense Matrix Multiplication using **vindexmac** 59
3.3.3 Tiled execution for larger matrices A and B 62
3.3.4 Hardware support for `vindexmac` execution 62
3.4 Experimental Evaluation 64
3.4.1 Experimental setup 64
3.4.2 Identifying peak performance using only instructions from the current RISC-V ISA 65
3.4.3 Identifying the best configuration when using the new `vindexmac` instruction 68
3.4.4 Comparisons 69
3.4.5 Performance scaling: Increasing hardware vector length or employing multiple cores 70
3.5 Related Work 72
3.6 Conclusion 74

4 Vectorized FlashAttention 75
4.1 Baseline Attention and FlashAttention algorithm 76
4.1.1 Baseline Attention with Lazy Softmax Division 76
4.1.2 FlashAttention Algorithm 77
4.2 Vectorization of FlashAttention 79
4.2.1 Exponent approximation 81
4.3 Tiling in Vectorized FlashAttention 84
4.3.1 Handling arbitrary head dimensions 84
4.3.2 Operating in parallel on multiple queries 84
4.4 Experimental results 86
4.4.1 Speedup achieved through vectorization 87
4.4.2 The effect of block size in the vectorized performance 88
4.4.3 Scaling hardware vector length 89
4.4.4 Comparison to state-of-the-art RISC-V Vector Function Evaluation Libraries . . 91
4.4.5 Accuracy of exponent approximation on LLM applications 91
4.5 Related Work 93
4.6 Conclusion 94

5 Vector Engines with Reduced Number of Physical Vector Registers 97
5.1 Introduction 97
5.2 Vector Register dispersion 99
5.2.1 Dispersion Mechanism 99
5.2.2 Supporting LMUL under Register Dispersion 100
5.3 Experimental Evaluation 101
5.3.1 Experimental Setup 102

5.3.2 Performance results 103
5.3.3 Performance comparison under equal-area VRF organizations 105
5.3.4 Performance scaling with increasing vector length multiplier and software unrolling 107
5.3.5 The effect of cache associativity on the dispersion mechanism 112
5.3.6 Hardware cost analysis 113
5.4 Related Work 117
5.5 Conclusion 118

6 Conclusions and Future Work **121**
6.1 Summary 121
6.2 Future work 122

Chapter 1

Introduction

The pursuit of energy efficiency has fundamentally reshaped computer architecture. For decades, performance scaling relied on complex general-purpose CPUs that extracted instruction-level parallelism from sequential code [113]. As power and thermal constraints have tightened, from battery-powered edge devices to large-scale datacenters, this scalar-centric approach has reached its limits. In traditional superscalar cores, instruction fetch, decode, and control can consume substantial energy relative to the arithmetic operations themselves [36].

In the AI era, optimal processor design is increasingly about doing more with less: electricity constraints demand higher performance per W, while memory constraints call for denser logic and higher performance per mm^2.

At the same time, computing workloads have changed fundamentally. Applications such as large language models, computer vision, robotics, and security increasingly rely on kernels that operate on large collections of data. Similar operations are repeatedly applied across tensors, images, signal streams, sparse matrices, or encrypted data. These workloads expose substantial data-level parallelism and favor throughput-oriented execution over the instruction-centric execution of traditional CPUs.

The emergence of such workloads has therefore driven a broader architectural shift, from control-heavy scalar execution toward architectures that can exploit parallel computation with lower control overhead and more efficient data movement [49, 46]. This shift has led to a broad spectrum of data-parallel architectures, ranging from flexible vector processors to highly specialized accelerators [8, 46],. Each offers a different balance between programmability, area, and energy efficiency.

1.1 Modern Data-Parallel Workloads

Modern workloads span a broad range of applications, each with distinct computational characteristics and performance requirements. Large language models (LLMs) are dominated by matrix and vector operations, with attention and normalization emerging as key computational kernels. Convolutional neural networks (CNNs) rely heavily on convolution and other dense tensor operations, while pruning introduces sparse computation and irregular data accesses. Robotics and signal-processing applications similarly combine regular vector operations with tight latency and energy constraints, whereas security or privacy-preserving applications, rely on computationally intensive arithmetic over large data sets. These

workloads highlight the need for programmable architectures that can efficiently support diverse data-parallel computations while remaining suitable for resource-constrained systems.

Transformers and Large Language Models. Transformers are one of the dominant model families in modern machine learning. They are widely used in natural language processing [111], vision transformers, speech and audio processing, and multimodal systems [92]. Their success has made transformer inference and training a major driver of accelerator design, especially in systems where high throughput must be delivered under strict power and memory-bandwidth constraints.

The original transformer is commonly instantiated as an *encoder–decoder* architecture: an encoder first builds contextual representations of the input, and a decoder then generates the output autoregressively while attending to those encoder representations. In contrast, many modern Large Language Models (LLMs) used for general-purpose text generation are *decoder-only*, but they reuse the same computational primitives that appear in the encoder–decoder design (attention, feed-forward layers, residual paths, and normalization) [92].

Figure 1.1 summarizes the typical encoder-decoder transformer. Both stacks begin with token embeddings augmented by positional encodings. Each encoder block applies multi-head *self-attention* over the full input sequence (token mixing), followed by a position-wise feed-forward network (channel mixing); both sub-layers are wrapped with residual connections and normalization ("Add & Norm"). Each decoder block applies *masked* self-attention to preserve causality, then performs *cross-attention* over the encoder outputs, and finally applies the feed-forward network, again with residual connections and normalization. The decoder output is mapped to vocabulary logits by a linear layer and converted to probabilities by softmax [92].

A transformer processes an input as a sequence of discrete elements, or *tokens*. In a language model, tokens correspond to pieces of text; in a vision transformer, they may correspond to image patches. Each token is represented by a dense numerical vector called an *embedding*. The number of tokens processed together is the *sequence length*, while the size of each embedding vector is the *hidden dimension*, or model dimension. Multi-head attention further partitions this representation into several *attention heads*, allowing the model to compute multiple token-to-token interaction patterns in parallel.

The core computation in transformers is built around attention and dense linear layers. Figure 1.2 illustrates the main dataflow of a self-attention block. The input token embeddings are first transformed by learned linear projections into three tensors: queries, keys, and values. The queries and keys are multiplied to produce a token-to-token score matrix, which captures how strongly each token should attend to every other token. A softmax operation normalizes these scores into attention weights. These weights are then multiplied by the value vectors to produce the output hidden states. Around this attention operation, transformer blocks also include output projections, feed-forward layers, normalization, residual connections, and activation functions.

From an architectural point of view, this dataflow is strongly GEMM-oriented. The query, key, and value projections are dense matrix multiplications; the query–key comparison is also a matrix product; and the final weighted aggregation over values is another matrix multiplication. The feed-forward layers that follow attention are typically large dense linear operations as well. Thus, the dominant arithmetic in

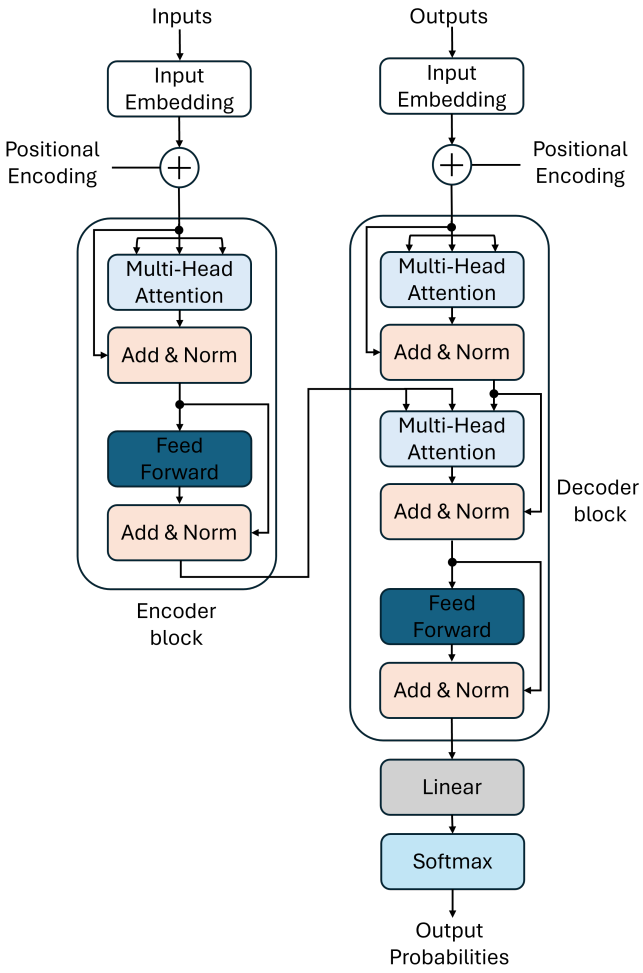


Figure 1.1: High-level transformer encoder–decoder architecture.

transformer blocks is composed of GEMM-like kernels, surrounded by vector-style operations such as softmax, normalization, residual additions, and activation functions.

This GEMM-oriented structure becomes even more important in LLMs, where model size is scaled by increasing the number of layers, hidden dimensions, attention heads, and parameters. Training and inference therefore require sustained multiply–accumulate throughput and high memory bandwidth. To reduce this cost, some transformer and LLM implementations exploit *structured sparsity*, such as block sparsity or fixed patterns like 2:4 sparsity. Structured sparsity is important architecturally because it reduces the number of effective operations while preserving enough regularity for efficient hardware execution. In contrast, arbitrary unstructured sparsity is harder to exploit because it introduces irregular indexing, load imbalance, and less predictable data movement.

Figure 1.3 shows this behavior for a BERT-base model on edge-computing platforms, where attention and linear layers occupy the majority of execution time across different input sequence lengths. These operations expose substantial data-level parallelism across tokens in the sequence, elements of the hidden dimension, independent attention heads, and batches of inputs. This is why transformer workloads are a major target for architectures that can sustain high multiply–accumulate throughput while efficiently reusing operands close to the compute units.

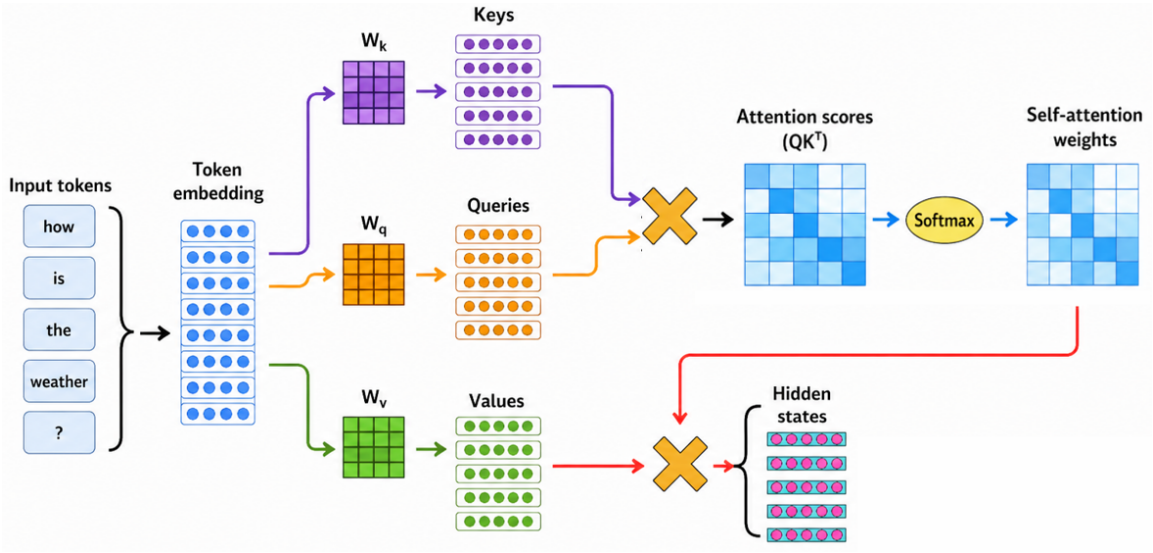


Figure 1.2: Self-attention dataflow. Input token embeddings are projected into query, key, and value tensors. Query–key matrix multiplication produces attention scores, softmax normalizes them into attention weights, and the resulting weights are multiplied with the values to produce hidden states.

However, transformer efficiency is not determined by GEMM alone. Softmax introduces reductions and nonlinear operations; normalization and residual paths require vector-style elementwise operations; and LLM decoding introduces key/value-cache accesses and memory-bandwidth pressure. Therefore, transformers motivate architectures that can accelerate dense matrix operations while also supporting the surrounding vector, reduction, and data-movement kernels.

Commercially, transformer-based LLMs are now offered by multiple providers and ecosystems. Representative examples include OpenAI's GPT-family models [125], Anthropic's Claude models [107], Google's Gemini models [26], and Meta's Llama-family models [151] (widely used both via hosted services and as open-weight deployments), as well as offerings from companies such as Microsoft (Phi) [2].

Spatial computer vision. Computer vision workloads form another major class of data-parallel computation. CNNs, which are widely used in vision and perception [74], repeatedly apply convolutional filters over image tensors. Applications include image classification, object detection, semantic segmentation. These workloads differ from transformer models because their data has strong two-dimensional spatial locality: neighboring pixels and feature-map elements are spatially correlated, and the computation is organized around local windows over the input.

A typical CNN architecture is composed of multiple layers including convolutional layers, activation functions, pooling layers, and fully connected layers, as depicted in Figure 1.4. From a computation viewpoint, a CNN is a feed-forward graph that repeatedly applies a MAC-dense linear operator to activation tensors, interleaved with vectorized elementwise transforms and local reductions [98]. In typical CNN inference and training workloads, convolutional layers account for the dominant share of arithmetic operations [60] and often constitute the largest fraction of execution time, since they are invoked repeatedly across the network and require a large number of multiply–accumulate operations over high-dimensional activation tensors and filters.

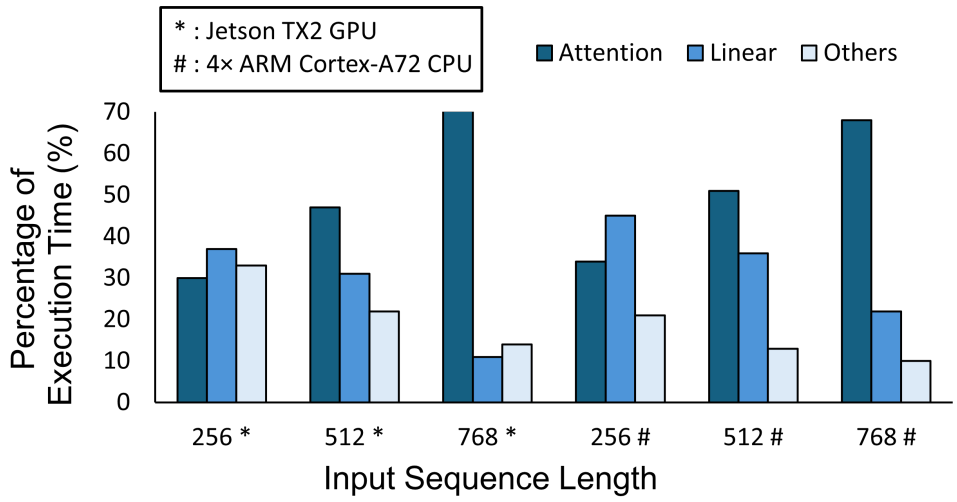


Figure 1.3: Execution-time breakdown of a BERT-base model on edge-computing platforms, highlighting the dominant contribution of attention and linear layers, as measured by [168].

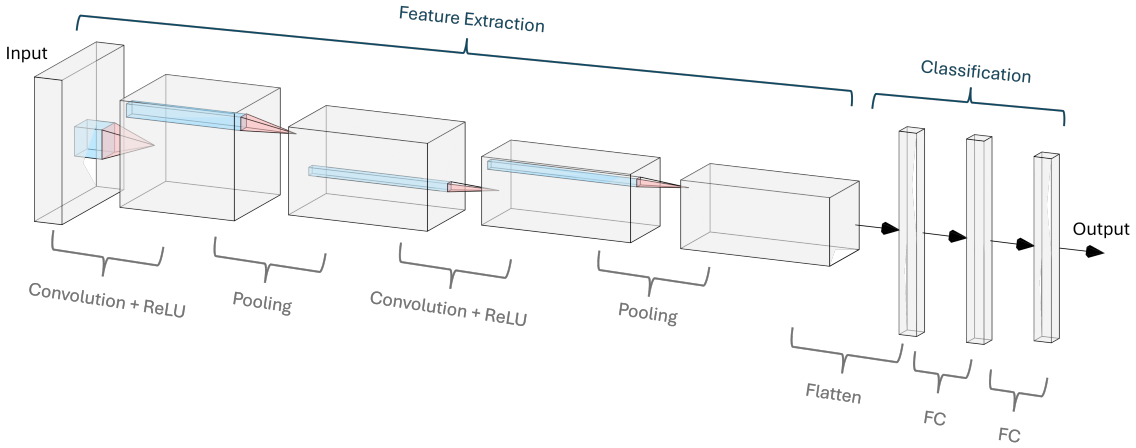


Figure 1.4: A typical CNN architecture for image classification, highlighting convolution, activation, pooling, and fully connected layers.

A convolution can be viewed as a large collection of *sliding dot products*. Each output element is produced by multiplying a small spatial–channel patch of the input by a filter and reducing the products into a single value. Across an image, and across batches and output channels, the same filter weights are reused many times, while neighboring output points access highly overlapping input regions. This windowed structure creates both massive parallelism and significant data reuse, making convolutions highly suitable for data-parallel acceleration when the hardware can exploit locality through tiling, buffering, and reuse of input activations and weights.

A convolutional layer maps an input feature map (IFM) to an output feature map (OFM). For a single sample, we represent the IFM as a 3D tensor $I \in \mathbb{R}^{C\times H\times W}$ and the OFM as $O \in \mathbb{R}^{M\times E\times F}$, where C is the number of input channels, M is the number of output channels (filters), and $H \times W$ and $E \times F$ are the input and output spatial resolutions. The layer parameters are the filter weights $W \in \mathbb{R}^{M\times C\times R\times S}$,

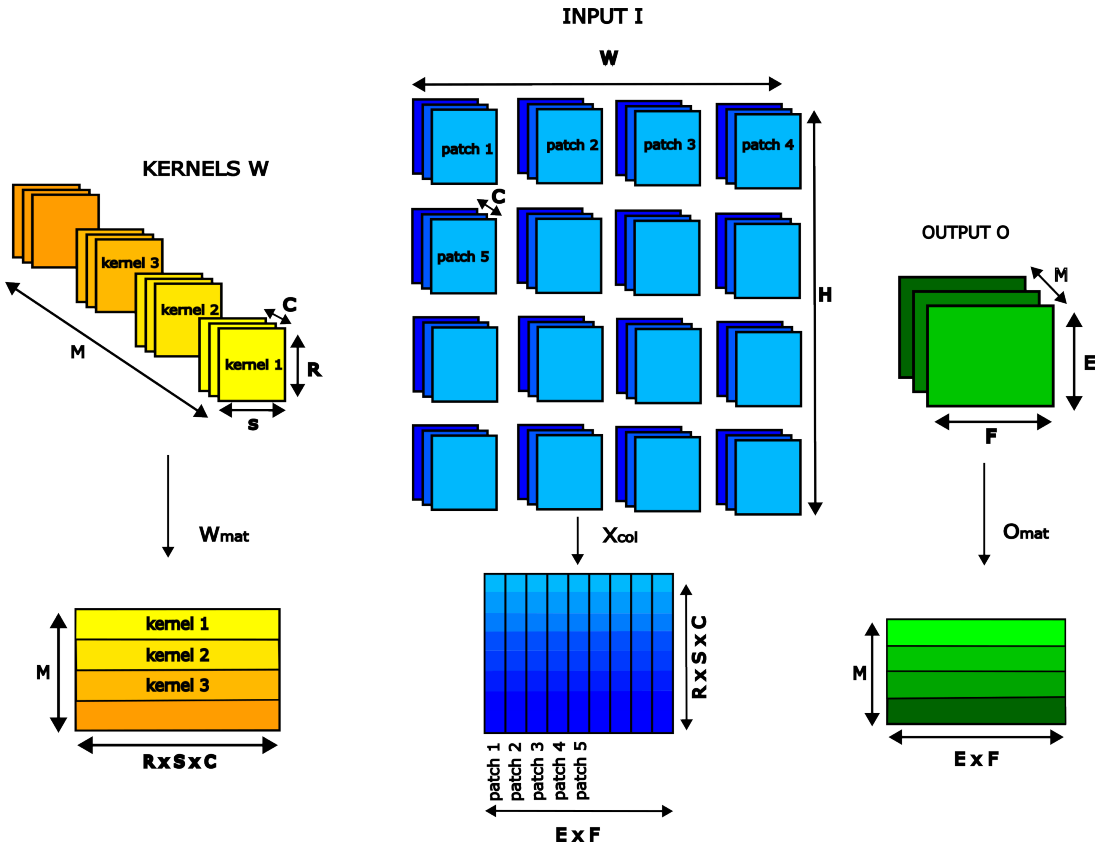


Figure 1.5: Convolution lowering to matrix multiplication.

with kernel size $R \times S$, which lead the convolution to be computed as

$$O[m, y, x] \;=\; \sum_{c=0}^{C-1} \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} I[c,\; y+i,\; x+j] \cdot W[m, c, i, j]. \tag{1.1}$$

This expression makes explicit that each output element is a reduction of multiply–accumulates: an inner product between a flattened receptive-field vector of length RSC and a flattened filter vector. Across all (m, y, x), the layer executes a very large number of such dot products. This structure exposes data-level parallelism across output channels, spatial positions, batches, and reduction dimensions. It also creates significant reuse, because the same filter weights are applied repeatedly across many spatial positions, while neighboring output elements access overlapping input regions.

The same computation can be reorganized into an explicit matrix multiplication [154], which is one of the key enablers for accelerator mappings. By collecting all receptive fields into a matrix $X_{\text{col}} \in \mathbb{R}^{(RSC)\times(EF)}$ (the classical *im2col* view, or an equivalent implicit-GEMM layout) and reshaping the filters into $W_{\text{mat}} \in \mathbb{R}^{M\times(RSC)}$, the layer becomes

$$O_{\text{mat}} \;=\; W_{\text{mat}} X_{\text{col}}, \tag{1.2}$$

where $O_{\text{mat}} \in \mathbb{R}^{M\times(EF)}$ is the OFM flattened over (y, x). This exposes convolution as dense GEMM with a simple epilogue and allows direct use of MAC-dense engines designed to maximize operand reuse

and sustain high throughput [98]. The transformation of convolution into matrix multiplication using im2col is presented in Figure 1.5. However, explicitly materializing X_{col} can increase memory traffic and storage footprint because overlapping input windows are duplicated in the lowered matrix. Practical implementations therefore often use implicit-GEMM formulations that generate the required tiles on demand, preserving the GEMM mapping while reducing unnecessary data movement.

Although the GEMM view is useful, computer-vision workloads are not purely dense matrix multiplication. Activation functions such as ReLU introduce dynamic activation sparsity by setting negative values to zero, so a significant fraction of feature-map elements may become zero during inference. This creates opportunities for hardware to save energy by bypassing zero-valued operations, reducing memory traffic, or clock-gating inactive datapaths. Pooling, normalization, layout transformations, and quantized operators also appear throughout practical networks. Therefore, CNNs motivate accelerators that can provide high throughput for convolution and GEMM while also supporting vector operations, reductions, sparsity, and data-layout movement.

TinyML and edge inference. TinyML and edge inference push machine learning into deeply constrained devices, such as microcontrollers, wearable sensors, and battery-powered embedded platforms. Representative applications include wake-word detection, vibration-based predictive maintenance, activity recognition, wearable health monitoring, and low-power sensor analytics. These systems operate under strict power and memory budgets, often with only small on-chip SRAMs and no large cache hierarchy.

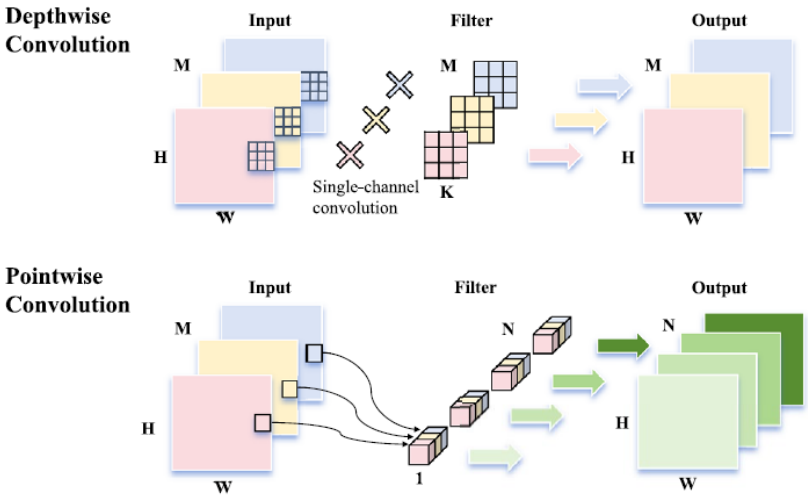


Figure 1.6: Depthwise separable convolution. The depthwise stage applies independent spatial filters per channel, while the pointwise stage uses 1×1 convolutions to mix channels. Taken from [82].

In this regime, large dense GEMM kernels are often reduced or fragmented. Efficient models commonly rely on compact operators such as depthwise separable convolutions, pointwise convolutions, matrix–vector operations, and aggressive low-precision arithmetic. A depthwise separable convolution decomposes a standard convolution into two stages, as illustrated in Figure 1.6. The depthwise stage applies one spatial filter independently to each input channel, while the pointwise stage uses 1×1 convo-

lutions to mix information across channels. This decomposition greatly reduces the number of parameters and multiply–accumulate operations compared with a full convolution, making it attractive for edge models such as MobileNet-style networks [51].

However, this efficiency changes the hardware requirements. Instead of a few large dense GEMM kernels, TinyML models often contain many smaller operations with lower arithmetic intensity and tighter memory constraints. Quantization to INT8 or INT4 and model pruning are widely used to reduce storage and computation. The resulting workloads still expose data-level parallelism, but the parallelism is shorter, more fragmented, and more sensitive to memory movement. This favors low-power architectures that can exploit vector or sub-word parallelism without requiring large systolic arrays or massive thread-level parallelism.

Autonomous robotics and spatial computing. Robotics and spatial-computing workloads require agents to interpret and interact with the physical world in real time. Applications include simultaneous localization and mapping (SLAM), LiDAR point-cloud processing, visual odometry, autonomous drone navigation, and sensor fusion [18, 30]. These workloads often combine dense numerical kernels with sparse, graph-like, or geometry-dependent data structures.

Unlike CNNs and transformer blocks, many robotics kernels are not dominated by large dense GEMM operations. Instead, they frequently involve sparse matrix computations, irregular neighborhood operations, and graph-based optimization. Sparsity arises both in the input data and in the mathematical structure of the algorithms. For example, a LiDAR point cloud samples only a small subset of the surrounding 3D space, making the input inherently sparse and irregular. Similarly, SLAM back-end algorithms such as pose-graph optimization or bundle adjustment construct large sparse linear systems, because each observation typically constrains only a small subset of the robot state variables.

A common pattern is sparse matrix–vector or sparse matrix–matrix computation, where only the nonzero entries are stored and processed. Although this reduces the total number of arithmetic operations, it also makes execution more difficult. The sparsity pattern is often dynamic, since it depends on the current scene, sensor measurements, and motion of the robot. As a result, efficient execution requires support for indexed memory accesses, gather/scatter operations, irregular reductions, and high cache and memory bandwidth. These workloads still expose data-level parallelism, but the parallelism is less regular than in dense ML kernels and is more tightly coupled to the structure of the input data and the sparsity pattern of the underlying graph or matrix.

Digital signal processing and streaming sensors. Digital signal processing (DSP) and streaming-sensor workloads operate over continuous streams of samples under strict real-time constraints. Examples include radar and sonar processing, wireless baseband processing, audio filtering, sensor fusion, and event-driven sensing [129]. Their dominant primitives are often FFTs, FIR and IIR filters, complex arithmetic, reductions, and stencil-like operations rather than dense GEMM.

These workloads expose data-level parallelism across samples, channels, frequency bins, antennas, or time windows. However, they also require predictable latency and efficient streaming behavior. For these applications, the ability to sustain regular dataflow and avoid unnecessary control overhead can be

more important than peak arithmetic throughput. This motivates architectures that support deterministic pipelines, vector-style execution, and efficient local buffering.

Security and networking. Security and networking workloads also contain abundant data-level parallelism. Cryptographic algorithms, hashing functions, packet processing, compression, and error-correcting codes apply repeated arithmetic, logical, and permutation operations over blocks, packets, or streams of data. Although these workloads are usually not GEMM-oriented, they benefit from executing the same operation over many independent data elements.

For example, block ciphers can encrypt multiple independent blocks in parallel, hashing algorithms repeatedly apply structured rounds over message blocks, and packet-processing pipelines often perform similar operations over many packets or headers. These workloads motivate support for vector integer arithmetic, bitwise operations, permutations, and efficient memory access. They also show that data-level parallelism is important beyond machine learning.

Across all these domains, the common theme is that performance and energy efficiency depend on exploiting repeated operations over many data elements. Some workloads, such as transformers and CNNs, are strongly GEMM-oriented and demand high MAC throughput. Others, such as robotics, DSP, TinyML, and security, expose parallelism through sparse, streaming, low-precision, or bit-level operations. Taken together, these domains show that modern data-parallel workloads differ not only in arithmetic intensity, but also in the regularity of their data layout, their reliance on dense GEMM, and the form of sparsity they expose. Therefore, modern processors and accelerators must not only provide raw arithmetic throughput, but also support flexible data-parallel execution across a range of data layouts, precisions, and memory-access patterns.

1.2 The Accelerator Design Space

The shift toward data-level parallelism has led to a broad range of accelerator architectures. These architectures differ in how much computation is specialized in hardware and how much flexibility is left to software. At a high level, they can be viewed along a spectrum from fixed-function to programmable execution [76, 75].

Non-programmable accelerators implement a specific operation, or a narrow class of operations, directly in hardware. Examples include video codecs, cryptographic engines, image-signal-processing blocks, and specialized matrix-multiply engines. By tailoring the datapath, control logic, and data movement to a known computation, these accelerators can achieve very high energy efficiency while avoiding much of the overhead of instruction fetch, decode, dynamic scheduling, and general-purpose control.

This efficiency, however, comes at the cost of flexibility. A fixed-function accelerator is effective only when the workload matches the assumptions built into the hardware. Changes in algorithms, data layouts, tensor shapes, operators, or numerical precision can reduce utilization or require a new accelerator. This limitation is particularly relevant in machine-learning and edge systems, where workloads and their computational requirements evolve rapidly.

Programmable accelerators take the opposite approach by exposing a software-visible execution model that can support different kernels on the same hardware. This may be provided through an in-

struction set, a thread-based programming model, or a configurable dataflow. While such architectures retain some overhead for instruction sequencing, control, and flexible memory access, they can adapt to a much broader range of computations.

The choice between fixed-function and programmable acceleration therefore represents a fundamental tradeoff between specialization and flexibility. Fixed-function designs maximize efficiency for well-defined computations, whereas programmable designs sacrifice some specialization to support a wider range of workloads. This tradeoff becomes particularly important when a single application combines diverse kernels, such as dense and sparse matrix operations, reductions, normalization, data rearrangement, low-precision arithmetic, and streaming computations.

Within this spectrum, data-parallel accelerators form a broad architectural design space. The main approaches relevant to this thesis include SIMT (Single Instruction Multiple Thread) architectures, systolic arrays and tensor accelerators, coarse-grained reconfigurable architectures, and vector processors and SIMD (Single Instruction Multiple Data) extensions. They differ in how parallelism is exposed to software, how much control is handled by hardware, how data movement is organized, and how effectively they handle different workload characteristics.

A useful way to understand these architectures is in terms of the balance between programmability and specialization. SIMT architectures expose massive thread-level parallelism and execute groups of threads in lockstep. Systolic arrays and tensor accelerators move toward greater specialization by implementing spatial dataflows optimized for dense matrix operations. Coarse-grained reconfigurable architectures (CGRAs) provide programmable spatial execution through configurable functional units and interconnects. Vector processors occupy a distinctive position by exposing data parallelism directly through the ISA while retaining a close connection to scalar control and allowing the same software model to scale across implementations with different hardware widths. SIMD extensions represent a more tightly integrated alternative, adding fixed-width data-parallel execution to scalar processors through dedicated registers and execution units.

SIMT architectures. SIMT architectures, most prominently GPUs, expose parallelism through large numbers of scalar-looking threads. Hardware groups these threads into lockstep execution batches, commonly called warps or wavefronts, and executes them over wide datapaths. In this way, the programmer sees a thread-based programming model, while the hardware exploits data-parallel execution internally.

Figure 1.7(a) illustrates a GPU at a high level. Rather than a small number of latency-optimized cores, a GPU is organized as many throughput-oriented streaming multiprocessors (SMs), each capable of executing large numbers of threads concurrently. These multiprocessors share access to a broader memory hierarchy and rely on large register files and on-chip local memories to keep many thread groups resident on chip. This organization is designed not to minimize the latency of a single thread, but to maximize overall throughput by keeping the arithmetic units busy across many parallel threads. Figure 1.7(b) shows the corresponding CUDA programming hierarchy: a kernel launch defines a grid, which is composed of multiple thread blocks; each block is assigned to one SM and shares the local resources of that multiprocessor, such as the register file and shared memory; and threads within a block are grouped into warps, which form the basic scheduling unit.

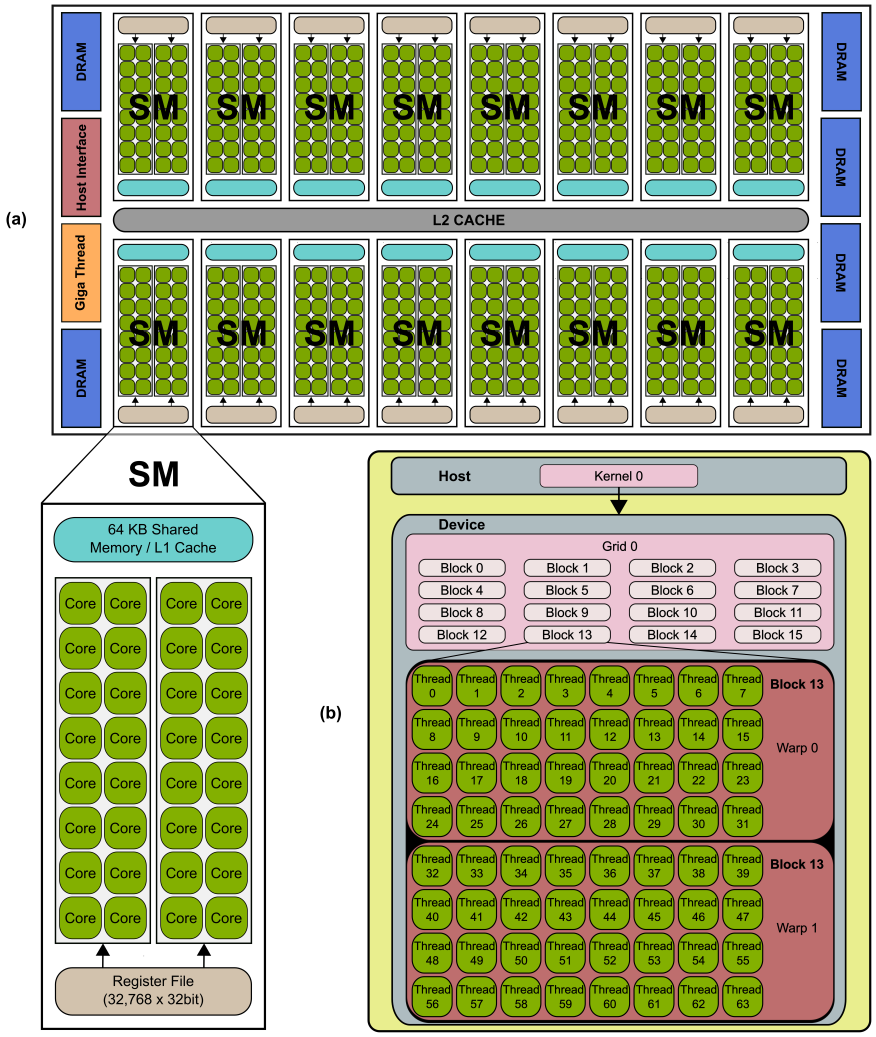


Figure 1.7: Representative SIMT GPU organization and programming hierarchy. (a) A GPU consists of multiple throughput-oriented streaming multiprocessors (SMs), each containing many arithmetic cores and on-chip storage resources. (b) Under the CUDA execution model, a kernel launches a grid of thread blocks, each block is assigned to an SM, and threads within a block are executed in warps. Taken from [47]

For dense numerical kernels such as matrix multiplication, GPU execution is typically organized hierarchically. As illustrated in Figure 1.8, the problem is first decomposed into tiles, and each tile is assigned to a thread block. A block is then scheduled onto a streaming multiprocessor, where its threads cooperatively load submatrices into low-latency shared memory and compute partial results in parallel. This organization allows the GPU to exploit parallelism at multiple levels: across thread blocks, across threads within a block, and across arithmetic lanes within the underlying datapath. It also improves data reuse, since matrix tiles loaded once into shared memory can be reused by many threads before being evicted.

A defining feature of SIMT architectures is latency hiding through massive multithreading. When one warp stalls on a memory access, the hardware scheduler can quickly switch to another ready warp whose operands are available. This makes SIMT architectures highly effective for large, throughput-oriented workloads with regular control flow and coalesced memory accesses. However, the same lockstep exe-

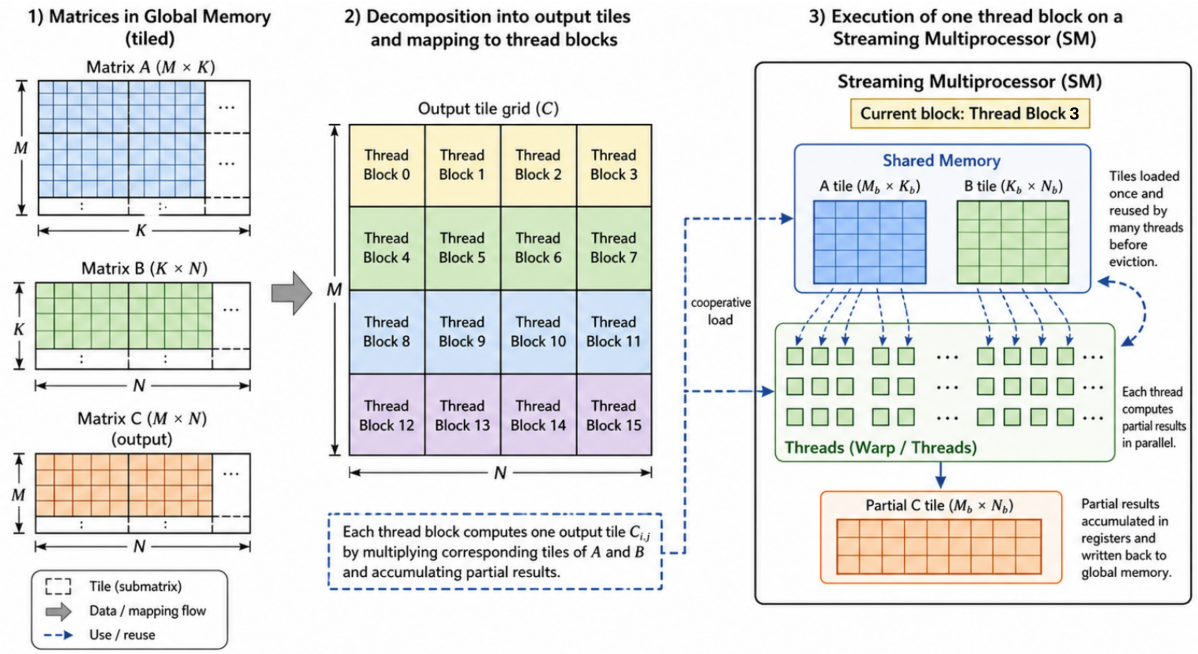


Figure 1.8: Hierarchical mapping of dense matrix multiplication onto a GPU. The computation is decomposed into tiles, assigned to thread blocks, and executed cooperatively by threads that reuse matrix tiles through on-chip shared memory.

cution also introduces characteristic inefficiencies. If threads within a warp follow different control-flow paths, execution must be partially serialized, reducing utilization. Likewise, irregular and uncoalesced memory accesses can fragment memory transactions and reduce effective bandwidth.

As a result, SIMT architectures provide very high throughput for regular, massively parallel workloads, which is why GPUs [27, 128, 139] dominate many dense machine-learning and graphics workloads. At the same time, their efficiency depends strongly on workload regularity, occupancy, and memory-access behavior. Relative to vector processors, GPUs offer greater thread-level scale and a mature software ecosystem, but they expose less explicit ISA-level control over data-parallel execution and are more sensitive to divergence and memory coalescing.

Systolic arrays and tensor accelerators. Systolic arrays specialize aggressively for dense matrix computation. They organize multiply–accumulate units into regular two-dimensional grids and move operands rhythmically through the array. During the compute phase, data reuse is captured by the spatial structure of the array rather than by repeatedly fetching operands from a general-purpose register file. Figure 1.9 shows this style of organization in a TPU-like design centered on a matrix multiplication unit implemented as a systolic array.

This organization provides very high arithmetic intensity and excellent energy efficiency for dense GEMM, convolution lowered to matrix multiplication, and related tensor operations. The drawback is reduced flexibility. Systolic arrays are most efficient when the computation is dense, regular, and well matched to the array dimensions. Control-flow divergence, sparse operands, irregular indexing, reductions, normalization, softmax, and data-layout transformations often require additional programmable support outside the array. Therefore, tensor accelerators are extremely effective for dense matrix kernels,

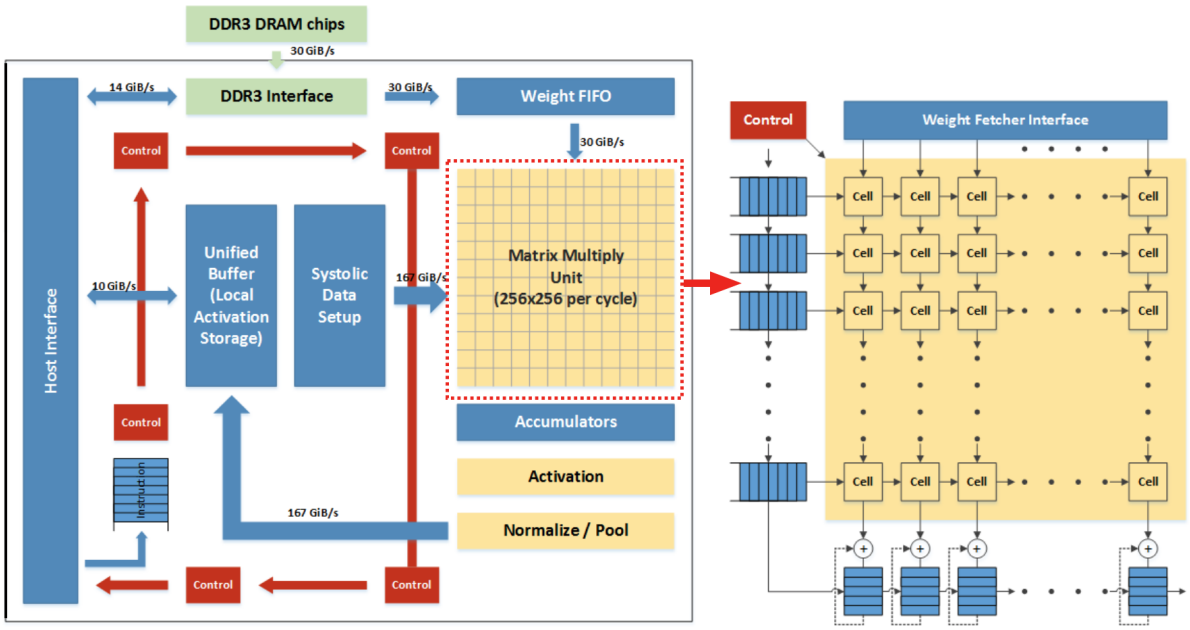


Figure 1.9: Example tensor/systolic accelerator organization centered on a matrix multiplication unit implemented as a systolic array. Taken from [100].

but they do not eliminate the need for more flexible execution engines in real systems.

Commercially, this approach is exemplified by Google's TPU family in the cloud (from early TPU [62] generations to recent TPU [61] pods used for large-scale training and inference), and by dedicated edge inference devices such as Coral-class accelerators [121] that target quantized workloads in constrained environments.

Coarse-grained reconfigurable architectures. CGRAs provide spatial acceleration through an array of programmable functional units connected by a configurable interconnect. Rather than executing a traditional instruction stream at runtime, a compiler maps a kernel's dataflow graph onto the array, configuring both the computation and the routing between units. Figure 1.10 illustrates this style of programmable spatial organization.

Because the configured hardware directly reflects the structure of the kernel, CGRAs can approach ASIC-like efficiency for stable computation patterns while retaining more flexibility than fixed-function accelerators. They are particularly attractive for loop-oriented kernels, software-managed local memory, predication, sparse analytics, and some irregular graph workloads. Their main challenges are compiler complexity, mapping overhead, and the difficulty of efficiently supporting highly dynamic control flow or unpredictable memory behavior. CGRAs therefore occupy an intermediate point between rigid spatial accelerators and fully programmable processors.

Vector processors. Vector processors occupy a distinctive point in the accelerator design space by exposing data-level parallelism explicitly at the instruction-set level while preserving ISA-level programmability [76, 75]. Instead of issuing a separate scalar instruction for each data element, a vector processor allows a single vector instruction to describe an operation over an ordered set of elements. The hardware

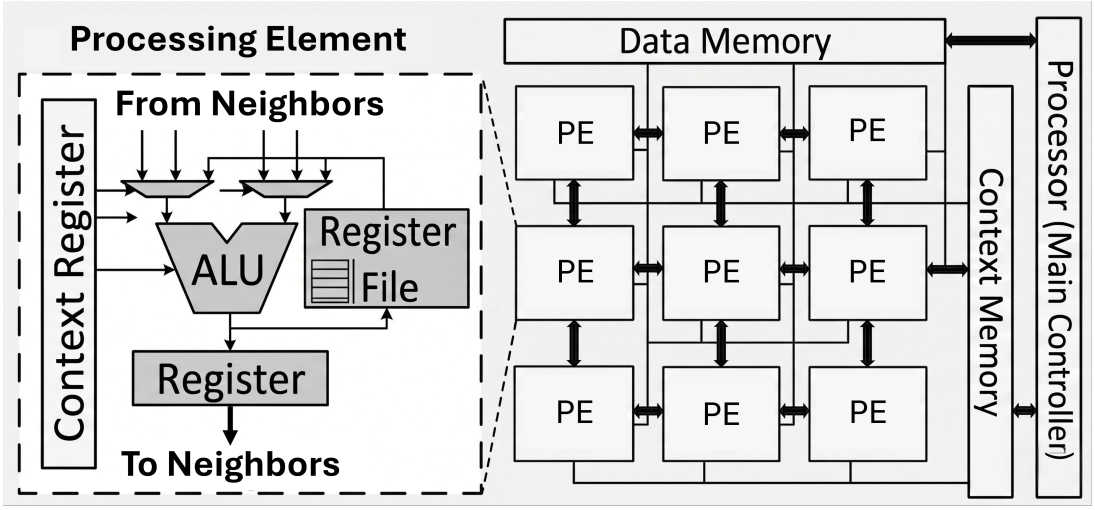


Figure 1.10: Representative coarse-grained reconfigurable architecture. Computation is mapped spatially onto programmable functional units connected through a configurable interconnect.

then executes that operation across a vector datapath composed of multiple lanes, amortizing control overhead across many arithmetic operations and improving both throughput and energy efficiency.

Figure 1.11 illustrates the basic principle of vector execution. The same vector instruction may be implemented with different degrees of hardware parallelism. In one implementation, vector elements are processed over multiple cycles, while in another, multiple lanes compute different vector elements in parallel. More generally, vector processors combine temporal and spatial parallelism: vectors are executed across multiple cycles, while several lanes may operate in parallel on different elements during the same cycle. This allows the same architectural vector abstraction to scale across implementations with different datapath widths.

Vector architectures can differ in how vector execution is attached to scalar control. In decoupled organizations [101, 106, 142, 104], vector execution is placed in a more separate engine attached to a scalar control processor. This allows the vector backend to be specialized for throughput, with more freedom in lane organization, register-file structure, and memory interfacing, but it also requires additional queues, control interfaces, register storage, and memory-coherence mechanisms. In contrast, an integrated vector unit [117, 130] is embedded more tightly within the main scalar processor pipeline. Rather than attaching a separate vector backend to a scalar control core, the integrated approach places vector execution resources alongside the scalar execution units and typically reuses more of the surrounding frontend, scheduling, register, and memory infrastructure. This tighter coupling reduces communication overhead between scalar and vector execution and can lower total area by avoiding the replication of control structures. For this reason, integrated vector units are especially attractive in systems where implementation cost, silicon area, and power consumption are tightly constrained.

A representative example of an integrated [130] organization is shown in Figure 1.12, where vector execution resources are incorporated into the broader out-of-order core organization. In such a design, vector instructions share parts of the fetch, decode, rename, and retirement machinery with scalar instructions, while the vector execution pipelines are connected more directly to the processor's existing register and cache hierarchy. Compared with a decoupled organization, this approach usually offers lower area

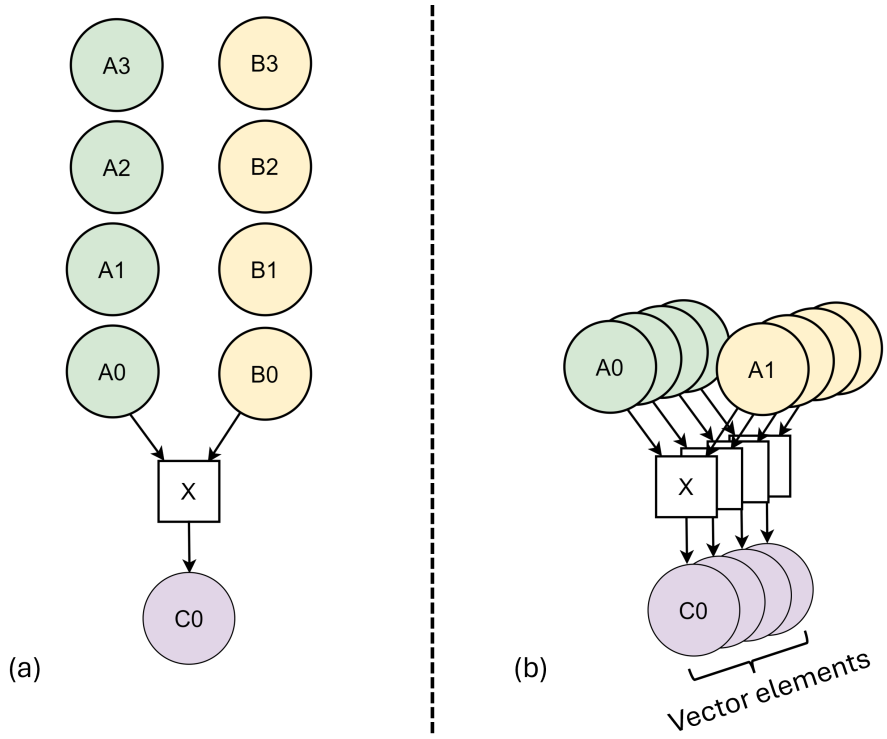


Figure 1.11: Execution of the same vector multiplication instruction in two different functional units. In (a), one vector element is processed per cycle, while in (b), multiple elements of the vector are computed in parallel.

overhead and simpler integration into a general-purpose core. By contrast, Figure 1.13 shows the organization of a representative decoupled-style vector processor, where a scalar core remains responsible for general-purpose control flow and issues vector instructions to a separate vector unit. These instructions are queued, decoded, and dispatched to the vector front-end, which manages vector issue, hazard tracking, the vector register file, and vector memory operations. Execution is performed by multiple vector lanes, each lane containing the resources needed to process a subset of vector elements. This organization preserves a close cooperation between scalar and vector execution: scalar code handles branches, orchestration, and irregular control, while vector instructions accelerate the regular, data-parallel portions of the workload.

The distinction between decoupled and integrated vector organizations therefore reflects a fundamental tradeoff. Decoupled vector engines allow a more specialized backend with higher vector throughput, greater freedom in lane organization, and more dedicated support structures, but they typically incur higher hardware complexity and area overhead. Integrated vector units, by contrast, achieve a more compact and cost-effective realization by reusing scalar-core structures, but they usually provide more modest vector throughput and less backend specialization. Both organizations preserve the same core principle of explicit data-level parallel execution, while occupying different points in the tradeoff space between performance, area, coupling, and implementation complexity.

A defining feature of modern vector architectures is that they separate the software-visible vector model from the exact physical width of the hardware. In vector-length-agnostic execution, software

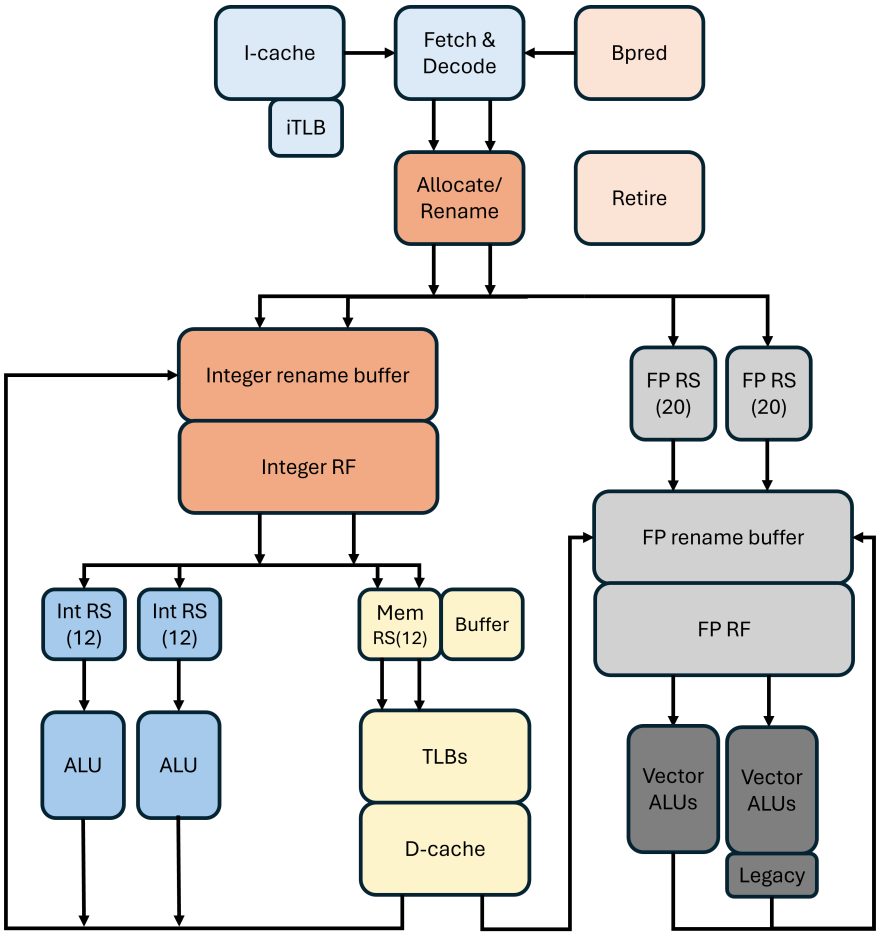


Figure 1.12: Knights Landing Core and VPU dataflow block diagram showing important microarchitectural structures and data bus connections between them. (ALU: arithmetic logic unit; iTLB: instruction translation look-aside buffer; RF: register file; RS: reservation station.)

specifies operations over vectors without being tied to a single fixed hardware width. The same program can therefore execute on implementations with different numbers of lanes or different maximum vector lengths, improving portability and allowing the architecture to scale across different performance and power targets.

Vector processors are also distinguished by their memory-system support. Many data-parallel workloads require more than simple contiguous loads and stores: they involve strided accesses, indexed accesses, masking, reductions, and irregular layouts. Vector architectures typically provide these capabilities directly through the ISA and the microarchitecture [37, 144], allowing them to sustain efficient execution across regular dense kernels as well as less regular patterns. This is particularly important for workloads such as machine learning, signal processing, sparse linear algebra, and data rearrangement, where performance depends not only on arithmetic throughput but also on the efficient movement and organization of data.

Because of this combination of explicit data-level parallelism, scalable execution width, and close cooperation with scalar control, vector processors offer an attractive balance between programmability and efficiency. They are flexible enough to support a broad range of kernels, yet structured enough to achieve high utilization on regular data-parallel workloads.

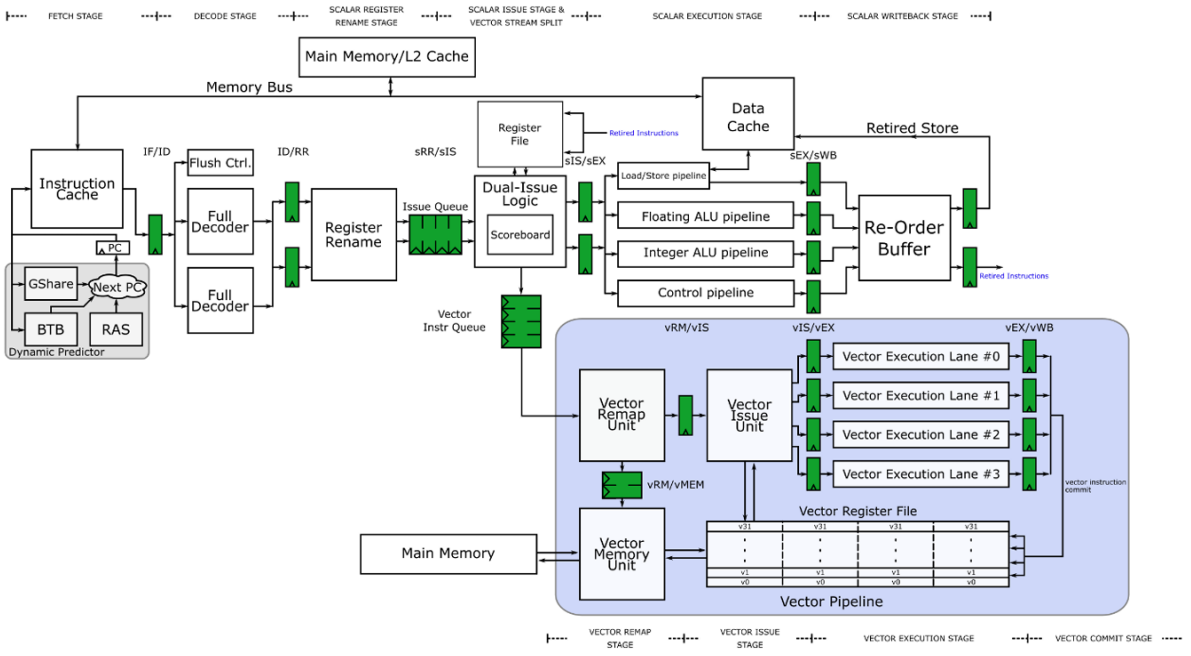


Figure 1.13: A representative decoupled vector-processor microarchitecture [101].

SIMD extensions. SIMD extensions can be viewed as a more tightly constrained form of vector-style execution integrated directly into a conventional scalar processor. Like vector processors, SIMD architectures apply one instruction to multiple data elements in parallel. However, this parallelism is expressed through fixed-width registers and fixed-width execution units that are directly tied to the hardware datapath. Typical examples include 128-bit, 256-bit, and 512-bit packed registers, where each instruction operates on a fixed number of elements determined by the architectural register width and operand precision.

This design makes SIMD attractive as an architectural extension to general-purpose CPUs. SIMD units reuse much of the surrounding scalar processor infrastructure and can accelerate common data-parallel kernels with relatively low additional complexity. They are especially effective when the working set is already close to the core, the data layout is regular, and low-latency execution is more important than absolute throughput. Common applications include multimedia processing, image and signal processing, cryptography, and short vectorized kernels inside general-purpose software[103].

The main limitation of SIMD is its fixed-width nature. Because the software-visible width is tied directly to the hardware width, code often needs to be recompiled, retuned, or rewritten to fully exploit wider generations of the same ISA family. Moreover, the fixed packing of elements can make execution less natural for workloads with variable vector lengths, irregular tails, or more complex memory-access patterns. Explicit shuffles, alignment handling, and data rearrangement can further reduce effective utilization.

Figure 1.14 highlights an important distinction between vector and SIMD execution through a side-by-side comparison of scalar C code, RISC-V Vector code [116], and AVX2 SIMD code [55] for integer addition. In the vector case, the arithmetic instruction form remains the same, while the element width is configured separately through the vector configuration mechanism. In the SIMD case, by contrast, the

Algorithm 1.1 Scalar C code

```
for i = 0 until i = n − 1 do
    C[i] = A[i] + B[i]
end for
```

Algorithm 1.2 RISC-V Vector code

```
loop:
vsetvli t0, a0, e32, m1
vle32.v v0, (a1)
vle32.v v1, (a2)
vadd.vv v2, v0, v1
vse32.v v2, (a3)
slli t1, t0, 2
add a1, a1, t1
add a2, a2, t1
add a3, a3, t1
sub a0, a0, t0
bnez a0, loop
```

Algorithm 1.3 AVX2 SIMD code

```
loop:
vmovdqu ymm0, [rdi]
vmovdqu ymm1, [rsi]
vpaddd ymm2, ymm0, ymm1
vmovdqu [rdx], ymm2
add rdi, 32
add rsi, 32
add rdx, 32
sub rcx, 8
jnz loop
```

Figure 1.14: Illustrative comparison of scalar C code, RISC-V Vector code, and AVX2 SIMD code for integer addition.

instruction is tied more directly to a specific packed operand width, so changing the datatype requires a different instruction mnemonic. For example, an AVX2 integer addition uses different instructions such as `vpaddb`, `vpaddw`, `vpaddd`, or `vpaddq` depending on whether the operands are 8-bit, 16-bit, 32-bit, or 64-bit integers. This comparison illustrates that SIMD exposes data parallelism through a fixed-width, datatype-specific software interface, whereas vector processors provide a more uniform and scalable execution model.

Conceptually, SIMD and vector processors share the same fundamental motivation: both reduce instruction overhead by applying one operation to many data elements. The main difference is that SIMD exposes this capability in a fixed-width form closely embedded in the scalar core, whereas vector processors provide a more scalable and explicit architectural model for data-level parallel execution. This makes SIMD highly effective for localized, regular acceleration inside general-purpose processors, but less flexible than full vector architectures as a standalone substrate for evolving data-parallel workloads.

1.3 The Role of Vector Processing in Heterogeneous Acceleration

While the accelerator design space offers multiple paradigms for exploiting data-level parallelism, each inherently trades flexibility for efficiency. SIMD extensions are easily integrated into scalar pipelines but suffer from rigid, fixed-width execution boundaries. SIMT architectures excel at massive throughput, yet their efficiency degrades sharply under divergent control flow or uncoalesced memory accesses. Systolic arrays and tensor engines push energy efficiency to its physical limits for dense linear algebra, but become highly inefficient when tasked with irregular memory accesses, sparsity, or layout transformations. Conversely, coarse-grained reconfigurable architectures (CGRAs) provide spatial flexibility, but rely on immense compiler complexity to achieve high hardware utilization.

Vector processors occupy a highly effective middle ground within this design space. By explicitly encoding data-level parallelism into the instruction set, a single vector instruction can dictate operations across an arbitrary number of data elements while remaining tightly coupled to the scalar control flow. This makes vector architectures significantly more flexible than fixed-function tensor engines, yet more expressive and scalable than traditional SIMD. Within a unified framework, vector processors natively support variable vector lengths, masked execution, complex permutations, mixed-precision arithmetic, and indexed (scatter/gather) memory operations.

This architectural flexibility is critical because modern workloads—particularly in machine learning and edge computing—are rarely composed entirely of dense matrix multiplications. The end-to-end execution of a Transformer model, for example, relies heavily on dense MatMul, but is fundamentally bottlenecked by non-linear and memory-bound operations such as softmax, layer normalization, masking, and attention-specific data permutations. Similarly, sparse and structured-sparse network topologies introduce indirect memory indexing and load imbalances that easily stall rigid spatial arrays.

The reality of these diverse, mixed-compute workloads strongly motivates a heterogeneous approach to acceleration. Rather than relying on a single, monolithic accelerator, modern SoC design increasingly pairs highly specialized compute blocks with programmable data-parallel engines. In this hybrid paradigm, matrix and tensor engines are deployed as heavy lifters for dense GEMM-like phases, while a flexible vector processor handles the critical ”glue” operations: reductions, elementwise computations, permutations, and sparse memory accesses.

Consequently, vector processors and matrix engines are highly complementary. As the computational burden of non-GEMM operations grows in modern neural networks, the efficiency of the programmable vector substrate becomes a primary determinant of overall system performance. Therefore, researching and optimizing vector architectures is a critical imperative. Whether deployed as standalone processors in highly constrained edge devices, or as the versatile, programmable half of a high-performance heterogeneous system, advancing the energy efficiency and microarchitectural capabilities of vector processors directly addresses the most pressing bottlenecks in modern computing.

1.4 Thesis Contributions

This thesis introduces a comprehensive suite of hardware and software optimizations designed to improve the efficiency of vector processors for machine-learning applications operating within strict power and silicon constraints. The work spans multiple levels of the design stack and is structured along two complementary directions: (micro)architectural innovations for resource efficiency, and algorithm-hardware co-design for optimized kernel execution. Taken together, these contributions aim to improve performance and energy efficiency while preserving the programmability and portability of modern vector architectures.

A fundamental architectural contribution of this thesis is the introduction of *register dispersion* [145, 147], a technique for reducing the physical footprint of the VRF without sacrificing vector length or ISA compatibility. Since the VRF is often the dominant area consumer in low-cost vector processors, simply providing a full set of physical vector registers can make the integration of vector support prohibitively expensive. Register dispersion addresses this issue by replacing the conventional monolithic VRF with a compact physical structure that behaves as a fully associative working-set storage for recently used (“hot”) vector registers, while colder registers are transparently dispersed into the existing memory hierarchy and fetched when needed. The thesis shows that many important workloads, including CNN kernels and vectorized FlashAttention, use only a small instantaneous working set of vector registers, making this approach highly effective in practice. Experimental results reported in the thesis show that a compact VRF with only eight 256-bit entries can closely match the performance of a conventional 32-entry design

while using only one quarter of its storage capacity. Moreover, reducing the stored registers from 32 to 8 yields a 3.5× reduction in VRF area, translates to a 53

Beyond register storage, the thesis addresses the execution bottlenecks associated with within-vector data motion through the design of a fully configurable vector microarchitecture and a *unified vector permutation unit* [146]. The baseline microarchitecture is parameterized across all major dimensions, including vector length, datapath organization, functional unit composition, and chaining mechanisms for larger LMUL configurations, enabling a systematic exploration of the energy-efficient design space. Within this flexible framework, the unified permutation unit efficiently executes RISC-V vector permutation instructions (such as `vrgather`, `vcompress`, and slide instructions) that occur frequently in real workloads and often introduce non-negligible overhead. Instead of relying on separate specialized datapaths, the proposed implementation maps both output-driven and input-driven permutation patterns onto a common crossbar-based structure. In this way, instructions with different semantic behaviors are supported by the same underlying hardware organization, reducing implementation complexity and turning complex data movement into a predictable, low-overhead execution primitive.

Shifting to algorithm-hardware co-design, the third contribution investigates how targeted vector ISA customization and optimized execution strategies can accelerate structured-sparse convolutional neural networks [149, 150]. This part of the thesis studies SpMM kernels that arise when convolutions are transformed into matrix operations. To overcome the irregular data accesses and index handling typical of SpMM, the thesis develops an optimized pure-RVV mapping of common N:M sparsity patterns (e.g., 1:2 and 2:4). By strategically storing non-zero values in vector registers while keeping sparse indices on the scalar side, the proposed row-based kernels reduce data dependencies and enable aggressive interleaved unrolling. Complementing this, a minimal fused-instruction extension, *IndexMAC*, is introduced to fuse an indirect VRF access with a multiply-accumulate operation. This allows tiles of the dense operand to remain resident in the VRF, heavily reducing repeated memory traffic. The resulting designs show that carefully combining data residency, instruction scheduling, and minimal ISA extensions can deliver large benefits without sacrificing the vector programming model.

The final contribution addresses the execution of Transformer models on resource-constrained platforms through the vectorization of FlashAttention-2 [148]. The thesis develops a fully vectorized implementation that remains strictly compliant with the standard RVV ISA and minimizes scalar intervention throughout the computation. All major stages of the kernel, including score computation, maximum tracking, exponential evaluation, accumulation, and normalization, are efficiently mapped to vector hardware using vector-length-agnostic tiling. A central element of this design is a low-cost, lightweight exponential approximation tailored to the softmax computation, which enables efficient vector execution directly using standard vector floating-point operations without requiring new custom instructions. By efficiently vectorizing the attention kernel and reducing its computational overhead, this contribution demonstrates that large-language-model workloads can be supported efficiently even on cost-sensitive vector machines.

Overall, the thesis demonstrates that energy-efficient vector processing is not the result of a single isolated optimization, but of coordinated design decisions across the architecture, microarchitecture, and kernel-mapping levels. By combining compact, demand-driven storage structures (Register Dispersion),

efficient support for data movement (Unified Permutation Unit), targeted ISA extensions for sparse linear algebra (IndexMAC), and vectorized attention mechanisms, the thesis advances the design of practical, generalizable, and highly efficient vector processors for contemporary machine-learning workloads.

1.5 Thesis organization

The remainder of this thesis progresses from the underlying vector architecture to workload-specific optimizations and advanced architectural techniques.

To provide a clear logical flow, Chapter 2 begins by establishing a fully configurable vector processor architecture, serving both as a practical tutorial on modern vector microarchitecture and as the hardware baseline for the entire work. It describes the core architectural parameters, issue mechanisms, and instruction sequencing required for variable-latency execution and large LMUL configurations. The chapter also presents the unified vector permutation unit, integrating it into the processor datapath to support both arithmetic operations and efficient internal data movement. It concludes with the memory subsystem and physical implementation.

Chapter 3 focuses on the efficient execution of structured-sparse convolutional neural networks. It develops RVV-based sparse-dense matrix multiplication kernels using optimized non-zero operand placement and loop unrolling. The chapter also presents and evaluates IndexMAC, a minimal fused-instruction extension that reduces memory traffic and improves dense-operand reuse.

Chapter 4 addresses Transformer workloads through a fully vectorized, RVV-compliant implementation of FlashAttention-2. It describes vector-length-agnostic tiling that reduces scalar control overhead and presents a low-cost exponential approximation for the softmax operation. The results demonstrate that resource-constrained vector processors can efficiently execute attention without dedicated tensor hardware.

Chapter 5 addresses the high cost of vector state by introducing Register Dispersion. The proposed organization replaces the monolithic vector register file with a demand-driven cache, transparently moving cold registers to the memory hierarchy while preserving ISA compatibility. The chapter evaluates its impact on area, performance, and power across representative machine-learning workloads.

Finally, Chapter 6 summarizes the main findings and contributions of the thesis and discusses future research directions for improving the efficiency and adaptability of vector processors.

Chapter 2

Configurable Vector Microarchitecture

Designing an energy-efficient vector processor requires navigating a complex multidimensional space of hardware trade-offs. Rather than presenting a single rigid implementation, it is essential to first unpack the fundamental microarchitectural knobs that dictate system performance, silicon area, and power consumption. By dissecting the anatomy of a decoupled RISC-V vector processor, the underlying mechanics of vector execution become clear, exposing the critical interactions between instruction sequencing, datapath organization, operation chaining, and memory subsystem organization.

Exploring these configurable parameters serves as a practical tutorial on modern vector architectures, illustrating how tuning vector lengths and functional unit compositions directly impacts computational efficiency. Ultimately, defining this highly parameterized landscape establishes the robust hardware baseline required to systematically evaluate the enhancements proposed later in this work.

Within this flexible design space, the architecture also provides a framework for addressing specific microarchitectural bottlenecks that arise in real-world applications. One of the most prominent challenges in vectorized kernels is the frequent need to reorganize data within the vector register space using gathers, compressions, and slide operations. Traditional designs often handle these permutations through disjointed, specialized datapaths that inflate hardware complexity and disrupt execution flow. To resolve this without compromising the configurable nature of the baseline, a novel unified vector permutation unit is introduced. By mapping the full spectrum of RISC-V permutation semantics onto a single streamlined datapath, this contribution demonstrates how complex internal data movement can be handled efficiently, turning a historical source of overhead into a predictable and low-cost execution primitive.

2.1 RISC-V Vector Instruction Set Extension

The microarchitecture presented in this chapter implements RISC-V Vector Extension (RVV) [116]. RVV provides a standard instruction-set interface for expressing data-level parallelism while remaining compatible with a scalar RISC-V processor. In contrast to a scalar instruction, which describes one operation on one set of scalar operands, a vector instruction describes an operation over a sequence of elements. More specifically, a vector instruction specifies the source and destination vector operands, the operation to be applied, and the active vector length over which the operation is performed. This allows a compact instruction stream to represent many element-wise operations and enables the hardware to

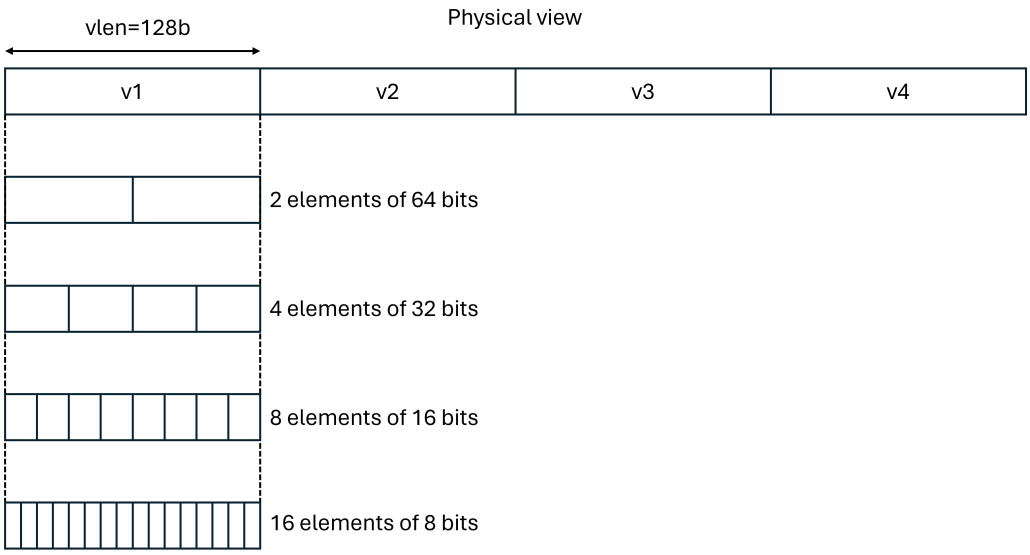


Figure 2.1: Interpretation of a vector register for different selected element widths. For a fixed vector register length of VLEN=128 bits, the same architectural vector register can hold two 64-bit elements, four 32-bit elements, eight 16-bit elements, or sixteen 8-bit elements.

amortize instruction and control overhead across multiple data elements.

RVV extends the scalar RISC-V ISA with architectural vector state. In particular, it defines a set of 32 architectural vector registers and 7 control/status registers that configure how vector instructions interpret and process data. The most important parameters for the microarchitecture described in this chapter are the vector register length (VLEN), the active vector length (`vl`), the selected element width (SEW), and the vector length multiplier (LMUL). These parameters define the architectural view of the vector register file and determine how much work is represented by each vector instruction.

The parameter VLEN denotes the number of bits in each architectural vector register. It is a property of the implementation and defines the maximum storage capacity of a single vector register. The selected element width, SEW, defines the width of each element currently stored and operated on by vector instructions. Therefore, the same vector register can contain a different number of elements depending on the active SEW. For example, if VLEN=128 bits, then a vector register can be interpreted as two 64-bit elements, four 32-bit elements, eight 16-bit elements, or sixteen 8-bit elements. This behavior is illustrated in Figure 2.1. The RISC-V Vector ISA defines several standard element widths, including 8, 16, 32, and 64 bits, allowing the same architectural register file to support different data types and precisions. In the proposed implementation, the supported SEW values are 8, 16, and 32 bits, matching the targeted integer, fixed-point, and single-precision floating-point workloads.

The active vector length, `vl`, specifies the number of elements that participate in the current vector operation. Unlike VLEN, which is fixed by the implementation, `vl` is configured dynamically by software through the RVV vector-configuration instructions. This distinction is central to vector-length-agnostic programming. A program does not need to know the exact physical width of the vector unit; instead, it repeatedly requests a legal `vl` for the remaining loop iterations and advances through the data accordingly. As a result, the same vectorized code can execute correctly on implementations with different vector-register lengths or different numbers of execution lanes.

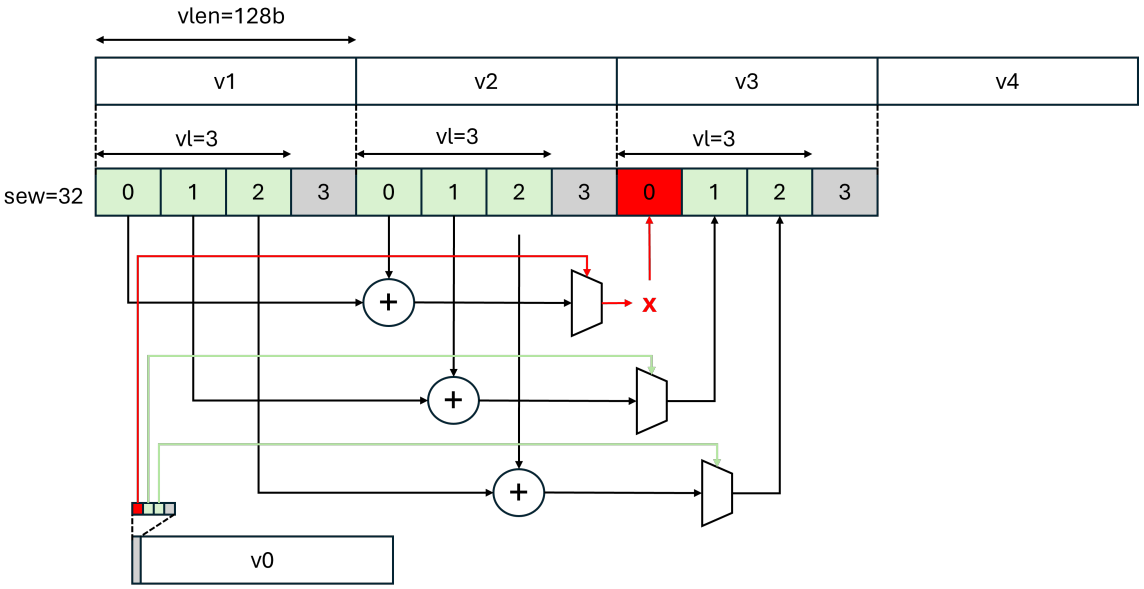


Figure 2.2: Example of active vector length and masking in RVV. With SEW=32 and v1=3, only the first three elements of each vector register participate in the operation. In addition, masked execution can selectively disable individual active elements using the mask register.

The difference between VLEN and v1 is important. VLEN defines how many bits exist in the architectural vector register, while v1 defines how many elements are active for a particular instruction. For example, with VLEN=128 bits and SEW=32 bits, each vector register can hold four 32-bit elements. If v1=3, only the first three elements are active in the vector operation, while the fourth element is outside the active vector length. This is useful for handling loop tails without requiring separate scalar cleanup code.

RVV also supports the vector length multiplier, LMUL, which is part of the vector type configured by the vector-configuration instructions. The LMUL parameter allows multiple architectural vector registers to be grouped together and treated as a larger logical vector operand. For example, when LMUL=1, each vector operand corresponds to one architectural vector register. When LMUL=2, two consecutive vector registers form one logical vector operand; when LMUL=4 or LMUL=8, four or eight registers are grouped, respectively. This mechanism increases the number of elements represented by a single vector instruction without changing the architectural instruction format. However, larger LMUL values also increase pressure on the vector register file, the issue logic, the dependency-tracking mechanism, and the writeback path.

The vector configuration is established by dedicated RVV configuration instructions, such as vsetvli, vsetivli, and vsetvl. These instructions set the active vector length vl and configure the vector type state, which includes the selected element width SEW and the vector length multiplier LMUL. In a typical vectorized loop, software executes a vector-configuration instruction before a sequence of vector operations. The configuration instruction chooses a legal vl for the remaining number of loop iterations and updates the vector type according to the required element width and register grouping. The following vector instructions then execute using this configuration until a new vector-configuration instruction changes it.

For example, a vectorized loop operating on 32-bit elements may execute a vsetvli instruction to re-

quest SEW=32 and LMUL=1. The hardware then sets vl to the maximum number of 32-bit elements that can be processed by the implementation without exceeding the remaining loop count. The subsequent vector arithmetic and memory instructions use this configuration until the next vector-configuration instruction is executed.

Another important feature of RVV is predicated or masked execution. Vector ISAs can express a large amount of data parallelism compactly, but many real loops include conditional behavior. Since vector instruction streams do not naturally introduce separate control flow for each element, conditional execution is commonly expressed through masks. In RVV, the vector register v0 is used as the architectural mask register. Each mask bit corresponds to one vector element and determines whether the corresponding element is active for a masked instruction. If a mask bit is enabled, the element is updated normally; if the mask bit is disabled, the element is not updated according to the selected masking policy. This enables vectorization of loops with conditional updates and also helps handle irregular or partially active data.

Figure 2.2 illustrates this behavior. The example uses SEW=32 and vl=3, so each vector instruction operates on the first three 32-bit elements of the source registers. The mask register then further controls which of these active elements are allowed to update the destination. In the example, one active element is disabled by the mask, so the corresponding destination element is preserved rather than overwritten by the newly computed result. This behavior has an important hardware implication: masking is not only an issue-time condition, but also a writeback-time commitment rule. The writeback logic must merge the newly computed result with the previous destination value on an element-by-element basis.

The memory instructions supported by RVV also influence the backend organization. Vector memory operations are not limited to contiguous accesses. The ISA supports unit-stride accesses, strided accesses, and indexed accesses (or also known as scatter/gather memory operations [144]). Unit-stride memory operations access consecutive memory locations and are the most regular and efficient case. Strided memory operations access elements separated by a constant stride, which is useful for structured layouts such as columns, interleaved data, or regularly spaced fields. Indexed memory operations use per-element offsets stored in a vector register and correspond to gather/scatter-style accesses. These instructions are important for sparse data structures and irregular memory layouts, but they are also more demanding for the memory subsystem because each element may refer to a different address.

Taken together, these ISA mechanisms determine the main requirements of the proposed microarchitecture. The backend must support different element widths, variable active vector lengths, register grouping through LMUL, predicated execution, intra-vector data movement, and multiple memory-access patterns. These requirements influence the organization of the register file, issue logic, execution datapaths, writeback stage, and memory subsystem.

2.2 Organization of the Vector Core

Vector processors can generally be organized according to two different backend styles [164], depending on the target vector length and the desired balance between scalability, complexity, and cost.

In *long-vector* designs [77, 20, 104, 93], the backend is typically organized into multiple homoge-

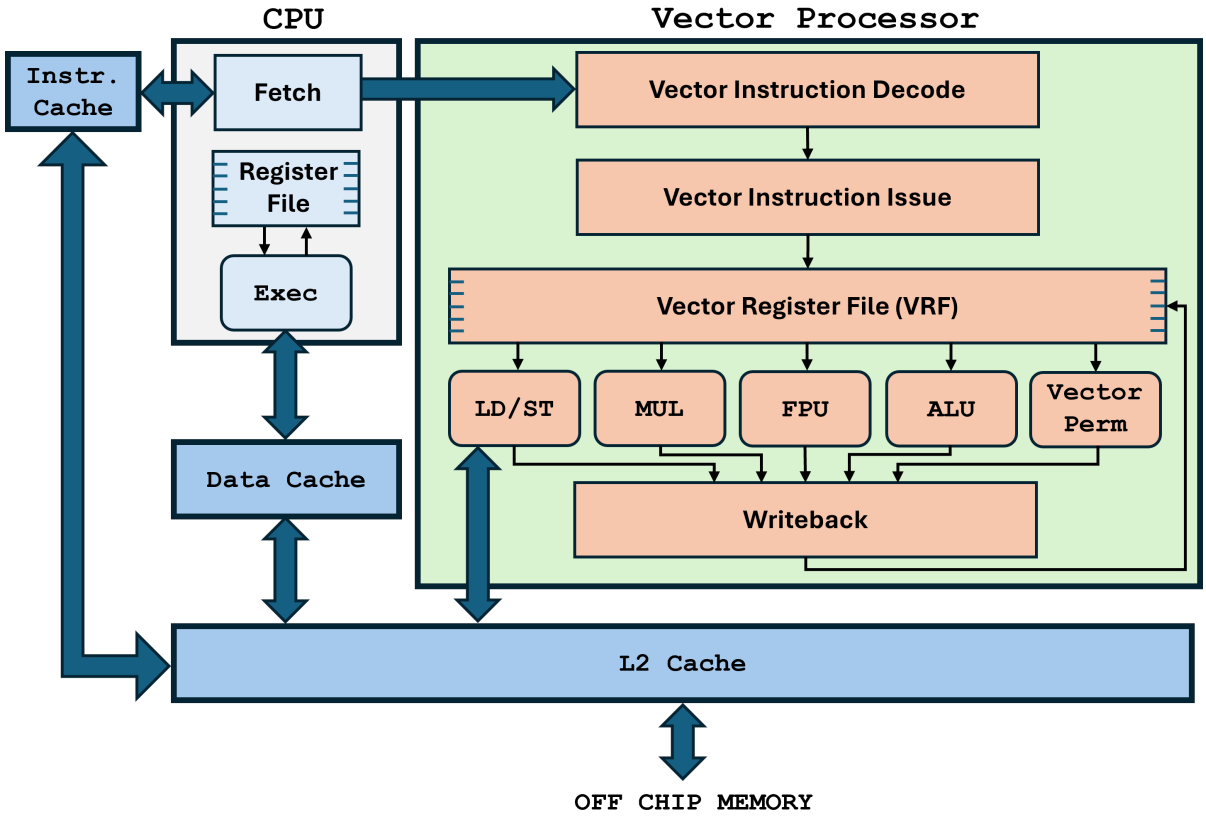


Figure 2.3: High-level organization of the implemented superscalar CPU and the attached decoupled vector processor used as the basis for the configurable microarchitecture of this chapter.

neous vector lanes. Each lane contains a local portion of the vector register file and a replicated set of execution units, and overall vector throughput is scaled by increasing the number of lanes. This organization is well suited to large vector widths, since it distributes both storage and computation across the lanes and allows the design to scale in a modular manner. Modern long-vector implementations typically rely on lane-distributed vector register files and are optimized for very long machine vectors and high datapath utilization.

In *short-vector* designs [142, 164], by contrast, the backend is more commonly organized around *distinct execution paths* rather than fully replicated homogeneous lanes. In this case, the vector register file is not primarily viewed as a collection of lane-local partitions driving identical execution units, but rather as a shared backend structure that feeds specialized datapaths dedicated to different classes of operations. Such an organization is often more appropriate for cost-sensitive and embedded systems, where the target vector width is smaller and where reducing hardware complexity, storage overhead, and datapath duplication is a primary design goal. Short-vector implementations are commonly characterized by a unified multi-ported register file and a compact backend datapath, and therefore require careful scheduling and high utilization of the available execution resources.

The vector architecture presented in this thesis follows the short-vector approach introduced above. More specifically, the implemented backend is a decoupled vector core connected to a 6-stage, 2-wide superscalar RV32IMF host core, as shown in Figure 2.3. The scalar core handles general-purpose con-

trol flow and dispatches vector instructions to the separate vector backend. The backend is organized around distinct execution paths rather than a fully replicated lane-based organization, which makes it more suitable for the area- and cost-constrained design point targeted in this thesis.

Figure 2.3 illustrates the organization of the proposed vector backend. Vector instructions are decoded in the vector front end and then dispatched to the appropriate backend execution path. At the center of the backend lies the vector register file (VRF), which stores vector operands and feeds the functional units. Unlike a lane-partitioned organization, the VRF in this design acts as a shared backend storage structure serving multiple specialized datapaths.

As shown in the figure, the backend consists of several distinct execution paths. The MUL path is used for integer multiplication-oriented operations, the ALU path supports arithmetic and logical operations, and the FPU path implements vector floating-point execution through a custom-designed floating-point unit. In addition, a dedicated permutation path supports the data-rearrangement instructions of RVV, while a separate load/store unit handles memory accesses between the vector core and the memory system. This separation of functionality across backend paths allows each class of vector operations to be handled by a dedicated datapath, reducing complexity and enabling a more efficient organization for short-vector execution.

A key property of the proposed vector backend is its modular organization. The design is structured so that execution units may be added or removed with limited modifications to the surrounding control logic. More specifically, the issue stage and the writeback stage rely on generic backend interfaces, allowing a new execution unit to be integrated with minimal effort. In practice, this mainly requires extending the issue-side dispatch and dependency handling for the corresponding operation type, while at writeback the arbiter simply includes the completion request of the new unit as an additional input. This modularity is particularly useful for a configurable vector processor, as it enables exploration of different backend compositions without extensive redesign of the core.

The arithmetic execution units are custom-designed to reuse the same hardware resources across different SEW configurations. More specifically, the design supports SEW values of 8, 16, and 32 bits, enabling the execution of multiple data types on the same datapath. This reuse is particularly important for multiplication and addition, where the same underlying hardware is adapted to different operand widths instead of being replicated for each supported data type.

The permutation unit is shown as a separate execution path because permutation operations often require data movement patterns that differ significantly from conventional arithmetic execution. Such operations may involve element reordering, compaction, or movement across positions that are not naturally handled by arithmetic datapaths. Treating permutation support as an independent backend component also avoids the need for multiple specialized permutation datapaths or the sequential handling of complex operations, both of which would increase implementation cost or execution latency.

To facilitate the discussion of the following sections, Table 2.1 summarizes the main architectural and microarchitectural features of the proposed vector processor.

Table 2.1: Summary of the proposed configurable vector microarchitecture.

Parameter / Feature	Description / Options
Vector datapath width	32/64/128/256/512/1024
Memory datapath width	32/64/128/256/512/1024
Supported element widths (SEW)	8/16/32
Supported LMUL configurations	1/2/4/8
Issue policy	In-order instruction issue
Hazard management	Scoreboard-like pending-register table
Writeback policy	Centralized arbitration of completed results
Mask handling	Element-wise masked result commitment during writeback
Memory support	Unit-stride/strided/indexed memory operations

2.3 Issue Logic and Hazard Management

The vector backend of the proposed processor contains multiple execution paths, including arithmetic, multiplication, floating-point, permutation, and memory units. Although this organization improves efficiency by specializing each datapath, it also introduces a control problem at issue time. In particular, different vector operations may exhibit different execution latencies, and therefore the writeback of a destination vector register does not necessarily occur at a fixed distance from decode. As a result, a newly decoded instruction cannot be issued solely based on structural availability; it must also be verified that its source operands are already produced and that its destination does not conflict with an earlier in-flight instruction.

A further architectural choice concerns the instruction-issue policy of the vector unit. In general, vector processors may employ either out-of-order or in-order issue. Out-of-order execution can improve utilization by allowing later instructions to proceed when earlier ones are stalled, but it requires significantly more complex control logic, including larger scheduling structures, more extensive dependency tracking, and mechanisms to monitor and retire operations that complete out of order. These features increase both area and energy consumption, which conflicts with the design goals of low-cost, energy-efficient vector architectures. For this reason, the proposed microarchitecture adopts an in-order issue policy with lightweight hazard tracking, preserving predictable execution and low control overhead while still enabling efficient support for the targeted workloads.

For this reason, the Vector Instruction Issue block incorporates a simple hazard-management mechanism based on a scoreboard-like pending-register table. The purpose of this mechanism is to preserve correct execution in the presence of variable-latency vector operations while keeping the control logic compact and suitable for a cost-sensitive short-vector processor. Instead of performing aggressive out-of-order scheduling or register renaming, the proposed design follows a conservative in-order issue policy: an instruction is allowed to advance only when all of its operand dependencies have been resolved.

The hazard-management unit tracks destination registers of issued vector instructions through a one-bit pending flag per architectural vector register. When an instruction is accepted for issue and it writes a destination register, the corresponding bit is set. The bit remains asserted until the execution backend signals writeback completion for that destination register, at which point the bit is cleared. In this way,

the issue stage maintains a compact view of which architectural registers are still awaiting results from earlier instructions.

Before issuing a new vector instruction, the control logic checks whether any of its referenced operands correspond to registers whose pending bit is set. More specifically, the mechanism examines the validity of the source operands `vs1`, `vs2`, and, where applicable, the destination operand used as an input (e.g., for destructive updates or masked/accumulating forms). In addition, the vector mask register is treated in the same manner, so a masked instruction is stalled if the mask register is still awaiting writeback. If any of these checks indicate that an operand is pending, the instruction remains at the issue stage and a stall is inserted. Otherwise, the instruction is allowed to proceed to the appropriate execution path.

This mechanism effectively implements a simple scoreboard for read-after-write dependencies and also prevents unsafe reuse of a destination register before the completion of an earlier writer. The design is intentionally minimal: it does not attempt dynamic wakeup or speculative bypass across multiple queued instructions, but instead relies on stalling at issue until the required register becomes available. This choice is consistent with the overall philosophy of the proposed microarchitecture, where compact control structures are preferred over more aggressive scheduling machinery.

An important detail is that the writeback notification is also considered during dependency checking in the same cycle. Thus, if a register is marked as pending but is simultaneously being written back, the waiting instruction may be released without incurring an extra stall cycle. This small optimization preserves the simplicity of the scoreboard approach while avoiding unnecessary bubbles at issue.

It should be noted, however, that the proposed hazard-management mechanism is intentionally limited in scope. The design does not perform dynamic instruction reordering, speculative issue, or register renaming, nor does it include a wakeup–select structure for searching across multiple buffered instructions in order to find a ready one. Instead, dependencies are resolved conservatively by stalling issue until the required operands become available and the destination register is no longer marked as pending. This restriction is deliberate and consistent with the target design point of the proposed microarchitecture, which prioritizes low hardware cost, reduced control complexity, and predictable timing behavior over the additional performance potential of a more aggressive dynamically scheduled backend.

Overall, the proposed hazard-management unit provides a lightweight and effective solution for coordinating issue in a multi-path vector backend. By maintaining per-register pending state and stalling dependent instructions until writeback completion, it ensures correct operand usage under variable execution latency without introducing the complexity of a fully dynamic scheduler.

2.4 Supporting Larger LMUL Configurations

As described in Section 2.1, `LMUL` allows a logical vector operand to span multiple architectural vector registers. From a microarchitectural perspective, however, LMUL introduces an additional level of complexity. When grouped registers are used, a single architectural vector instruction may require multiple physical register groups to be read, processed, and written back in the correct order. This affects backend scheduling, operand access, and result commit, especially in a short-vector processor where the execution resources are intentionally limited and shared across multiple classes of operations. The mi-

croarchitecture must instead preserve the architectural semantics of the grouped operand while mapping the operation onto the available datapath width.

At a high level, two approaches can be used to support such operations. The first is *instruction cracking* [164], in which a vector instruction is decomposed early in the pipeline into multiple simpler micro-operations, each operating on one portion of the logical register group. This approach simplifies some aspects of backend execution, since the issue logic only needs to handle narrower operations. However, it requires additional decode-side machinery to generate, track, and schedule the resulting micro-operations, and therefore increases control complexity.

The second approach is *instruction sequencing*. Under this organization, the original instruction is preserved as a single architectural operation, while the control path issues the corresponding physical register groups over multiple backend steps. In other words, wide logical execution is realized as an ordered sequence of narrower physical executions. This avoids the need for a dedicated cracking stage and is therefore more appropriate for the cost-sensitive vector microarchitecture targeted in this thesis. For this reason, the proposed design adopts instruction sequencing as the mechanism used to support larger LMUL configurations.

In the proposed implementation, each vector instruction is augmented during decode with a compact representation of the number of physical registers that participate in the logical operand group. Since RVV supports LMUL values up to 8, this information can be encoded using an 8-bit thermometer mask. The mask directly reflects the number of grouped registers associated with the instruction. For example, LMUL = 1 is encoded as `0000 0001`, LMUL = 2 as `0000 0011`, LMUL = 4 as `0000 1111`, and LMUL = 8 as `1111 1111`. This representation provides a simple way to track the progress of sequencing, since the number of remaining active bits corresponds to the number of physical execution steps that still need to be completed.

Table 2.2: Thermometer encoding of LMUL grouping.

LMUL setting	**8-bit thermometer mask**
LMUL = 1	`0000 0001`
LMUL = 2	`0000 0011`
LMUL = 4	`0000 1111`
LMUL = 8	`1111 1111`

The sequencing mechanism operates by updating this mask after each completed backend step of the instruction. In the simplest organization, the sequencing rate is one, meaning that one physical register group is processed per step. Under this configuration, an instruction with LMUL = 4 requires four sequencing steps, while an instruction with LMUL = 8 requires eight steps. More capable implementations may support higher sequencing rates, so that multiple physical register groups are processed in each step when sufficient datapath bandwidth and operand access capacity are available. In this way, the same control mechanism can be used across a family of implementations with different cost and performance targets.

The hardware organization of the sequencing control unit is illustrated in Figure 2.4. The central idea is that the thermometer mask associated with the current instruction is shifted whenever a sequencing step completes. The amount by which the mask is shifted is determined by the active sequencing rate. If

one physical group is processed per step, the mask is shifted by one bit; if two groups are processed, it is shifted by two bits, and so on. A comparator checks whether the shifted mask has become zero. Once this occurs, the instruction has completed all required sequencing steps and the control path may allow the next architectural vector instruction to proceed.

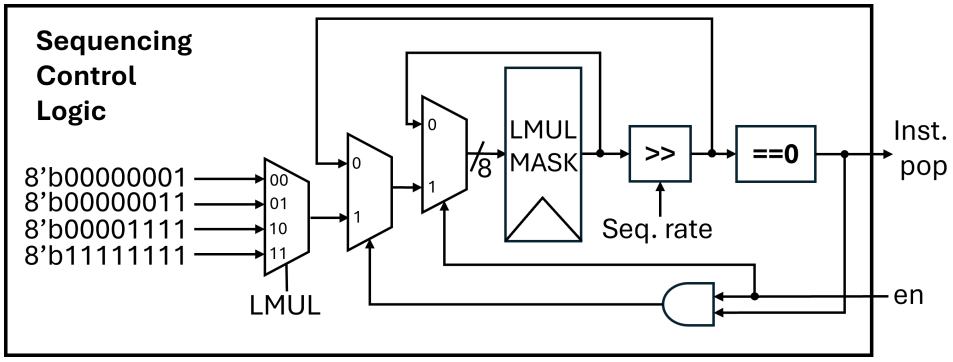


Figure 2.4: Control logic for LMUL instruction sequencing.

This organization provides a compact and scalable method for supporting grouped vector registers. Rather than introducing a separate decoding mechanism that expands each wide instruction into multiple internal operations, the backend retains the original instruction and tracks its progress through a small sequencing controller. This fits naturally with the short-vector design philosophy adopted in this chapter, where the backend is built around limited and reusable execution resources instead of aggressively replicated datapaths.

Algorithm 2.1 Vector add with LMUL = 1

```
1: vsetvli t0, a0, e32, m1
2: vle32.v v8, (a1)
3: vle32.v v4, (a2)
4: vadd.vv v12, v8, v4
```

Algorithm 2.2 Vector add with LMUL = 4

```
1: vsetvli t0, a0, e32, m4
2: vle32.v v8, (a1)
3: vle32.v v4, (a2)
4: vadd.vv v12, v8, v4
```

Figure 2.5: Illustrative RVV instructions with different LMUL configurations. The value of LMUL is set by `vsetvli`, which determines how many architectural vector registers are grouped together for each logical vector operand.

The effect of the mechanism can be illustrated with the two RISC-V Vector configurations shown in Figure 2.5. The value of LMUL is set by the vector-configuration instruction `vsetvli`, which determines how many architectural vector registers are grouped together for each logical operand. With LMUL = 1, the instruction `vadd.vv v12, v8, v4` operates on single architectural vector registers, so the backend treats `v4`, `v8`, and `v12` as individual source and destination operands. With LMUL = 4, however, the same instruction refers to grouped operands: the first source corresponds to `v4`–`v7`, the second source corresponds to `v8`–`v11`, and the destination corresponds to `v12`–`v15`. When the sequencing rate is one, this LMUL = 4 instruction is executed over four backend steps. In the first step, the processor operates on `v4`, `v8`, and `v12`. In the second step, it operates on `v5`, `v9`, and `v13`. The third step processes `v6`, `v10`, and `v14`, and the fourth step processes `v7`, `v11`, and `v15`. Collectively, these four narrower executions realize the semantics of the original LMUL = 4 vector instruction.

An important advantage of this approach is that it generalizes beyond LMUL itself. The same sequencing principle can also be applied whenever the logical vector width exposed to software exceeds the amount of data that the backend datapaths consume in a single cycle. As a result, support for larger LMUL configurations is not treated as an isolated special case, but as part of a broader microarchitectural strategy for executing wide vector operations efficiently on vector processors.

2.5 Execution Datapaths

Achieving high efficiency in a vector processor requires not only an appropriate backend organization, but also carefully designed execution units. Since vector workloads combine arithmetic intensity, mixed data types, and frequent data movement, the quality of the execution datapaths strongly affects both performance and hardware cost. In this thesis, particular attention was given to the design of specialized execution units that support the requirements of RVV-based machine-learning workloads while remaining compatible with a cost-sensitive short-vector organization. In this context, we redesigned key execution datapaths, including the floating-point unit, the multiplier, and the permutation unit, with the aim of improving flexibility, reuse of hardware resources, and overall execution efficiency.

2.5.1 Floating-Point Execution Unit

The custom floating-point unit is organized as the staged datapath shown in Figure 2.6. Throughout the figure, `exp` refers to the exponent component of the floating-point representation, whereas `sig` refers to the significand, commonly referred to as the mantissa. The figure illustrates the flow of operands through the floating-point execution path, starting from operand preprocessing, followed by parallel preparation of the multiplication and addend operands, accumulation, normalization, rounding, and final result packing. The same datapath organization is used to support multiple floating-point precisions, including FP32, FP16, and FP8. Through this shared design, the unit supports the core floating-point arithmetic operations required by the vector backend, including addition, subtraction, multiplication, and fused multiply-add variants.

Execution begins with an operand preprocessing stage. In this stage, the input floating-point operands are decoded from their packed architectural format into an internal representation. The sign, exponent, and significand fields are extracted, the implicit leading significand bit is restored when required, and special cases such as zero, infinity, and NaN are identified. This preprocessing step allows the later arithmetic stages to operate on separated internal fields rather than directly on the packed floating-point encoding.

After preprocessing, the datapath separates into two parallel computation paths. One path prepares the product term by multiplying the two primary operands and computing the corresponding sign, exponent, and significand information. In parallel, the other path prepares the addend operand by widening and extending it into the internal format used by the accumulation stage. This parallel structure is useful for fused multiply-add operations because the multiplication result and the addend can be prepared concurrently before being combined.

The outputs of the two parallel paths are then merged in the accumulation stage. This stage compares

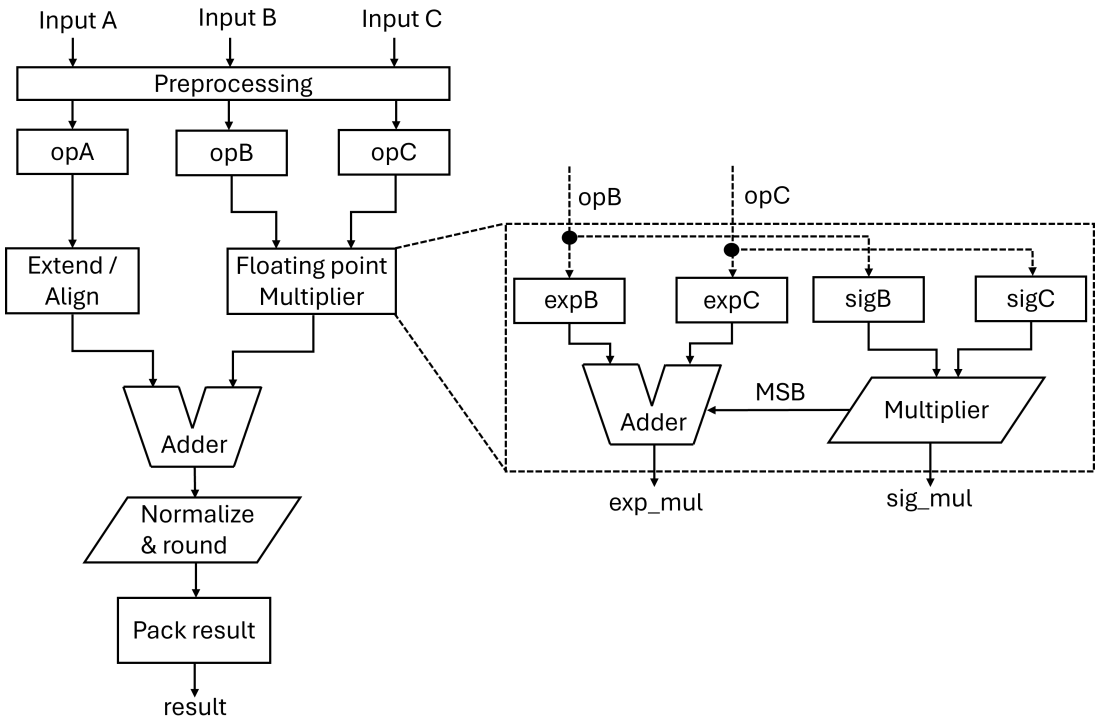


Figure 2.6: Dataflow of the custom floating-point execution unit. The datapath first preprocesses and unpacks the input operands, then prepares the product and addend operands in parallel. These intermediate values are combined by the accumulation stage, normalized and rounded, and finally packed back into the architectural floating-point format.

the exponents of the incoming operands, aligns the significands by shifting the smaller operand, and performs the required addition or subtraction depending on the selected instruction and operand signs. For standalone addition and subtraction operations, this stage performs the main arithmetic operation directly. For fused multiply-add variants, it combines the product term with the prepared addend term.

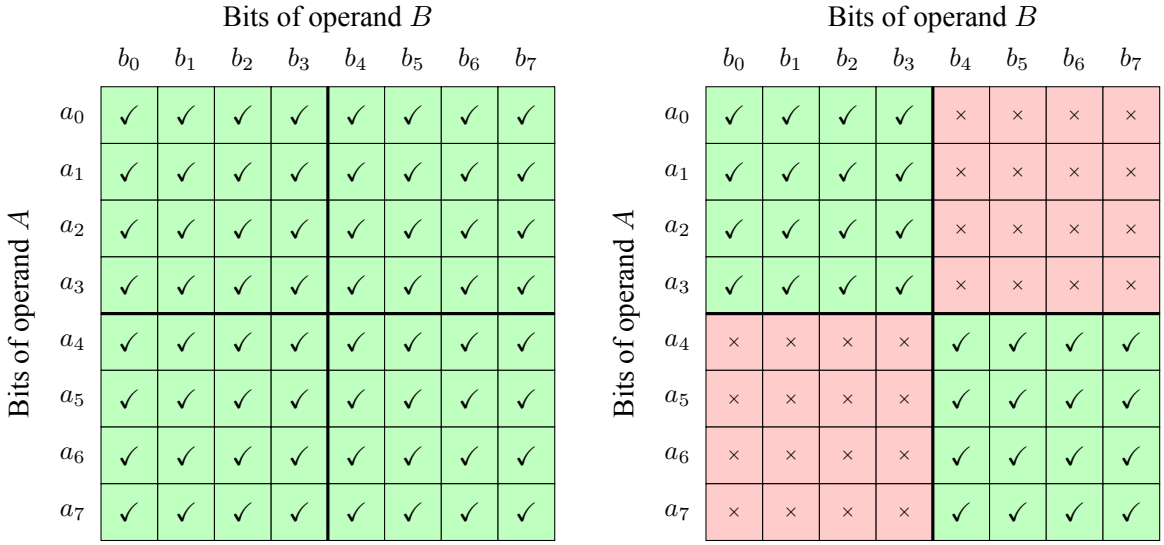
The intermediate result is then passed to the normalization and rounding stage. This stage adjusts the significand so that it is in normalized floating-point form, updates the exponent to account for any normalization shift, and applies the selected rounding mode. The rounded result is then forwarded to the final packing stage.

The final stage converts the internal floating-point result back into the architectural output format. It packs the final sign, exponent, and fraction fields and produces the value that is written back to the vector register file.

This staged organization offers two important advantages. First, it decomposes the floating-point computation into simpler functional blocks, which facilitates efficient hardware design and integration into the vector backend. Second, it allows the datapath to be tailored to the proposed short-vector backend rather than relying on a fixed standalone floating-point unit. As a result, the same compact execution path can support a broad class of RVV floating-point vector instructions, including addition, subtraction, multiplication, and fused multiply-add variants.

2.5.2 Multiplier Unit

Multiplication is one of the most important arithmetic operations in vector workloads and therefore constitutes a key execution datapath of the proposed vector backend. To support efficient execution across different SEW configurations, the multiplier unit was custom-designed as a configurable SIMD multiplier. The same hardware structure is reused across multiple operand widths, avoiding the need to instantiate separate multipliers for each supported element size.



(a) Mask for SEW configuration of 8

(b) Mask for SEW configuration of 4

Figure 2.7: Bit-level masking behavior of the configurable 8-bit SIMD multiplier for different selected element width (SEW) configurations: (a) for SEW=8, all partial products are preserved since the operands are treated as single 8-bit elements; (b) for SEW=4, only intra-element partial products are preserved, while cross-products between the two packed 4-bit subwords are suppressed.

The multiplier begins by generating the full bit-level partial-product matrix between the two input operands. This matrix spans the entire datapath width of the multiplier and is therefore formed across all bits of the input operands. To enable packed SIMD operation, it is then filtered through a configurable mask-generation mechanism driven by the active SEW configuration. The role of this masking logic is to preserve only the partial products that belong to the same packed element and to eliminate cross-products between bits that originate from different subword elements. This behavior is illustrated in Figure 2.7, which presents a simplified 8-bit example for explanatory purposes. The use of SEW=4 in Figure 2.7(b) is intended only to demonstrate the masking principle more clearly; it is not one of the supported SEW configurations of the proposed design. In particular, Figure 2.7(a) shows the case in which all partial products are propagated because the operands are treated as single 8-bit elements, whereas Figure 2.7(b) illustrates how the masking logic preserves only intra-element partial products and suppresses cross-products between packed subwords. In this way, the same multiplier structure can emulate multiple narrow element-wise multiplications or fewer wide multiplications using a single physical datapath.

Once the valid partial products have been selected, they are reduced through a dedicated compressor structure. In the current implementation, this reduction is realized by a tree composed of half adders and

full adders, which compresses the partial-product matrix into two intermediate rows. These rows are then added to obtain the full multiplication result.

The final stage of the multiplier performs result extraction. Depending on the requested operation, either the lower or the upper half of each element-wise product is selected and forwarded as the architectural result. This makes it possible to support both standard and high-part multiplication semantics using the same multiplier core, with the result-selection logic adapting automatically to the active element width.

Through this organization, the multiplier unit achieves efficient hardware reuse across different operand widths while maintaining a compact and configurable implementation. This makes it well suited to the short-vector backend of the proposed processor, where execution efficiency must be balanced carefully against area and implementation complexity.

2.5.3 Permutation Unit

In a short-vector processor, not all important operations are purely arithmetic. Many vectorized kernels also require frequent rearrangement of data within the vector register file, so efficient support for data movement becomes an important part of the backend design. To enable data rearrangement within vector registers, there is a variety of permutation instructions. These instructions facilitate arbitrary merging, grouping, and other complex data shuffling or data copying. This capability is crucial for optimizing performance in diverse kernels [50], such as matrix multiplication (for sparse or dense data), matrix transpositions [152], and cryptographic primitives [78]. These in-vector register permutations complement indexed and strided loads, which can rearrange data within vector registers during memory fetching [115].

Each class of vector permutation instructions uses different control information to specify how vector data elements should be rearranged in the destination register. For example, `vcompress` uses a binary mask to select which elements of the source vector register should be gathered at one end of the destination register while preserving their original relative order. Here, the control information describing the permutation directly relates to the input data elements. Conversely, in `vrgather`, the control information refers to the destination register elements; each destination element specifies which input element should be placed in its position.

The differentiation in semantics of control information for vector permutation instructions, both per-input and per-output, complicates their execution within a single datapath. This challenge is further intensified by the requirement for a fixed and data-independent execution latency, to protect against side-channel attacks in crypto accelerators. These challenges are addressed by designing a unified hardware architecture capable of executing all RVV vector permutation instructions with the same latency, regardless of their control information structure, thereby reducing area and hardware cost.

The focus is on the design of an efficient RVV-compliant hardware vector permutation unit for short-vector length machines (with vector lengths up to 256 bits) that operate on a unified multi-ported vector register file driving SIMD capable execution units. This design aims for single-cycle or pipelined execution of vector permutation instructions with a fixed latency of one or two cycles. To accommodate operations on longer vectors using register grouping, vector permutations can be executed as a series of single-vector permutation operations [115].

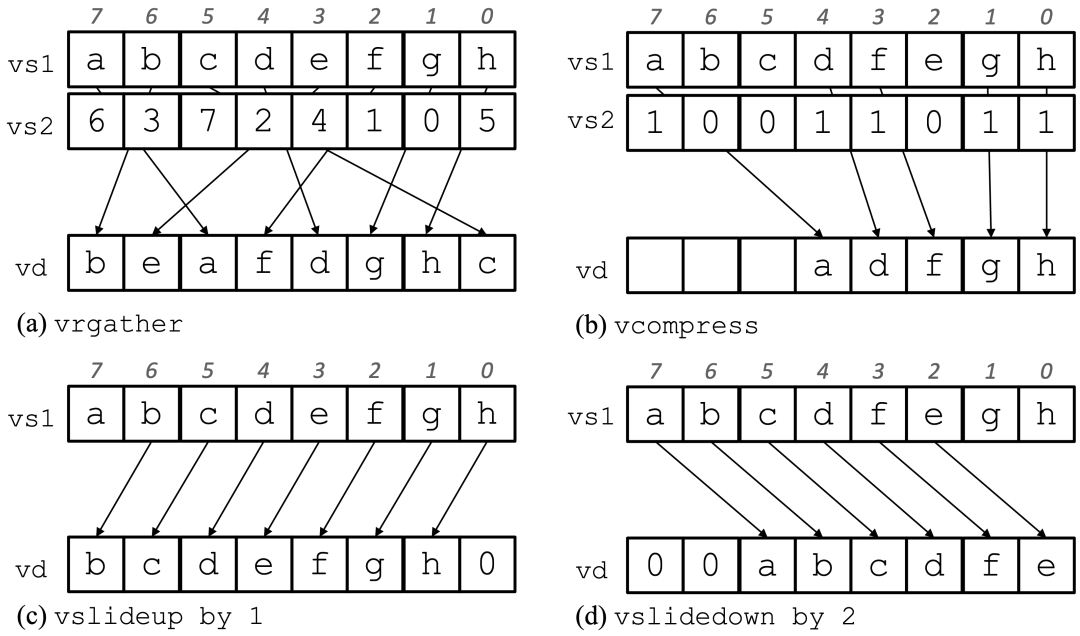


Figure 2.8: An example demonstrating the implementation of (a) `vrgather` and (b) `vcompress`, assuming a vector length of 8.

Permutation in RISC-V Vector Processors

A vector permutation involves the rearrangement of the elements within a vector register. RISC-V RVV permutation instructions are split in three categories.

The `vrgather` instruction offers the most versatile form of vector-element rearrangement. In this operation, data movement is explicitly determined by per-output indices. The `vrgather` instruction takes two operands: the first is the input vector, and the second is a vector of indices that dictate which input element is selected for each element of the output vector. An example of the execution of `vrgather` is shown in Figure 2.8(a).

The `vcompress` operation, on the other hand, gathers elements of the input vector into the starting positions of the output vector, based on a mask. The mask identifies which elements in the input vector should be included on the right side of the output vector, without losing their original relative position. An example of the execution of `vcompress` is shown in Figure 2.8(b).

The `vslide` instructions are also input-driven permutations and provide two main variants, regarding the direction of the slide: `vslidedown` and `vslideup`. These operations shift the elements of the input vector in the corresponding direction (`vslidedown` shifts to the right and `vslideup` shifts to the left), based on an offset value. Examples of the execution of `vslideup` and `vslidedown` are shown in Figs 2.8(c) and 2.8(d), respectively.

All-three permutation instructions can be masked using register `v0`, thus allowing the targeted data re-arrangement to affect only a subset of the output elements.

The implementation of vector permutation instructions has taken many forms in recent RISC-V vector processors.

Ara [20] supports the `vslideup` and `vslidedown` instructions, while more complex permuta-

tion operations, such as vector gathering and compressing, are implicitly implemented through indexed loads. This approach minimizes the cost of vector permutation hardware units, but incurs latency proportional to the vector register size. Other processors adopting this strategy include ZeroVex [166], Vecim [160], LEM [35], and newer versions of Ara [105, 104].

Processors like Virtuvius [94], big.VLITTLE [142], and Spatz [19] employ an inter-lane interconnect to enable communication-intensive operations (e.g., permutations, reductions). Ring networks are often preferred for their simplicity, allowing lanes to exchange data via the network. AraXL [108] also uses a ring network to connect to neighbor vector clusters to perform large vector permutations.

To execute a `vrgather` instruction in a network requires the transformation of the per-output source indices of the `vs2` register into per-input destination indices that can be handled by the network's routing. For this reason, two ring networks were employed in [153]: a data network and an index network. The index network converts source indices to destinations by routing element index `i` to position `vs2[i]`, enabling subsequent data network transfers. It should be noted that the use of networks for permutations makes permutation execution latency variable and data dependent.

Contrary to previous approaches, Vicuna [106], Saturn [164], and Ocelot – a vector engine for the BOOM processor [165] – permute one element per cycle (similar to loading an element per cycle in an indexed load). LEM [35] follows a similar strategy, but it also uses additional index registers to manage memory accesses and permutations.

The design of a unified datapath for vector permutation instructions is complicated by the fundamental difference in control semantics between *input-driven* and *output-driven* approaches.

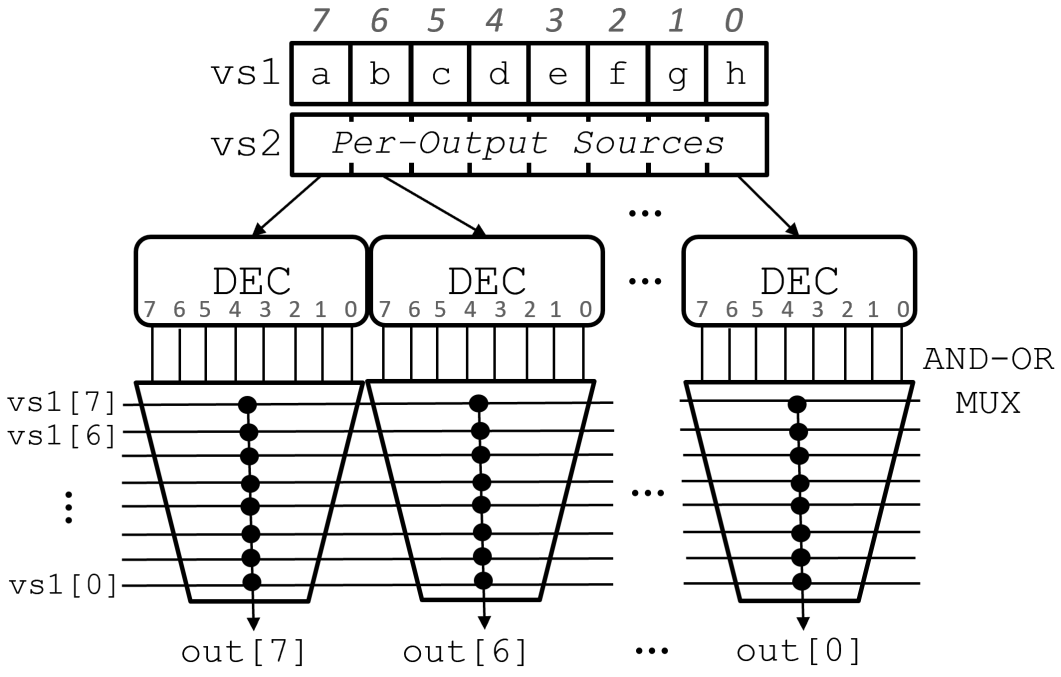


Figure 2.9: The crossbar logic consisting of an AND-OR multiplexer per output element that executes `vrgather` for an 8-element vector. The per-output source addresses are first decoded (to one-hot) before driving the output multiplexers.

Since the RISC-V RVV ISA extension allows for re-configurable vector element width, the presented permutation unit is designed to support the smallest possible element width (1 byte), and wider widths are treated as series of consecutive bytes. In this way, all possible vector element widths are fully and seamlessly supported.

The proposed architecture for a unified vector permutation unit relies on a crossbar with per-output AND-OR multiplexers, as shown in Figure 2.9. This structure is innately amenable to output-driven permutation instructions, such as `vrgather`. The select signals of each output of the crossbar are simply the one-hot equivalent vectors of the value in the corresponding element in the vs2 register of the instruction. Note that, as per the definition of `vrgather`, this setup allows for the same input element to be copied to multiple output elements.

This datapath can handle all *output-driven* permutation instructions. The next step is to devise a mechanism to facilitate the execution of all *input-driven* permutations – e.g., `vcompress`, `vslide`, etc. – on the same datapath with minimal overhead/complexity.

The control bits in the second source operand (`vs2`) of input-driven instructions include mask bits that refer to the corresponding elements in the first source operand (`vs1`), and these bits cannot be directly related to a specific output element. Thus, those bits must be converted into suitable information that can leverage the existing *per-output* crossbar structure.

To achieve this, we follow a two-step approach. First, the mask bits in `vs2` are transformed into a per-input, output-element destination. This means that, for each input element of `vs1`, we determine the corresponding output element to which it should move. Second, to *reuse the crossbar that executes* `vrgather`, we must rearrange the destination indices into a format compatible with the crossbar logic.

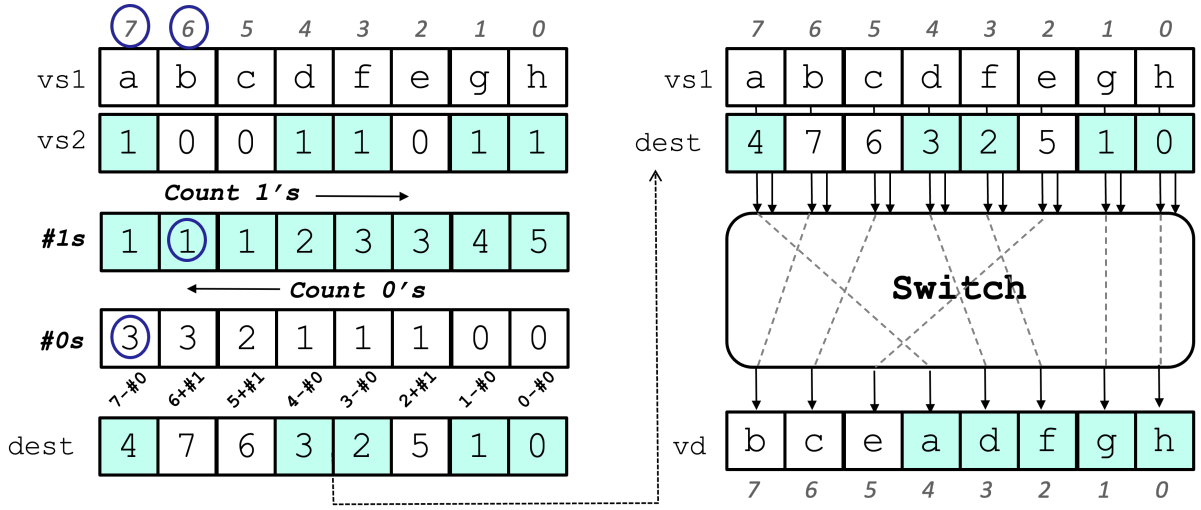


Figure 2.10: The transformation of the mask control bits of `vcompress` results in per-input destination indices that indicate which output element the corresponding input should move to.

The transformation algorithm comprises two operations: the calculation of two vectors of prefix sums and the addition/subtraction of these sums to/from the corresponding vector element indexes (positions). The whole transformation flow is illustrated in Figure 2.10. The first vector of prefix sums is generated from left to right (high to low vector indexes) and counts the number of 1s in the bit mask contained in the second operand (`vs2`) of the instruction. The second vector of prefix sums is generated in the opposite

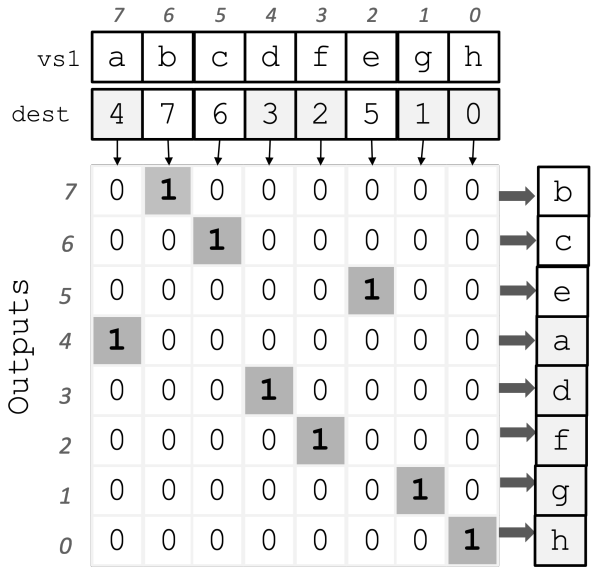


Figure 2.11: The decoded (one-hot) form of the per-input destinations can drive the outputs of a crossbar to move input elemenets according to the functionality of `vcompress`.

direction, i.e., from low to high vector indexes, and counts the number of 0s in vector `vs2`. The last step of the algorithm generates the desired vector of per-input destinations. The value of each element of this vector is calculated as follows: if the corresponding mask bit in `vs2` is 1, then the corresponding prefix sum-of-0s is *subtracted* to its vector index. On the other hand, if the corresponding mask bit in `vs2` is 0, then the corresponding prefix sum-of-1s is *added* from its vector index.

Hence, the final result of this pre-processing algorithm is a vector indicating the destination element (in the output vector register) of each input element in `vs1`. The right part of Figure 2.10 shows how this vector of output destinations can be used to implement the `vcompress` instruction. Although it may seem unnecessary to move the mask-0 elements to the left, this is done to ensure that no two elements in the vector of destinations have the same value. The importance of this attribute will be explained shortly.

The elements of this vector can now be used directly as select signals for the same crossbar used by output-driven permutation instructions. Figure 2.11 visualizes how this can be achieved by using the matrix abstraction for the crossbar. As can be seen, each element in the newly generated vector of destinations is converted into a one-hot *vertical* vector.

Same as with output-driven instructions, each *row* still corresponds to the select signals of the corresponding output of the crossbar. Each row of this matrix is guaranteed to also be one-hot (a pre-requisite for functional correctness) by the definition of the input-driven instructions themselves: no two (or more) input elements may go to the same destination element. By construction, the vector of destinations generated by the pre-processing algorithm includes different destination values in each of its elements, as previously mentioned.

Hardware implementation

The hardware implementation of the pre-processing algorithm described above is shown in Figure 2.12 for an 8-element datapath. The two prefix sums are computed in parallel using carry-save counters [59] at

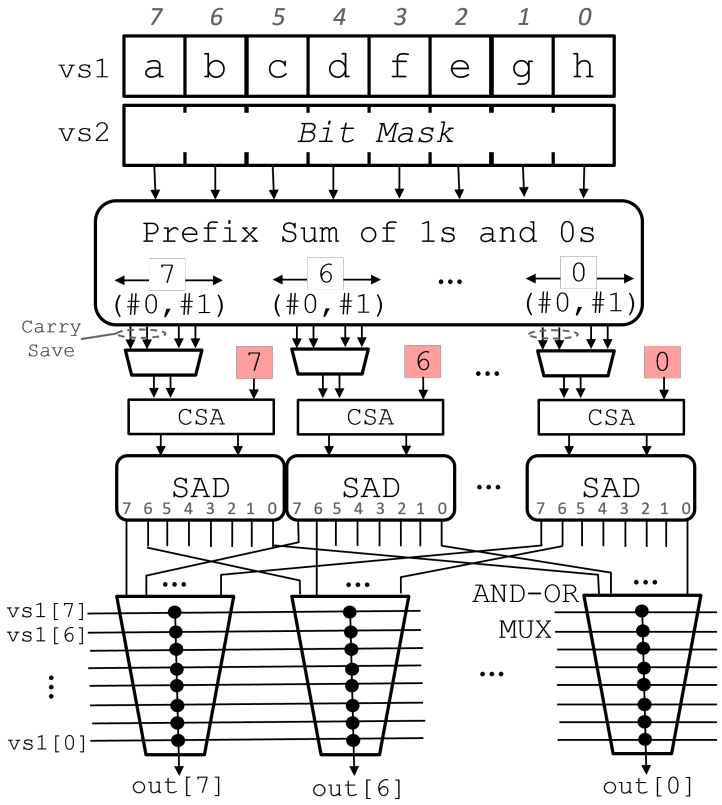


Figure 2.12: The crossbar-based logic that executes the `vcompress` instructions. Sum-Addressed Decoders (SAD) implement fuse addition and decoding without the need for carry propagation.

each element position. To eliminate unnecessary delays caused by carry propagation within the counting logic, the prefix sums are maintained in carry-save form. Figure 2.13 illustrates the carry-save counting logic used to count the number of ones from position 7 down to position 0 while keeping the result in carry-save form. Similar cells are used for the remaining positions and also for counting the number of zeros.

Returning to Figure 2.12, the prefix sum at each position must be added to – or subtracted from – the index of the corresponding position, and the resulting value should be directly decoded. Since the prefix sum at each position is in carry-save form, it is added to the position index using a Carry-Save Adder (CSA). Since position indexes are fixed values, the CSA logic is simplified after logic synthesis.

The output of the CSA specifies the destination of the corresponding element in a redundant carry-save format. To determine the destination in a non-redundant format, a carry-propagate adder is typically used to sum two outputs of the CSA. Instead of following this approach, the CSA result is passed directly to a Sum-Addressed Decoder (SAD) [87, 45]. The SAD computes the final sum while simultaneously converting it into a one-hot vector representation. This add-and-decode operation inherently eliminates the need for internal carry propagation [45]. As a result, the final decoded destinations are produced with minimal delay, as carry propagation is avoided at every step.

Each input element requires one such SAD unit to determine its corresponding one-hot destination vector. These vectors correspond to the one-hot vertical vectors shown in Figure 2.11. After appropriate wire reshuffling of the SADs' output, we derive the horizontal one-hot vectors of Figure 2.11, which

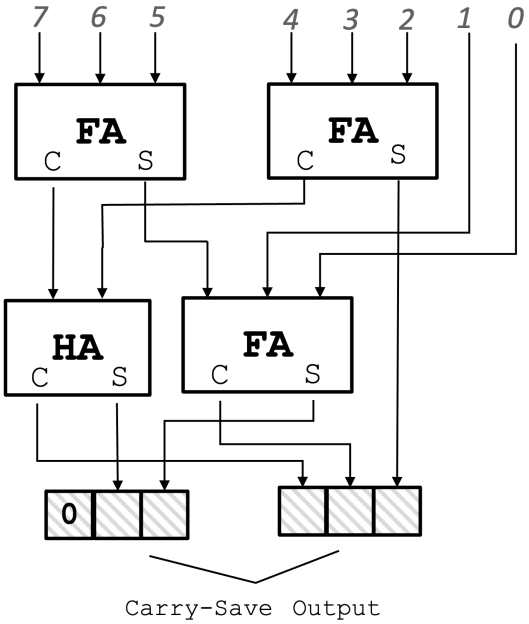


Figure 2.13: Counting the number of 1's from position 7 down to 0 and returning the sum in carry-save format. The redundant representation of the output eliminates the need for internal carry propagation.

serve as select signals for the crossbar's output AND-OR multiplexers.

Vector slides can share parts of the datapath presented for the `vcompress` instructions. Specifically, the slide operations of `vslideup` and `vslidedown` do not need the calculation of prefix sums. Instead, the slide offset (included within the instruction itself) is simply added (or subtracted in two's complement) to/from each input element index to calculate the destination in the output register. Hence, the indexes and slide amount are fed directly into the SAD, thereby bypassing the prefix sum logic. By construction, if a sum exceeds the vector-element width, then the SAD outputs an all-zeros vector, which correctly ensures that the affected input element is not sent to any output. Such out-of-bounds sums correspond to the input elements that "slide out" of the output vector register after the slide operation. The inclusion of the positive or negative slide offset using an additional multiplexer at the input of each SAD is highlighted in Figure 2.14.

Furthermore, since we reuse the same AND-OR multiplexers for `vrgather`, `vcompress`, and `vslide`, the *per-output* and *per-input* select-signal distribution interconnects are multiplexed before reaching the output multiplexers, depending on the class of the permutation instruction (output- or input-driven, respectively). These extra multiplexers are highlighted in gray at the lower part of Figure 2.14.

Implementation Cost of the Unified Permutation Unit

To assess the implementation cost of the proposed unified design, the permutation unit was compared against a baseline organization that uses separate mechanisms for permutation instructions. In the baseline version, `vrgather` is executed using the crossbar logic shown in Figure 2.9, `vslide` is supported by a dedicated byte-level logarithmic shifter, and `vcompress` is implemented using a sequential datapath that moves one element with asserted mask bit per cycle to its final position in the destination register,

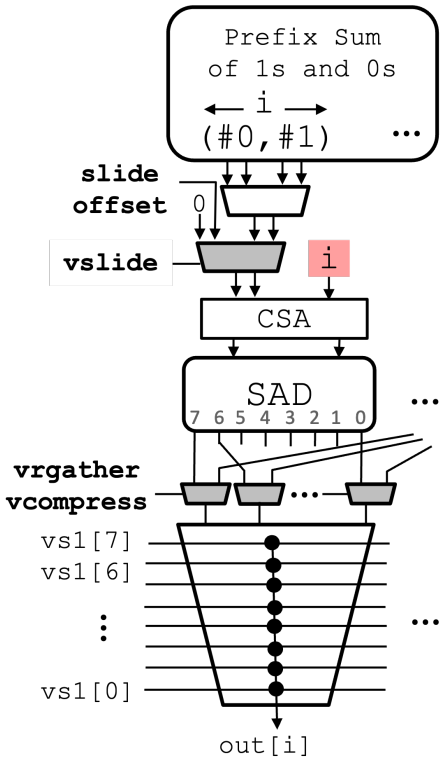


Figure 2.14: The bit slice of the permutation logic at the i^{th} output. A positive or negative offset is added to the index of the i^{th} bit position and directly decoded via the SAD. In the case of `vrgather`, the decoded signals directly drive the multiplexer of the corresponding output, while in the cases of `vcompress` and `vslide`, the decoded signals are reshuffled to different outputs.

similarly to [164]. As a result, `vcompress` is a multi-cycle operation in the baseline design. In contrast, the proposed permutation unit executes the full class of RVV permutation instructions within a unified datapath.

For a fair comparison, both versions were designed to support byte-granular permutations, with one byte as the smallest movable element. Both processor variants were synthesized through physical implementation using the OpenROAD flow [3], the ASAP 7 nm standard-cell library, and a target clock frequency of 500 MHz under identical physical design constraints. Table 2.3 reports the area of the complete vector processor after integrating the permutation unit, assuming a vector length of 256. The results show that the proposed permutation unit introduces only a modest area increase relative to the baseline organization. This overhead arises from the additional address-generation logic placed in front of the crossbar to support a wider range of permutation operations within a unified datapath. Even so, the added cost remains limited while enabling single-cycle execution of operations that would otherwise require separate hardware mechanisms or multiple cycles. Moreover, the structure is amenable to higher-frequency implementations, since the indexing logic and crossbar can be pipelined into separate stages without violating the requirement of data-independent execution latency.

To assess the impact of the minimum supported movable-element size, the same two designs were also examined with the supported permutation granularity relaxed from one byte to two bytes. This change significantly reduces the area of the permutation logic in both cases, showing that the granularity

Table 2.3: Area of Vector Processor with VL=256 bits at 7 nm

Size of Element	Baseline	Proposed
1 byte	96,630 um^2	98,124 um^2
2 bytes	93,537 um^2	93,708 um^2

of supported permutations is an important design parameter for controlling cost.

Power consumption was evaluated using representative vectorized kernels, each containing at least one vector permutation instruction [118]. The switching activity of all nets was quantified through post-layout gate-level logic simulation, and power was estimated in OpenSTA [3] using extracted wire parasitics. The average power trends are consistent with the area results. In both designs, `vrgather` and `vslide` exhibit very similar power behavior, since in these cases the carry-save prefix summation logic of the unified datapath is bypassed. By contrast, `vcompress` consumes more power in the unified design, which directly reflects its single-cycle implementation, unlike the sequential implementation used in the baseline processor. For applications that rely only on primitive one-position vector slides, it can be advantageous to execute them outside the main unified datapath. For the broader class of permutation operations, however, the unified permutation unit remains the more capable solution.

2.6 Writeback Arbitration and Result Commitment

As the execution units operate independently and may complete at different times, their results must be merged through a dedicated writeback stage before being committed to the vector register file. The role of the writeback stage is therefore twofold: first, to serialize concurrent completion requests from multiple execution units onto the shared register-file write port, and second, to finalize the value that will eventually be committed, including the handling of masked vector operations.

To support this behavior, the writeback stage receives a valid signal and a result payload from each execution unit. These requests are examined by an arbitration mechanism, which determines which unit is granted access to the shared writeback path during the current cycle. In the proposed implementation, arbitration is performed using a rotating-priority policy. A priority register is maintained and updated after each successful grant, so that the unit that was just served loses highest priority in the following cycle. This realizes a round-robin style arbitration scheme, avoiding the permanent preference of one execution path over the others and ensuring fair access when multiple units complete close together in time.

Once a grant is produced, the corresponding result payload is selected through a multiplexer and becomes the active writeback transaction. Only the granted execution unit observes a ready signal, indicating that its result has been accepted by the writeback stage, while all other units must keep their outputs stable until they are selected in a later cycle. In this way, the arbitration and handshake logic together guarantee that at most one result is forwarded to the vector register file per cycle, even if several execution units finish simultaneously.

Beyond arbitration, the writeback stage is also responsible for final result construction in the presence of masked vector operations. In masked execution, not all lanes of the computed result are necessarily

written back. For inactive mask positions, the architectural semantics require preservation of the previous destination element rather than overwriting it with the newly computed value. To support this behavior, the writeback unit combines the newly produced result with the old destination value on an element-by-element basis. The exact granularity of this merge depends on the configured element width, as determined by the current `vsew` setting. Thus, the writeback stage performs the final mask-aware commit of vector results before they are written to the destination register.

This organization is particularly important because it centralizes architectural result commitment. While the individual execution units are responsible for performing the actual computation, the writeback stage defines the point at which a result becomes globally visible to the rest of the vector backend. In other words, completion of execution alone is not sufficient; a result is considered architecturally committed only after it passes through the arbitration and masking logic and is accepted on the shared writeback interface. This distinction is important for the hazard-management mechanism described previously, since the pending status of a destination register should only be cleared when the corresponding writeback transaction is actually performed.

The writeback stage therefore acts as the convergence point of the vector backend. It resolves simultaneous completion events from heterogeneous execution units, applies the final masking semantics required by the vector ISA, and serializes all result updates toward the register file. By combining fair arbitration with mask-aware result selection, the proposed design provides a compact and robust solution for integrating multiple execution paths into a single coherent vector commit interface.

2.7 Memory Subsystem

In addition to the computational backend and its writeback path, the processor also requires a memory subsystem that supplies vector operands and handles memory-visible result transfer. This is particularly important in a short-vector design, where overall efficiency depends not only on the quality of the arithmetic units but also on the ability of the backend to move data to and from the vector register file with low overhead. As already shown in the high-level organization of the processor in Figure 2.3, memory access is handled through a dedicated load/store path that connects the vector core to the surrounding memory system and to the scalar processor.

From a microarchitectural point of view, memory operations differ from purely computational vector instructions because they interact with an external subsystem whose latency is not necessarily fixed. While arithmetic and permutation instructions are executed inside specialized backend paths fed directly by the vector register file, load and store instructions must additionally perform address generation, transfer data through the memory hierarchy, and return results to the architectural vector state through the writeback path. For this reason, the memory subsystem is a first-class component of the vector backend rather than a secondary implementation detail.

The role of the memory subsystem is therefore threefold. First, it must support RVV vector memory instructions and provide the bandwidth required by vectorized workloads. Second, it must integrate cleanly with the short-vector backend organization adopted in this chapter, where vector execution resources are compact and shared across multiple instruction classes. Third, it must preserve a consistent

view of memory between the scalar and vector domains, since both operate within the same core and may access shared data structures during the execution of a program.

2.7.1 Support for Vector Memory Operations

The proposed microarchitecture includes a dedicated load/store unit responsible for transferring data between the memory system and the vector register file. This unit complements the arithmetic, floating-point, multiplication, and permutation paths described earlier, and enables vector instructions to fetch operands from memory and commit results back to memory without overloading the computational datapaths. In this way, the organization of the backend remains modular, with each major instruction class mapped to an appropriate execution path.

Supporting vector memory instructions efficiently is essential because many important machine-learning and signal-processing kernels are limited not only by computation, but also by data movement. Even when the arithmetic datapaths are highly optimized, insufficient support for vector loads and stores can reduce effective throughput and leave execution resources underutilized. The load/store path must therefore sustain the data supply required by the vector backend while remaining compatible with the cost-sensitive design goals of the processor.

In this design, the load/store unit is treated as an independent backend path. Vector loads return their data through the shared writeback interface, while vector stores read their operands from the vector register file and issue memory requests through the memory path. This keeps memory execution modular and consistent with the rest of the backend organization.

2.7.2 Scalar–Vector Memory Ordering

An additional requirement arises from the fact that the scalar and vector domains share the same memory subsystem. In this context, the processor must preserve a consistent view of memory between the two execution domains, so that memory operations issued by one side observe the effects of the previously completed memory operations issued by the other. Here, the term coherence does not refer to a distributed cache-coherence protocol of the kind used in multicore systems, but it refers to preserving correct ordering and visibility between the scalar and vector pipelines of the same core when both access a common memory system.

To enforce this property, the proposed design adopts a simple ownership-based arbitration mechanism at the memory interface. At any given time, the memory subsystem is owned either by the scalar core or by the vector unit. Memory requests issued by the current owner are allowed to proceed normally, whereas requests from the other domain remain pending until ownership is transferred. Ownership transfer is permitted only when all outstanding memory operations of the current owner have completed.

This policy prevents reordering hazards between scalar and vector accesses. For example, if the scalar core performs a store to a memory location and the vector unit subsequently issues a load from the same address, ownership must not be transferred to the vector side until the scalar store has completed. Otherwise, the vector load could observe an older value. By delaying ownership transfer until the current owner has no in-flight memory operations, the design preserves correct memory visibility between the two execution domains.

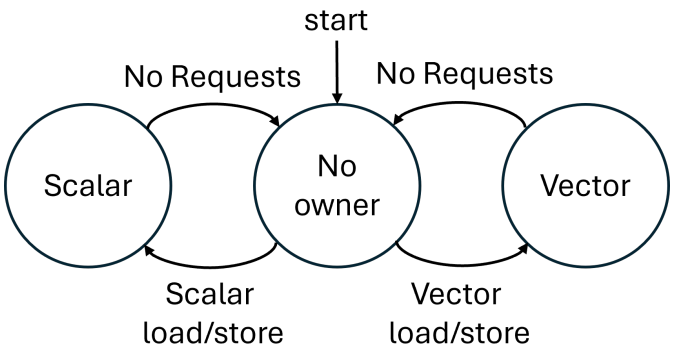


Figure 2.15: Ownership-based finite-state machine used to coordinate scalar and vector memory accesses.

The policy can be represented by a simple finite-state machine with three states: no owner, scalar owner, and vector owner, as shown in Figure 2.15. Initially, the memory subsystem is in the no-owner state. Once a scalar or vector memory instruction is issued, ownership is assigned to the corresponding domain. The subsystem remains in that state while memory operations from the current owner are still outstanding. When the number of in-flight requests of the owner reaches zero, the controller may either release ownership and return to the neutral state or transfer ownership to the other domain if pending requests exist there. In this way, the mechanism provides a deterministic means of coordinating accesses from the scalar and vector sides without requiring more elaborate dynamic scheduling.

2.7.3 Design Trade-offs

The main advantage of this organization is its simplicity. Because only one execution domain owns the memory subsystem at a time, the design avoids subtle ordering hazards between scalar and vector requests without introducing a complex coordination mechanism between the two pipelines. This is well aligned with the overall philosophy of the microarchitecture presented in this chapter, where compact and carefully chosen control structures are preferred over more aggressive but more expensive alternatives.

The main drawback is that the mechanism serializes access to the shared memory interface. If both scalar and vector domains generate memory traffic at the same time, one side must wait until the other has completed all of its outstanding operations before gaining access. This can reduce overlap between scalar and vector memory activity and therefore limit the amount of memory-level parallelism that the processor can exploit.

A related issue is fairness. If one domain continuously generates memory requests, ownership transfer may be delayed for a long time, increasing the latency experienced by the other side. Thus, while the proposed ownership-based policy is sufficient to preserve correctness and keeps the hardware organization simple, it may need to be augmented by a more refined arbitration policy in designs where stronger progress guarantees or higher overlap between scalar and vector accesses are required.

Overall, the memory subsystem reflects the same design principles as the rest of the vector backend: the goal is not to maximize aggressiveness at any cost, but to provide a compact and effective organization that supports vector execution efficiently while preserving correct interaction with the scalar core. In this sense, the proposed memory design complements the configurable execution datapaths of the processor

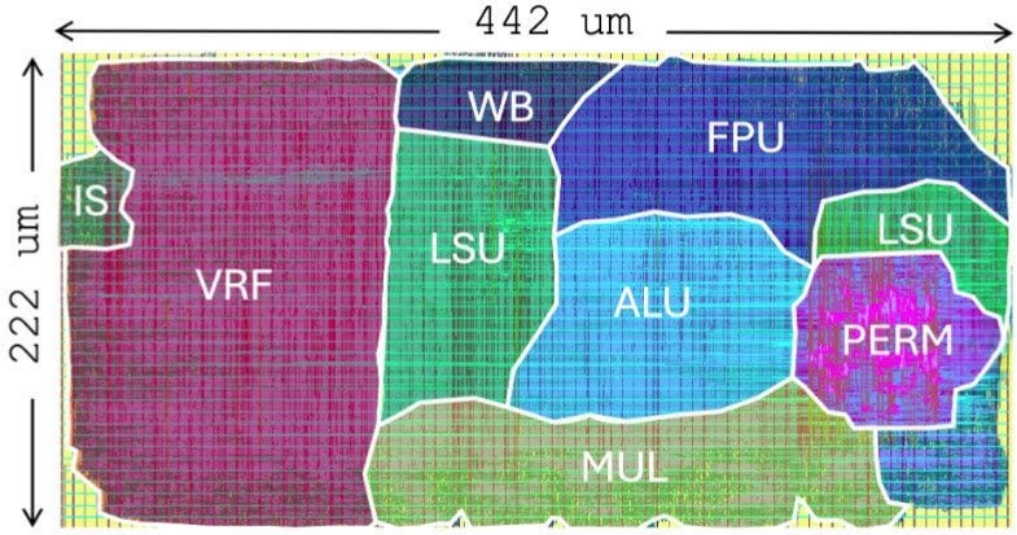


Figure 2.16: Final physical layout of the proposed vector processor. The design includes the integer execution units, floating-point datapath, vector permutation unit, vector register storage, and the supporting control and memory-access logic integrated within a single RVV-capable microarchitecture.

and completes the microarchitectural organization of the short-vector backend.

2.8 Layout of the Proposed Vector Microarchitecture

Having described the main architectural and microarchitectural components of the proposed vector processor, this section examines the physical layout of the complete implemented design. The goal is to assess whether the proposed backend organization can be realized efficiently under realistic physical-design constraints.

The evaluated physical implementation includes a vector ALU and MUL unit for vector integer arithmetic, a vector floating-point unit configured for Bfloat16 operation, and the unified vector permutation unit.

Using the same implementation flow described in Section 2.5.3, the complete vector processor was synthesized down to the layout level. The resulting physical layout, shown in Figure 2.16, confirms that the proposed vector backend can be realized as a complete processor microarchitecture without introducing routing congestion or impractical floorplanning constraints.

From a physical-design perspective, the overall organization remains well balanced. The main execution datapaths, register storage structures, and control logic fit within a compact layout, while the permutation unit integrates cleanly alongside the more conventional arithmetic and floating-point units. This result is important because it shows that the broader functionality introduced in the proposed microarchitecture does not come at the expense of physical implementability.

Overall, the layout-level evaluation confirms that the configurable RVV microarchitecture presented in this chapter is not only architecturally flexible, but also practical from an implementation standpoint. The complete design sustains the intended functionality and execution resources within a compact physical footprint, making it suitable as a realistic vector-processing backend for further architectural explo-

ration and workload execution without excessive physical-design cost.

2.9 Conclusion

This chapter presented a configurable short-vector microarchitecture for RISC-V Vector processing. The proposed organization combines a decoupled scalar–vector interface, a shared vector register file, specialized execution paths, lightweight in-order issue, compact hazard tracking, and a simple memory-ordering mechanism between scalar and vector accesses.

Within this framework, the chapter focused on the microarchitectural mechanisms needed to support flexible RVV execution under area and energy constraints. Larger `LMUL` configurations are handled through instruction sequencing rather than instruction cracking, reducing control complexity. Masked operations are resolved at writeback through element-wise result merging. Permutation instructions are supported by a unified datapath that avoids separate hardware structures for each permutation class while retaining predictable execution behavior.

The physical implementation shows that the proposed backend can be realized as a complete vector microarchitecture without impractical layout overhead. Overall, the design provides a practical foundation for the storage-level optimizations and kernel-aware execution strategies explored in the following chapters.

Chapter 3

Structured Sparsity Acceleration on Vector Processors

3.1 Introduction

As neural networks scale in size and complexity, exploiting inherent model redundancy has become a crucial strategy for deploying these workloads on energy-constrained hardware. Accelerating sparse-dense matrix operations on vector processors requires overcoming the irregular memory access patterns and indirect indexing that typically stall data-parallel execution pipelines. Addressing these bottlenecks demands a tightly coupled algorithm-hardware co-design approach, combining highly optimized loop scheduling and data placement strategies with targeted customizations to the instruction set architecture. To contextualize how these specific execution optimizations unlock accelerator-like efficiency on programmable vector engines, it is essential to first define the nature of neural network redundancy and how it is formally organized.

3.1.1 Unstructured vs. Structured Sparsity

To reduce memory storage and computation cost, the weights of ML models are often pruned, thereby leading to sparse models [48]. This means that zero weights are not stored, and the corresponding calculations are skipped. While sparsity offers high performance and low storage overhead, it may, in some cases, result in less accurate ML models [170, 48]. In such cases, the accuracy loss can be mitigated by retraining the model to adapt to the removal of specific weights, as demonstrated for CNNs [167, 95] and Transformers [53, 57].

The resulting sparsity can be either *unstructured* [34] or *structured* [95, 167]. In unstructured sparsity, there is no constraint on the locations of the zeros, as shown in Figure 3.1(a). In this case, together with the non-zero elements, multiple indices are required to identify the original position of each non-zero element.

By contrast, in structured sparsity, there is an upper limit on the number of non-zero elements that may be present within a block of consecutive elements. For instance, in Figure 3.1(b), for every 4 elements in each row, there are up to two non-zero elements. In most practical applications [95, 83], blocks are

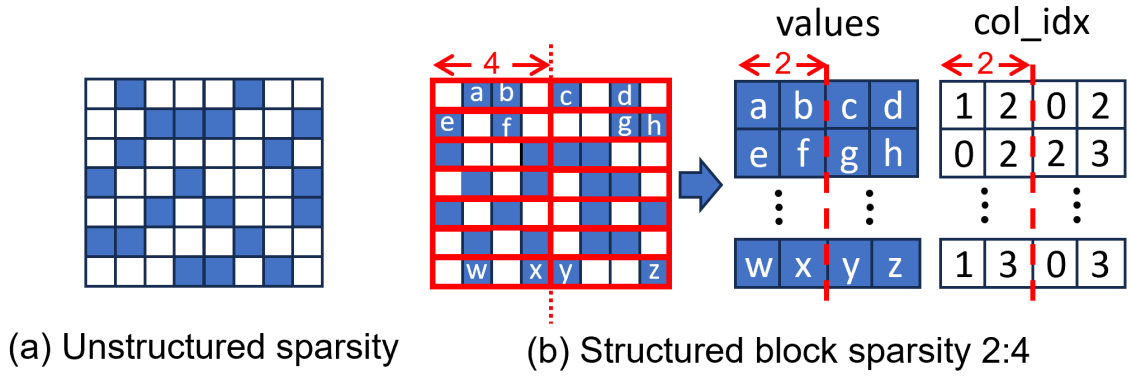


Figure 3.1: Example of (a) unstructured sparsity; and (b) structured block sparsity of 2:4 (i.e., up to 2 non-zero elements in every 4 consecutive elements) and their respective representation. Blue squares represent non-zero elements.

small and N:M sparsity patterns of 1:2, 1:4, or 2:4 are supported, where each block of M elements may contain up to N non-zero elements. As also shown in Figure 3.1(b), for such block-based structures, the indexing (`col_idx`) required to identify the position of each non-zero element inside each block consists of only a few bits, owing to the small size M of each block. Recent work has shown that matrices with unstructured sparsity can be transformed offline to structured-sparse matrices either by row (column) reordering and zero padding [124] or matrix splitting [39]. Tensor units on GPUs can accelerate applications with structured sparse data [167, 95]. Our goal is to extend this capability to programmable CPUs with vector engines, making it applicable to a broader range of current and future workloads.

3.1.2 Vectorizing Sparse-Dense Matrix Multiplication

Unstructured sparsity is already supported in vector processors. Various sparse vector matrix multiplication algorithms have been developed [38, 12], and recently extended [79, 73], with the goal of improving performance by efficiently handling unpredictable sparsity patterns. Moving one step further, VIA [102] adds a scratchpad memory and new custom vector instructions to deal with the complexity of unstructured sparse data.

A key observation is that structured sparsity changes the balance between flexibility and hardware cost. Prior vector-oriented work on sparse computation has largely focused on unstructured sparsity, where irregular access patterns often require more elaborate algorithms, additional indexing overhead, or even dedicated scratchpad-based support [38, 12, 79, 73, 102]. By contrast, the bounded indexing and regular block structure of N:M sparsity make it possible to derive efficient vector kernels with much lower complexity. This motivates the first contribution of this work, which revisits and optimizes the state-of-the-art row-based sparse×dense matrix multiplication algorithm [79, 134, 84] for structured-sparse data on RVV vector processors. In particular, we study how non-zero values and their associated indices should be distributed across the vector and scalar register files so as to reduce register dependencies and improve the effectiveness of interleaved loop unrolling.

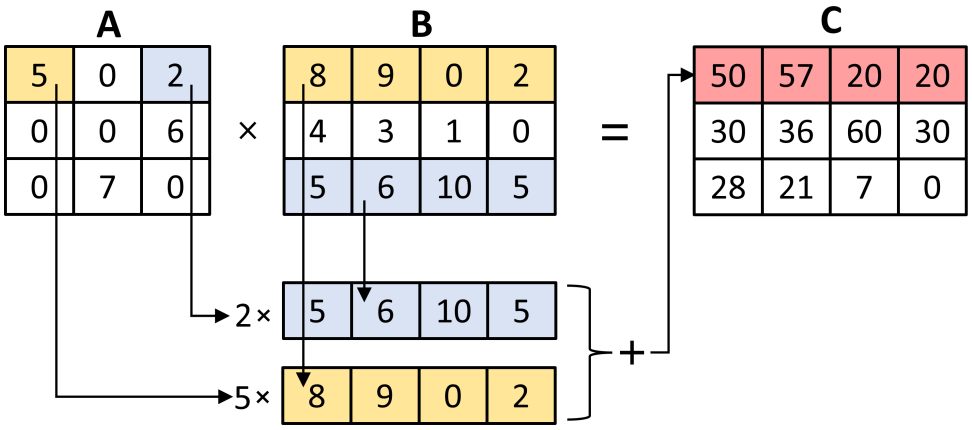


Figure 3.2: Row-wise matrix multiplication to compute output row $C[0,:]$.

3.1.3 Vector ISA customization

Building on this RVV-compliant baseline, a minimal ISA extension called `vindexmac` is introduced. This instruction combines an indirect read from the vector register file with a multiply-accumulate operation, allowing rows or tiles of the dense operand to remain in the vector register file and thereby reducing repeated memory traffic. The motivation for this extension is not to replace the standard vector programming model, but to demonstrate that a small and targeted hardware enhancement can capture additional locality that is otherwise difficult to exploit with existing RVV instructions alone. The proposed instruction was integrated into a decoupled RISC-V vector processor at negligible hardware cost. Decoupled vector processors [20, 101, 94, 33] have independent decode and execute pipelines, separate from the scalar core, which fetches and forwards the corresponding vector instructions. Additionally, a decoupled vector processor accesses memory, and more generally the cache hierarchy, independently of the scalar core, unlike integrated vector engines [135] that share the memory access path with the scalar core.

The evaluation uses structured-sparse CNN workloads to quantify the benefit of the proposed techniques. The results show that careful placement of sparse values and indices across the scalar and vector register files substantially improves the performance of RVV-based sparse×dense multiplication kernels. They further show that the proposed `vindexmac` instruction provides additional gains by reducing instruction count and memory traffic while incurring negligible hardware cost. More broadly, the results demonstrate that structured sparsity can be exploited efficiently on programmable vector architectures without requiring the heavy specialization often associated with sparse acceleration.

3.2 Design Space Exploration of Vector Sparse×Dense Matrix Multiplication

Vectorized matrix multiplications with sparse data can be implemented with many approaches [38, 79, 73]. The row-wise approach [134, 97, 143], also known as Gustavson's algorithm [42], has been shown to be highly effective in computing the matrix product $A \times B$, and it is a better fit to the targeted structured sparsity context. Other approaches [38, 79, 73] target extremely high sparseness and are not as effective

Algorithm 3.1 Row-wise vector matrix multiplication with data placed only in the vector register file

```
1:  set vector length
2:  for i=0 until i=num_of_rows_of_A-1 do
3:      vload v0, A[i,:]                    ▷ load the ith row of A
4:      v2 = {0,0,...,0}                    ▷ Initialize v2 to zero
5:      for j=0 until j=num_of_col_of_A-1 do
6:          vload v1, B[j,:]                ▷ load jth row of B
7:          s0 = A[i,0]                     ▷ move A[i,0] to a scalar reg
8:          v2+= s0 * v1                    ▷ scalar-vector mul-acc
9:          v0 = v0 ≫ 1                     ▷ vector 1 slide to the right
10:     end for
11:     vstore v2, C[i,:]                   ▷ store to mem the ith row of C
12: end for
```

Algorithm 3.2 Row-wise vector matrix multiplication with mixed data placement in the scalar and the vector register files

```
1:  set vector length
2:  for i=0 until i=num_of_rows_of_A-1 do
3:      v2 = {0,0,...,0}                    ▷ Initialize v2 to zero
4:      for j=0 until j=num_of_col_of_A-1 do
5:          vload v1, B[j,:]                ▷ load jth row of B
6:          load s0, A[i, j]                ▷ load the value of A to s0
7:          v2 += s0 * v1                   ▷ scalar-vector mul-acc
8:      end for
9:      vstore v2, C[i,:]                   ▷ store to mem the ith row of C
10: end for
11:
12:
```

Algorithm 3.3 Row-wise vector matrix multiplication with data placed only in the VRF by utilizing `vrgather`

```
1:  set vector length
2:  for i=0 until i=num_of_rows_of_A-1 do
3:      vload v0, A[i, :]                   ▷ load ith row of A
4:      v3 = {0,0,...,0}                    ▷ Initialize v3 to zero
5:      for j=0 until j=num_of_col_of_A-1 do
6:          vload v1, B[j,:]                ▷ load jth row of B
7:          vrgather.vx v2, v0, j  ▷ fill v2 with the jth el. of A
8:          v3 += v2 * v1                   ▷ vector-vector mul-acc
9:      end for
10:     vstore v3, C[i,:]                   ▷ store to mem the ith row of C
11: end for
12:
```

with modest structured sparsity.

Matrix A is assumed to be structured sparse and matrix B is considered to be dense. Algorithmically, all the non-zero elements in a single row of matrix A should be multiplied in parallel with the corresponding rows of matrix B, where the row index of matrix B is determined by the column index of the non-zero value in matrix A. The product of the multiplication is produced row-by-row, as follows:

$$C[i,:] = \sum_{k} A[i,k]B[k,:] \tag{3.1}$$

Figure 3.2 illustrates a simple example of how row 0 of the result matrix C is produced. Row-wise matrix multiplication can be easily vectorized, since each element of A is multiplied with *all* the elements of a row of matrix B. This yields a vector of partial results for a row in the result matrix C.

3.2.1 Exploring data placement in vector row-wise matrix multiplication

The vectorized version of the row-wise matrix multiplication can take three equivalent forms, albeit with different performance characteristics. The first approach, depicted in Algorithm 3.1, assumes that the elements of matrices A and B are placed exclusively in the VRF. Algorithm 3.1 *ignores, for clarity,* any sparseness in matrix A and assumes an arbitrarily large vector length. In line 3, all the elements of row i of matrix A are loaded as a vector into the VRF. Similarly, all the elements of the corresponding row of matrix B are also loaded as a vector in line 6. The vector multiplication between the first element of the loaded row of A and the entire row of matrix B is performed in line 8, which also accumulates the partial results. Following the RISC-V vector ISA format, the multiply-add operation of line 8 is performed on a vector register and a scalar value coming from the scalar register of line 7. The row of A is shifted to the right by one element in line 9 to enable the repetition of the above process for the remaining elements of the same row. The final values of all the elements of the corresponding row of the result matrix C are stored back in memory in line 11. The process is repeated until all the rows of matrix C are produced.

However, the placement of the elements of matrix A in the VRF in Algorithm 3.1 features many unnecessary data transfers between the VRF and the scalar RF (line 7). To tackle this inefficiency, we explore two alternative approaches: (a) keep the elements of matrix B in the VRF and place the elements of matrix A entirely in the scalar RF; and (b) utilize more vector registers in the VRF for the non-zero elements of matrix A.

Starting with the first approach, Algorithm 3.2 presents the row-wise vector matrix multiplication that follows the above-mentioned *hybrid* data placement. Specifically, rather than loading the rows of

Algorithm 3.1-S Row-wise vector sparse-dense matrix multiplication with data placed only in the vector register file

```
1:  set vector length
2:  for i=0 until i=num_of_rows_of_A-1 do
3:      vload v0, values[i, :]      ▷ load the non-zero val. of A
4:      v2 = {0, 0, ..., 0}         ▷ Initialize v2 to zero
5:      for j=0 until j=(num_of_col_of_A/M)*N-1 do
6:          block_id =⌊j/N⌋
7:          load s1, col_idx[i, j]  ▷ load a column index
8:          s1 += block_id * M      ▷ calculate actual col. idx
9:          vload v1, B[s1, :]      ▷ load the selected row of B
10:         s0 = A[i, 0]            ▷ move A[i, 0] to a scalar reg
11:         v2+= s0 * v1            ▷ scalar-vector mul-acc
12:         v0 = v0 ≫ 1             ▷ vector 1 slide to the right
13:     end for
14:     vstore v2, C[i, :]          ▷ store to mem the ith row of C
15: end for
```

Algorithm 3.2-S Row-wise vector sparse-dense matrix multiplication with mixed data placement

```
1:  set vector length
2:  for i=0 until i=num_of_rows_of_A-1 do
3:      v2 = {0, 0, ..., 0}         ▷ Initialize v2 to zero
4:      for j=0 until j=(num_of_col_of_A/M)*N-1 do
5:          block_id =⌊j/N⌋
6:          load s1, col_idx[i, j]  ▷ load a column index
7:          s1 += block_id * M      ▷ calculate actual col. idx
8:          vload v1, B[s1, :]      ▷ load the selected row of B
9:          load s0, A[i, j]        ▷ load the value of A to s0
10:         v2 += s0 * v1           ▷ scalar-vector mul-acc
11:     end for
12:     vstore v2, C[i, :]          ▷ store to mem the ith row of C
13: end for
14:
15:
```

Algorithm 3.3-S Row-wise vector sparse-dense matrix multiplication by utilizing the `vrgather` instruction

```
1:  set vector length
2:  for i=0 until i=num_of_rows_of_A-1 do
3:      vload v0, values[i, :]      ▷ load the non-zero val. of A
4:      v3 = {0, 0, ..., 0}         ▷ Initialize v3 to zero
5:      for j=0 until j=(num_of_col_of_A/M)*N-1 do
6:          block_id =⌊j/N⌋
7:          load s1, col_idx[i, j]  ▷ load a column index
8:          s1 += block_id * M      ▷ calculate actual col. idx
9:          vload v1, B[s1, :]      ▷ load the selected row of B
10:         vrgather.vx v2, v0, j   ▷ fill v2 with the jth el. of A
11:         v3 += v2 * v1           ▷ vector-vector mul-acc
12:     end for
13:     vstore v3, C[i, :]          ▷ store to mem the ith row of C
14: end for
15:
```

matrix A into the VRF, each element of a row of A is, instead, loaded (line 6) and used (line 7) one-by-one directly through the scalar RF. Thus, the multiply-add operation in line 7 directly utilizes the data placed in the scalar register 'S0,' thereby eliminating the need for vector-to-scalar register moves and vector slides. Note that the data type of the elements dictates the type of the register file into which the non-zero values will be stored: if the non-zero values in Algorithm 3.1 and Algorithm 3.2 are of type 'float,' they can be stored into the floating-point register file, thereby alleviating the overall pressure on the scalar register file.

Alternatively, the second approach is to keep matrix A in the VRF and utilize more vector registers, as shown in Algorithm 3.3. Line 3 shows that a row of matrix A is loaded into the VRF. We need to access each one of those elements one after the other and multiply them with the corresponding row of B. This is achieved using the `vrgather.vx` instruction. Said instruction copies the jth element of 'v0' into every element of the vector register 'v2.' The latter vector register is then used as an operand in the multiply and accumulate operation. The use of the `vrgather.vx` instruction enables the operation of the multiply-and-accumulate operation (line 8) to be performed with 3 vector operands, as opposed to the approach in Algorithm 3.2, where a scalar and 2 vector operands are used.

3.2.2 Vector Row-wise matrix multiplication for structured-sparse data

Algorithms 3.1, 3.2 and 3.3 can be reformulated to support a *structured-sparse* matrix A. In this case, only the non-zero elements of each row of A are stored, together with their corresponding index, i.e., their original position within a block of size M in the row of matrix A. The reformulation of Algorithms 3.1, 3.2 and 3.3 is shown in Algorithms 3.1-S, 3.2-S and 3.3-S respectively, where all mandatory changes/additions are highlighted with color.

To highlight the added changes we will use as an example Algorithm 3.3-S. In this case, the non-zero values of matrix A are loaded into the VRF, as shown in line 3 of Algorithm 3.3-S. On the contrary, the column indexes are stored in a *scalar* register in the scalar RF (as per line 7 in the algorithm), because they require some pre-processing before they can be used. The pre-processing steps arise from the structured sparsity format itself, which limits the range of the column indexes. Specifically, the column index refers to the position of the non-zero element inside a block of size M. Consequently, all column indexes have a range $[0, M-1]$, even though each row of matrix A typically consists of *multiple* blocks of size M.

Hence, whereas the column index 'j' in Algorithm 3.3 spans the entire row of matrix A (i.e., all its columns), the same index has a different meaning in Algorithm 3.3-S. Column index 'j' now spans only

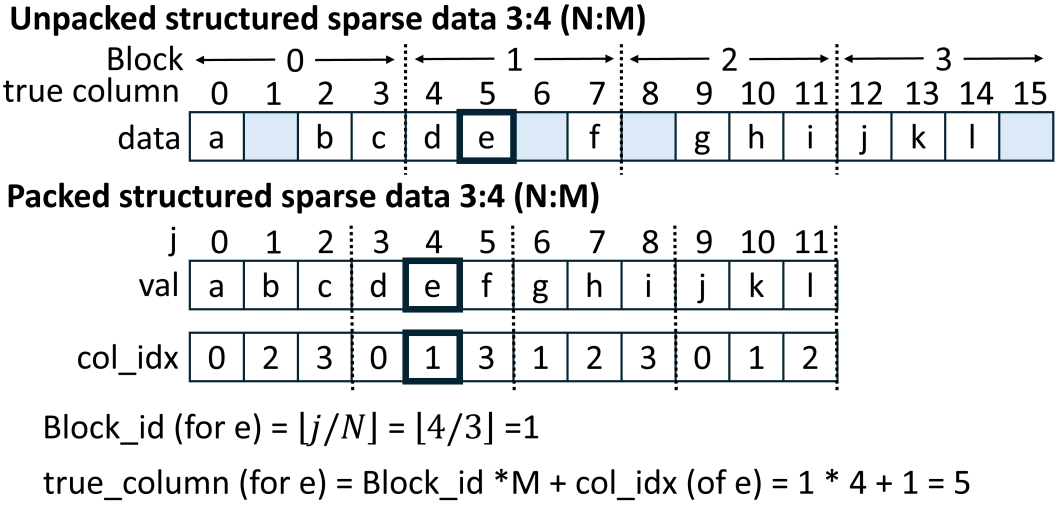


Figure 3.3: An example of recovering the actual column index of an element through the column index that is stored in the memory and the block_id.

the number of non-zero elements in a particular row of matrix A, as shown in line 5 of Algorithm 3.3-S. In this form, the index cannot be used directly to load the corresponding row of matrix B. Instead, the actual column index that points to a specific column across the *entire* row of matrix A must first be retrieved. To compute the actual column index, we must identify the particular block in which the corresponding non-zero element resides. This is performed in line 6 of the algorithm, where the block index is calculated by the integer division of the position 'j' of the non-zero element with the maximum permissible number of non-zero elements N in each block of size M. Once the block ID is identified, it is multiplied with the size M of each block and added to the original column index that was loaded from memory, as shown in line 8. This operation fully retrieves the actual column index that spans the entire row of matrix A. When N and M are multiples of 2, multiplication and division are performed with left and right shifts respectively.

An example of how the actual column index can be retrieved is depicted in Figure 3.3, assuming 3:4 (N:M) structured block sparsity. The example focuses on element 'e.' First, the block in which element 'e' resides (block_id) is identified by performing an integer division between the position of 'e' in the array of non-zero values (i.e., position 4), and the number N=3 of permitted non-zero elements in each block of size M. Next, the actual column index is computed by multiplying the block_id with the size M=4 of each block. Finally, this value is added to the column index of element 'e' that is stored in memory (i.e., the value 1) to retrieve the actual column within the row of A where element 'e' resided prior to compression.

To avoid such indexing calculations, full-width column indexes can be stored at the row level. However, as demonstrated in [95] this approach would significantly increase storage overhead for the targeted sparsity levels of 1:4 or 2:4.

Similarly, the low-to-medium levels of sparsity in the context of a sparse-dense multiplication make it less preferable to gather the non-zero values of A using indexed load instructions [73]. This approach would introduce additional runtime overhead, since indexed vector load/store instructions may generate

a substantial number of cache misses from only a few vector instructions.

Continuing with Algorithm 3.3-S, the actual column index retrieved in line 8 is used to load the corresponding row of matrix B into the VRF in line 9. The `vrgather` instruction is used in line 10 to copy the currently-processed non-zero element into all the vector register elements of vector 'v2.' Note that only the rows of matrix B that correspond to the column indexes of the non-zero elements of A participate in the multiply-and-accumulate operation (line 11). Each iteration of the inner loop moves to the next *non-zero* element in the current row of A, until the end of the row is reached.

Algorithms 3.1-S and 3.2-S employ a similar approach to retrieve the non-zero values of A and their corresponding column indices.

3.2.3 Loop unrolling row-wise vector sparse×dense matrix multiplication

Following the practice explored in other vector-based parallel matrix multiplication kernels [54, 4], and to increase the scope for instruction-level parallelism, both loops (outer and inner) of Algorithm 3.3-S can be unrolled. Indicatively, Algorithm 3.4 unrolls both loops by a factor of 2. The unrolling of the outer loop allows the processor to simultaneously work on *two consecutive* rows of matrix A, i.e., to simultaneously produce two consecutive rows of output matrix C. Similarly, the unrolling of the inner loop allows the processor to simultaneously work with two consecutive non-zero elements in each of the two consecutive rows of matrix A.

Lines 8 to 18 in Algorithm 3.4 refer to operations performed using the non-zero element 'j' of consecutive rows 'i' and 'i+1' of matrix A. As can be seen, due to the loop unfolding, each instruction (except the one calculating the block_id) in the inner loop of Algorithm 3.3-S is hereby duplicated. The two multiply-accumulate instructions in lines 17 and 18 generate two vectors of partial results for rows 'i' and 'i+1' of the output matrix C. Similarly, lines 19 through 29 repeat the same pattern for the next non-zero elements in the same two rows of matrix A; i.e., the non-zero element 'j+1' of consecutive rows 'i' and 'i+1.'

Since each loop was unrolled by a factor of 2, one would expect the final impact on the required scalar and vector registers to be four-fold. However, this cost is avoided through the *interleaving* of the two loops, which allows for the reuse of the same scalar and vector registers and reduces read-after-write dependencies. As can be seen in Algorithm 3.4, instructions from different loop iterations are executed in interleaved manner. The instructions that belong to the same loop are depicted with the same color. In all cases, no two instructions of the same loop are executed one after the other. Interleaving allows for exactly the same scalar and vector registers used in lines 8 to 18 to be reused in lines 19 to 29. This approach minimizes the amount of registers in use, while also maintaining the physical parallelism that is inherent to non-dependent instructions. Also, interleaving allows distinct loop iterations to share the column-index calculations. This effectively reduces the number of instructions after unrolling.

Overall, Algorithm 3.3-S requires 1 scalar and 4 vector registers, whereas the loop-unfolded Algorithm 3.4 requires 2 scalar and 8 vector registers. In general, (a) increased pressure on the register file, and (b) code bloating (that could adversely affect the cache performance) are the two well-known side effects of loop unrolling. Hence, loop unrolling should be applied judiciously up to the point where the side effects of the unrolling start to cause diminishing returns to the reaped performance improvement.

Algorithm 3.4 Unrolled row-wise vector sparse-dense matrix multiplication by utilizing the `vrgather.vx` instruction

```
1:  set vector length
2:  for i=0 until i=num_of_rows_of_A-1 , i+=2 do
3:      vload v0, values[i, :]
4:      vload v1, values[i+1, :]
5:      v6 = {0, 0, ..., 0}
6:      v7 = {0, 0, ..., 0}
7:      for j=0 until j=(num_of_col_of_A/M)*N-1 , j+=2 do
8:          block_id =⌊j/N⌋
9:          load s0, col_idx[i, j]
10:         load s1, col_idx[i+1, j]
11:         s0 += block_id * M
12:         s1 += block_id * M
13:         vload v4, B[s0, :]
14:         vload v5, B[s1, :]
15:         vrgather.vx v2, v0, j
16:         vrgather.vx v3, v1, j
17:         v6 += v2 * v4
18:         v7 += v3 * v5
19:         block_id =⌊(j + 1)/N⌋
20:         load s0, col_idx[i, j+1]
21:         load s1, col_idx[i+1, j+1]
22:         s0 += block_id * M
23:         s1 += block_id * M
24:         vload v4, B[s0, :]
25:         vload v5, B[s1, :]
26:         vrgather.vx v2, v0, j+1
27:         vrgather.vx v3, v1, j+1
28:         v6 += v2 * v4
29:         v7 += v3 * v5
30:     end for
31:     vstore v6, C[i, :]
32:     vstore v7, C[i + 1, :]
33: end for
```

3.3 Speeding up sparse-dense matrix multiplication with a custom instruction

The implementation of sparse-dense multiplication eliminates unnecessary multiplications, due to the structured-sparse format of A. However, one crucial bottleneck of the computation is the abundance of *vector loads* from memory for the rows of matrix B (Line 9 in Algorithm 3.3-S). To tackle this issue, we leverage the structured sparsity of matrix A to reduce memory traffic and allow the computations to use *local* data that already resides in the vector register file.

The proposed optimization combines: (a) loading rows of matrix B in tiles in the vector register file; and (b) a custom index-multiply-accumulate instruction that replaces the vector loads of matrix B with indirect reads of the vector register file.

The key attribute that enables the loading of matrix B in the register file in groups of multiple rows (instead of the row-by-row approach) is the well-defined, regular structure in the format of matrix A. Since the sparsity of A is – by construction – *structured*, the blocks within said matrix have a constant and known size. In turn, this implies that the column indexes of the non-zero element values in A are 'bounded' by the block size M, i.e., all `col_idx` values reside within the range $[0, M-1]$. Recall that the block size M is the number of consecutive elements within a row of A that can contain up to a specific number (N) of non-zero elements. Exploiting this trait of structured sparsity, we may pre-load as many rows of matrix B as our vector register file can accommodate (with some restrictions, as will be explained in Section 3.3.2) and be sure that the column indexes in matrix A will only point to those local rows. On the contrary, with unstructured sparsity, the column indexes are, essentially, 'unbounded' and could point to any row of B (thereby rendering the pre-loading of specific rows of B in the register file futile).

The tile size of matrix B that is pre-loaded in the vector register file is $L \times Vector\_Length$; i.e., L rows, with each row having $Vector\_Length$ columns. The number of rows L must be a multiple of M.

3.3.1 The proposed vector index-multiply-accumulate vindexmac instruction

Since multiple rows of matrix B are now loaded in the VRF, there is a need to *index* into the register file to access a specific row. This is achieved by the proposed new instruction that effectively performs a scalar-vector multiply-and-accumulate operation, in combination with the indexing in the VRF. The scalar value refers to a non-zero element of a row of matrix A and the vector refers to a row of matrix B that is pre-loaded in the VRF and indirectly read by the provided column index.

Overall, the new instruction aims to alleviate the sparse-dense matrix multiplication process from the continuous vector loads of the rows of matrix B. Instead, the custom instruction reads the corresponding pre-loaded rows of matrix B directly from the vector register file and operates on them. With this objective in mind, the new `vindexmac` instruction is defined as follows:

`vindexmac.vx vd, vs2, rs` $\rightarrow$ `vd[i] += vs2[0] * vrf[rs`$_{4:0}$`][i]`

The instruction has three operands: one vector destination register (`vd`) and two source registers. One source register is a vector (`vs2`) and the other is a scalar (`rs`), as per the RISC-V ISA requirement for `.vx` instructions. To execute `vindexmac`, the scalar register rs is read and its contents (only the 5 least significant bits are actually needed) are used as an address to the vector register file. The content (i.e., value) of the vector register that is read via the address contained in `rs` is multiplied with the least significant element of vector register `vs2`. The operation of `vindexmac` is visualized in Figure 3.4.

3.3.2 Sparse-Dense Matrix Multiplication using vindexmac

Algorithm 3.5 demonstrates the use of the new vindexmac instruction for a tile of B, with the non-zero values of a single row of matrix A stored in a vector register. The essential differences between Algorithm 3.5 and the previous algorithms are highlighted. As shown in lines 2 and 4, a tile of matrix B is pre-loaded into the VRF. In line 6, the non-zero values of a row of matrix A are loaded into the VRF,

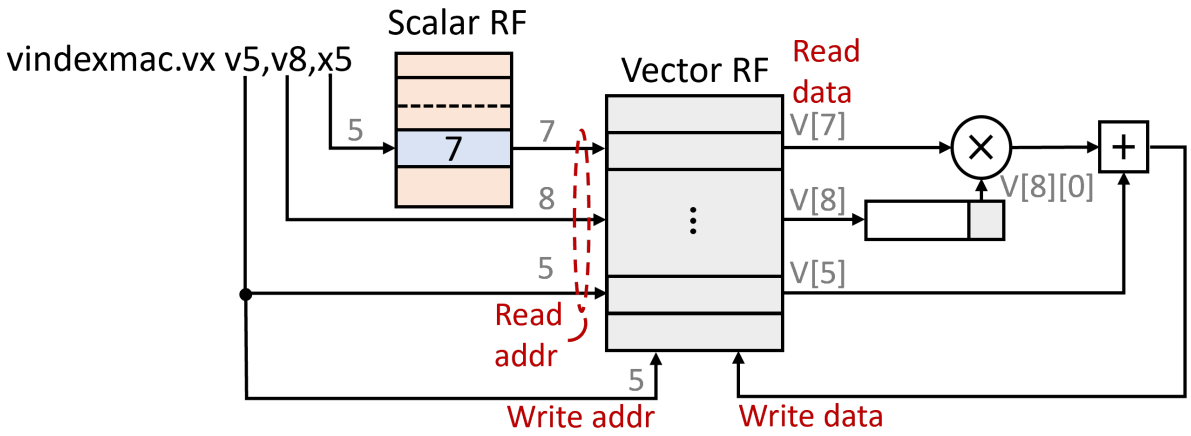


Figure 3.4: The operation of the proposed `vindexmac` instruction. The contents of the scalar register are used to address a specific vector register. The vector read is multiplied with the least significant element of another vector register that is read in parallel. The result of the multiplication is accumulated with the previous contents of the vector destination register.

while the corresponding column indexes remain in the scalar RF. The use of the new `vindexmac` instruction is shown in line 12. The end result is the elimination of the continuous vector loads of rows of matrix B, which take place in line 9 of Algorithm 3.3-S. Since `vindexmac` uses only element 0 of vector register `vs2`, a vector slide by one element position in `vs2` is enough to proceed to the next non-zero element of the current row of matrix A (Line 13 of Algorithm 3.5).

Algorithm 3.5 Vector sparse-dense matrix multiplication with the use of the new `vindexmac` instruction for **a tile** of B

```
1:  set vector length
2:  for k=0 until k=L-1 do
3:      vload vk, B[k, :]                          ▷ preload L rows of B
4:  end for
5:  for i=0 until i=num_of_rows_of_A-1 do
6:      vload vs2, values[i, :]                    ▷ load non zeros of ith row
7:      vload vd, C[i, :]                          ▷ load the ith row of C
8:      for j=0 until j=L*(N/M)-1 do
9:          block_id =⌊j/N⌋                        ▷ calculate the block id
10:         load s1, col_idx[i, j]                 ▷ load col. idx of A to s1
11:         s1 += block_id *M                      ▷ calculate actual col idx
12:         vindexmac.vx vd, vs2, s1
13:         vs2 = vs2 ≫ 1                          ▷ vector 1 slide to the right
14:     end for
15:     vstore vd, C[i, :]                         ▷ store to mem row i of C
16: end for
```

The number of rows L of the tile of matrix B that are loaded into the VRF and indexed by the `vindexmac` instruction is limited by the number of available vector registers. In practice, we consider that a tile of matrix B may consume half of the available vector registers, i.e., $L = 16$. Due to the symmetry of matrix multiplication, the L rows of matrix B correspond to a set of L columns of matrix A. Since matrix A follows N:M structured sparsity, it means that, in those L columns, one can find

only $L \cdot \left(\frac{N}{M}\right)$ non-zero elements. Effectively, this property results in many more iterations in the outer loop (lines 5–16) of Algorithm 3.5, since the inner loop (lines 8–14) is executed for a small number of non-zero elements.

To improve utilization and to take advantage of the fact that the vector load for a row of matrix A (line 6 of Algorithm 3.5) can bring – at once – $VectorLength$ elements in vector register `vs1`, we need to consider more tiles of matrix B. This approach leads to a reformulation of Algorithm 3.5, which is shown in Algorithm 3.6. An intermediate loop is inserted (using index m), which is responsible to fetch and operate on the tiles of matrix B that correspond to a complete row of non-zero elements of matrix A. The non-zero elements of a row of matrix A loaded in line 3 of Algorithm 3.6 are all processed by reloading the corresponding rows of matrix B in phases. Specifically, the appropriate tiles of matrix B are loaded in lines 6–8, in each phase.

Algorithm 3.6 Vector sparse-dense matrix multiplication with the use of the new `vindexmac` instruction by utilizing a complete row of $VectorLength$ non-zero elements of matrix A, which correspond to more than one tile of matrix B

```
1:  set vector length
2:  for i=0 until i=num_of_rows_of_A-1 do
3:      vload vs1, values[i, :]                          ▷ load values of row i
4:      vload vd, C[i, :]                                ▷ load the ith row of C
5:      for m=0 until m=(M/N)*VL/L do
6:          for k=L*m until k=(m+1)*L-1 do
7:              vload vk, B[k, :]                        ▷ load a tile of B
8:          end for
9:          for j=L*(N/M)*m until j=L*(N/M)*(m+1)-1 do
10:             block_id =⌊j/N⌋ % L                      ▷ calculate the block id
11:             load s1, col_idx[i, j]                   ▷ load col. idx of A to s1
12:             s1 += block_id * M                       ▷ calculate actual col idx
13:             vindexmac.vx vd, vs1, s1
14:             vs1 = vs1 ≫ 1                            ▷ vector slide to the right
15:         end for
16:     end for
17:     vstore vd, C[i, :]                               ▷ store to mem row i of C
18: end for
```

The number of required reloading phases m is determined by (a) the vector registers L available for the loading of a tile of matrix B, (b) the sparsity pattern $N{:}M$, and (c) the maximum vector length of the processor VL. As stated earlier, in every L columns of matrix A, we can find $L\ \left(\frac{N}{M}\right)$ non-zero elements. Therefore, we need to take m groups of L columns of matrix A to reach the $VectorLength$ (VL) non-zero elements that can be present in one vector register, i.e., $m\,L\ \left(\frac{N}{M}\right) = VL$. Solving for m, we get that

$$m = \left(\frac{M}{N}\right)\left(\frac{VL}{L}\right) \tag{3.2}$$

This limit is used for the bounds of m in line 5 of Algorithm 3.6. For the case of 1:4 structured sparsity, and assuming that we allocate half of the vector register file for the tiles of matrix B (i.e., L=16), then $m = (4/1)(VL/16) = VL/4$. It is crucial to point out that when the maximum value of m is 1, Algorithm 3.5 is preferred over Algorithm 3.6. When $m = 1$, there is no need to interchange the tile of

matrix B that is loaded in the vector register file. Instead, Algorithm 3.5 would, by construction, avoid the continuous loads that refer to the same rows of matrix B.

To further improve performance, the sparse×design matrix multiplication kernel that uses the newly proposed `vindexmac` instruction (as shown in Algorithm 3.6) could be unrolled across three dimensions: (a) the elements of the same row of matrix A (inner loop); (b) the tiles of matrix B (intermediate loop); and (c) across the various output rows (outer loop). The proposed approach, which preloads a tile of matrix B that utilizes a significant portion of the VRF, limits the depth of loop unrolling. To optimally apply unrolled sparse-dense matrix multiplication with the `vindexmac` instruction, a balanced approach is required, combining unrolling and appropriate tiling of B to efficiently utilize the 32 vector registers of a RISC-V architecture.

3.3.3 Tiled execution for larger matrices A and B

Algorithm 3.6 can work on a matrix A with any number of rows. However, it assumes that the non-zero elements of A loaded into a vector register at each step correspond to at most $m \times L$ columns. This is the reason for loading m tiles of B with L rows each at each round. Additionally, Algorithm 3.6 assumes that all columns of B can fit into one vector register.

In the general case, where the columns of A (and consequently the rows of B) exceed $m \times L$, and the columns of B are larger than VL, we need to apply Algorithm 3.6 repeatedly using the dataflow shown in Figure 3.5. For one vertical segment of VL columns of B, we process all rows of A in groups of $m \times L$ columns. In particular, we first process all rows of A that belong to the same vertical segment of $m \times L$ columns before moving to the next segment. Within each vertical segment of A, we follow the execution order of Algorithm 3.6, keeping one group of $m \times L$ rows of B stationary. Once all vertical segments of A are processed, we repeat the process for subsequent vertical segments of B by fetching the next VL columns and restarting.

This tiled dataflow can be employed also when executing sparse-dense matrix multiplication to a multicore environment. The computation for every vertical segment of B consisting of VL columns can be assigned to a different core. This partitioning, allows each core to operate on independent segments of the output matrix without facing any conflicts. However, with this dataflow, the elements of structured-sparse matrix A are accessed by all cores in parallel. The benefits of implementing sparse-dense matrix multiplication using `vindexmac` in a multicore setup under this dataflow is quantified in Section 3.4.5.

3.3.4 Hardware support for `vindexmac` execution

The vector register file in the RISC-V ISA is instrumental in supporting efficient vector processing. It consists of 32 vector registers, each capable of storing a large volume of data. Each register's length is an implementation-defined constant parameter decided by the processor designer. To support all available vector instructions, including three-operand instructions (e.g., `vmacc`, `vmadd`, etc.), the vector register file is required to have three read ports and one write port.

To execute the `vindexmac` instruction, we need to access both the scalar and the vector register files. This is an inherent attribute of all RISC-V scalar-vector instructions that have the letter '`x`' in the suffix, e.g., `.vx`, `.vxm`, `.wx`, etc.

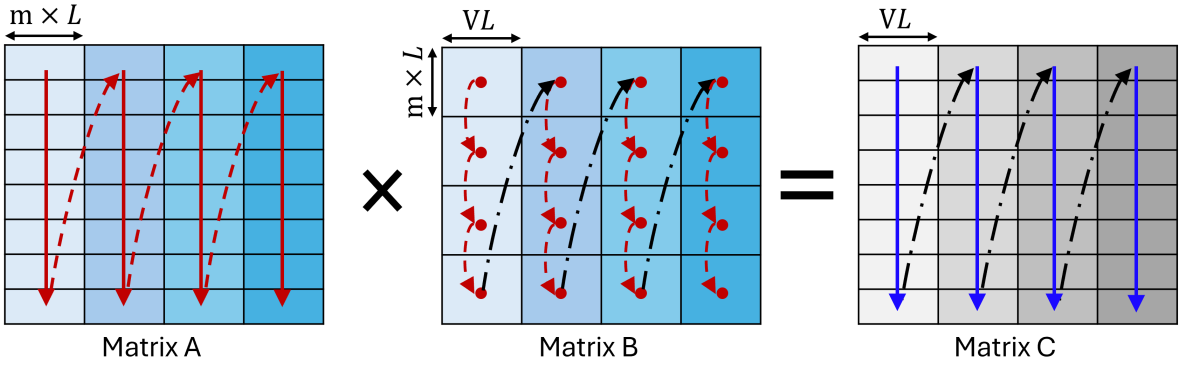


Figure 3.5: The order of execution of Algorithm 3.6 across vertical segments of A and B.

In this chapter, we target vector processors that follow a so called *decoupled* architecture, which consists of a scalar core that is responsible for instruction fetching and orchestration of execution, and a vector engine that executes the vector operations received from the scalar core [33, 21, 142, 20, 101, 94]. Since, at any given time, there may be many vector instructions that require the values of scalar registers, the scalar core is responsible to transfer these values to the vector processor, together with the vector instructions themselves.

Therefore, the value of the scalar register `rs` required by `vindexmac` to address the vector register file is already provided by the scalar core. The given address drives one of the read ports of the vector register file, which outputs the requested vector operand. Another read port is used to read the elements of `vs2` and the third one reads `vd`, as required by all multiply-and-accumulate scalar-vector instructions already present in the RISC-V ISA. Therefore, the only hardware requirement of the new `vindexmac` instruction is the addition of a multiplexer in front of the address bus of one of the read ports of the vector register file, which selects between `vs1` for normal vector arithmetic operations, or the 5 least significant bits of `rs` (as required by `vindexmac`). In other words, by re-using the hardware infrastructure of existing scalar-vector multiply-add instructions, the new instruction does *not* require an additional read port in the vector register file. Instead, it merely requires a 5-bit 2-to-1 multiplexer in front of an existing read port.

Specifically, a 32-register VRF, with each register being 512 bits wide (VL=16 for 32-bit words), with 3 read and 1 write port, synthesized with Cadence Genus at the 28 nm technology node for a 1 GHz clock period, requires a hardware area of 56,527 μm^2. The additional area needed to accommodate the address multiplexing logic is just 4 μm^2 extra.

Similarly, in fully integrated scalar-vector architectures [117, 136], the `vindexmac` instruction would be implemented in exactly the same manner as any of the other `.vx` instructions in a fully integrated scalar-vector setup. The only difference is that the scalar value provided would be used to drive one of the read ports of the vector register file, rather than directly participating in the computation.

Being a `.vx` instruction, `vindexmac` follows the standard encoding dictated by the RISC-V ISA for scalar-vector instructions [116]. However, it terms of the instruction functionality, there is a slight deviation from typical scalar-vector operations. The vector register identifier `vs2` is used only for its least significant element `vs2[0]`, i.e., it plays the role of the scalar value. The actual vector is fetched

Table 3.1: Simulated Processor Configuration

Scalar core	• RISC-V ISA (RV64GC), 8-way-issue out-of-order, 16-entry LSQ, 90 physical integer and 90 physical floating-point registers, 60-entry ROB • L1I cache: 1-cycle hit latency, 4-way, 64KB • L1D cache: 2-cycle hit latency, 4-way, 64KB • L1I and L1D have 64B cache line
Vector engine	• 512-bit vector engine with 16-lane configuration (32-bit elements $\times$ 16 execution lanes) • The vector engine is connected directly to the L2 cache through 16 store queues and 16 load queues
L2 cache	• 8-way, 8-bank • 8-cycle hit latency, 512KB shared by both the big core and the vector engine • 64B cache line
Main Memory	DDR4-2400, 19.2 GB/s memory bandwidth

via an indirect read using the 5 least significant bits of `rs` as an address into the vector register file.

3.4 Experimental Evaluation

The experimental evaluation presented in this section has two goals. The first one is to identify the performance limits of the introduced row-based formulation of sparse-dense matrix multiplication using only the current definition of the RISC-V ISA; i.e., using only existing instructions. The evaluation includes the execution of three *pruned* state-of-the-art CNNs – Resnet50 [44], DenseNet121 [52] and InceptionV3 [141] – under various loop unrolling scenarios on a superscalar out-of-order processor augmented with a decoupled vector unit. All calculations in all three CNNs are performed in the fp32 data format. The pruned CNNs employ structured block sparsities of 1:4 and 2:4. Note that, as demonstrated in [167] and [95], the selected structured-sparsity patterns do not change the classification accuracy of the specific CNNs, as compared to their fully dense counterparts.

After identifying the configuration that offers the greatest speedup when using the current RISC-V ISA, the obtained performance is compared with the performance achieved by the structured-sparse matrix multiplication kernels that employ the proposed new `vindexmac` instruction. For the latter case, various alternatives are also explored and the one that achieves the lowest execution time is used in the comparisons.

3.4.1 Experimental setup

For all experiments, we utilize a fully implemented decoupled vector unit connected to an out-of-order superscalar processor, i.e., model 1bDV in [142]. The choice of a high-end superscalar core (connected to the vector engine) aims to safeguard that the non-vector side of the processor does not skew the performance results of the vector engine. In other words, the superscalar processor helps to "isolate" the performance of the vector unit by ensuring that there are no "bottlenecks" on the non-vector side. As

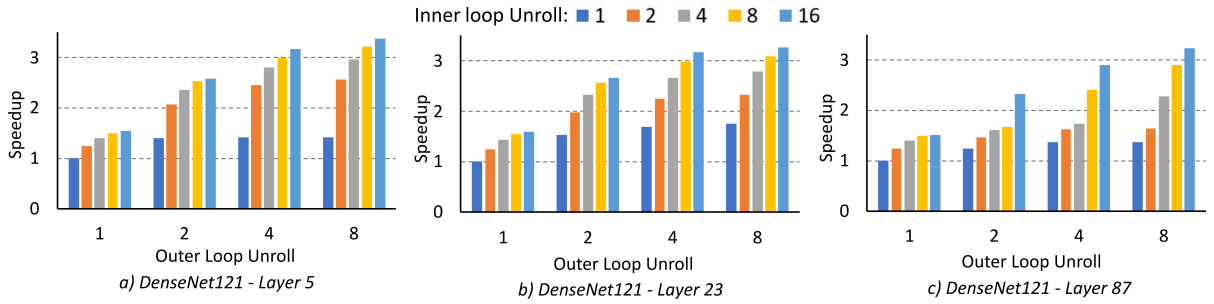


Figure 3.6: The performance of the optimized sparse-dense matrix multiplication using the current definition of the RISC-V ISA (Algorithm 3.3-S) used for the execution of three CNN layers of Densenet121 [52], assuming a 1:4 sparsity pattern: (a) Layer 5; (b) Layer 23; and (c) Layer 87. The reported cases examine the combined effect of the unrolling of the inner and outer loops of Algorithm 3.3-S. The speedup is reported relative to the performance of the rolled variant.

such, the reported results highlight purely the efficacy of the new vector framework, without any indirect effects/noise from the scalar side.

This design was modeled in the gem5 simulator [14, 85] and the salient design parameters of the simulated processor setup are summarized in Table 3.1. The new `vindexmac` instruction was incorporated in the RISC-V GNU toolchain and its operation was implemented in the decoupled vector engine in gem5.

The convolutions of each layer of the examined CNNs are mapped to sparse-dense matrix multiplications $A \times B$ [95]. Matrix A includes the structured-sparse weights and matrix B the input features of the corresponding CNN layers. Input features are considered dense, since there is no clear statistical attribute that can be exploited for them. Even if part of the input features contain zero values generated by the corresponding ReLU activation functions in each layer, their number is highly sensitive to the actual input values and filter weights.

3.4.2 Identifying peak performance using only instructions from the current RISC-V ISA

Our initial goal is to examine which of the approaches highlighted in Algorithms 1S–3S (that use only the current definition of the RISC-V ISA) offer the best scaling in execution time for various Sparse-dense matrix multiplications and and compare also the best of them to other state-of-the-art approaches.

As discussed in Section 3.2.3, the inner and the outer loops of all algorithms can be unrolled to increase performance. Outer-loop unrolling corresponds to processing multiple rows of sparse matrix A in parallel, while inner-loop unrolling enables the simultaneous processing of multiple non-zero elements of the same row of matrix A, in a flattened form. To highlight how unrolling can improve runtime, we perform several experiments using three representative layers from the DenseNet121 CNN [52], assuming 1:4 sparsity.

The effect of optimized unrolling presented in Section 3.2.3 is examined first for Algorithm 3.3-S. The speedup achieved in each case, relative to the rolled version of Algorithm 3.3-S, is shown in Figure 3.6. Each bar corresponds to a different inner-loop unroll factor, while outer-loop unrolling factors are represented by the different sets of bars along the x-axis. For instance, the unrolled example shown

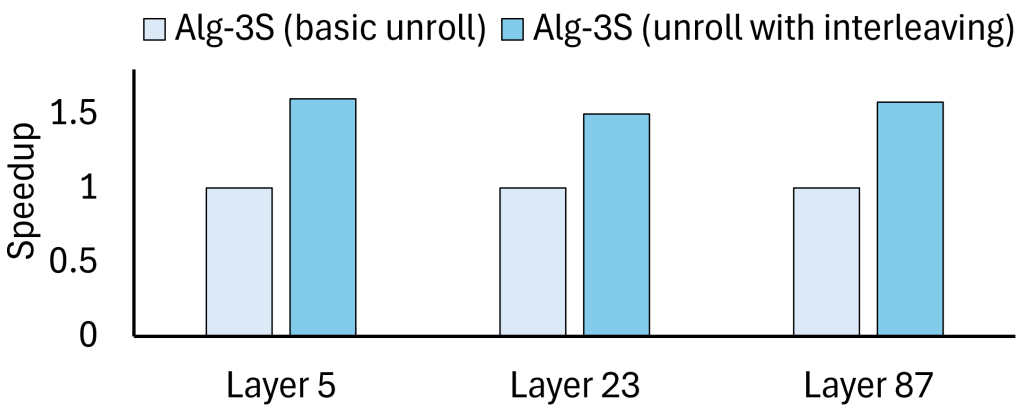


Figure 3.7: Speedup of interleaved unrolling on Alg-3S, normalized to the speedup of Clang's [71] basic unrolling methodology with inner and outer loop unrolling factors of 16 and 8, respectively.

in Algorithm 3.4 of Section 3.2.3 corresponds to the case of using an unrolling factor of 2 for both the inner and outer loops. As demonstrated in Figure 3.6, increasing unrolling improves performance in all cases, reaching a speedup of more than $3\times$ relative to the baseline implementation.

Speedup is maximized when the inner loop can process 16 non-zero elements in parallel and the outer loop is unrolled by a factor of 8. the factors are constrained by the "size" of the the vector unit. The inner loop unroll factor of 16 corresponds to the hardware $VectorLength$ shown in Table 3.1. These specific unrolling factors are directly linked to the hardware; i.e., the factors are constrained by the "size" of the the vector unit. The inner loop unroll factor of 16 corresponds to the hardware $VectorLength$ shown in Table 3.1. Similarly, the unrolling factor of 8 of the outer loop is limited by the total number of vector registers.

This significant gain is the combined result of (a) the structure of the row-based matrix multiplication algorithm, (b) the hybrid data placement across the vector and the scalar register files, and (c) the efficiency of interleaved loop unrolling shown in Section 3.2.3 that reduces unnecessary name dependencies and allows sharing of index calculations. The benefit of interleaving the execution of the operations of the unrolled inner loop has been quantified separately and compared to the traditional unrolling that can be performed automatically by Clang [71] The results for the same CNN layers and the same inner and outer loop unrolling factors are shown in Figure 3.7. In all cases, the interleaving presented in this chapter achieves 50% higher performance.

Algorithms 3.1-S, 3.2-S, and 3.3-S can all equally benefit from unrolling with interleaving. Figure 3.8 illustrates the execution time of each algorithm for the same CNN layers when the inner loop processes 16 non-zero elements in parallel and the outer loop is unrolled by a factor of 8. Algorithm 3.1-S exhibits the lowest performance, due to the overhead associated with transferring data between vector and scalar registers for each operation. In contrast, Algorithms 3.2-S and 3.3-S, which employ a mixed index and data placement across scalar and vector registers, achieve better performance. Algorithm 3.2-S uses scalar registers to store both the non-zero values and column indices of matrix A, increasing the demand on scalar registers to maintain this data locally. Conversely, Algorithm 3.3-S stores the non-zero elements of A in vector registers. This approach eliminates the need for independent scalar load instructions for

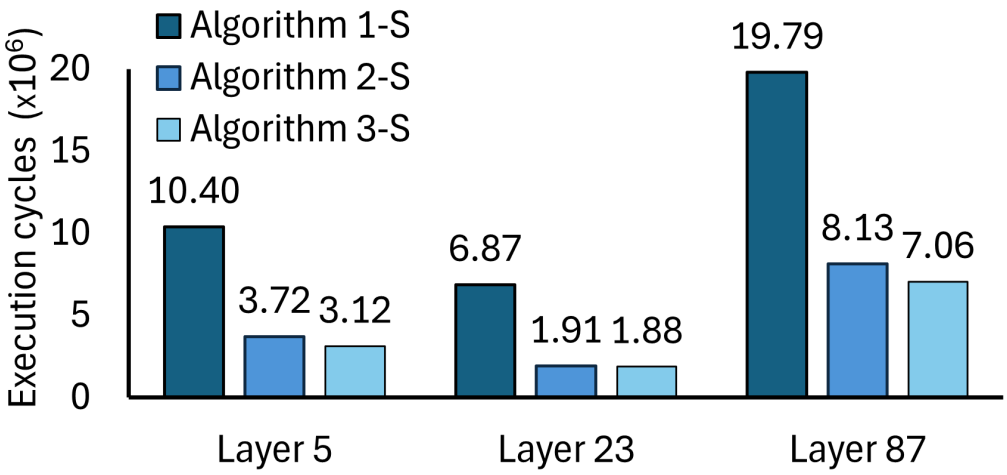


Figure 3.8: Comparison of the unrolled versions of Algorithms 3.1-S, 3.2-S, and 3.3-S, which achieve the best performance in each case for the same outer and inner loop unrolling configuration.

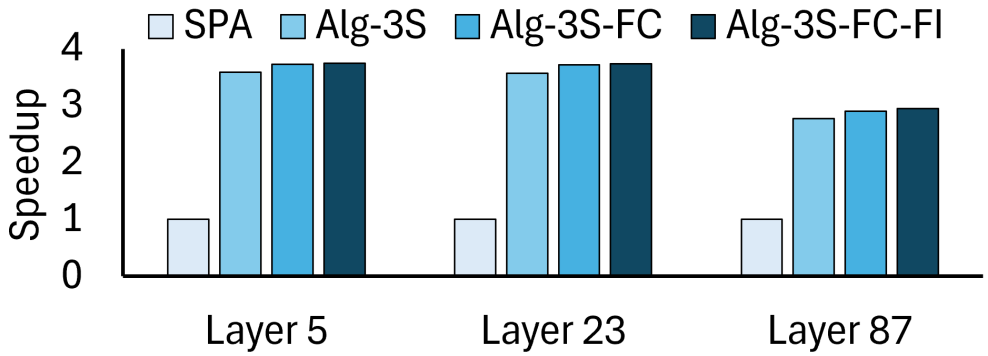


Figure 3.9: Performance analysis of the SPA, Alg-3S, Alg-3S-FC, and Alg-3S-FC-FI implementations, each normalized relative to the SPA implementation.

each non-zero element of A, as required by Algorithm 3.2-S. This change is the reason for the slight superiority of Algorithm 3.3-S relative to Algorithm 3.2-S.

To identify a state-of-the-art Sparse-dense baseline to compare with the implementation of `vindexmac`, we first compare the best configuration identified so far for Alg-3S, with various other alternatives:

- Alg-3S-FC (Full Column): It enhances Alg-3S by leveraging full column indexes to avoid the additional overhead in generating the columns themselves, as required by the structured sparse blocks. This approach is effectively equivalent to a CSR implementation, with the *added simplification* that the number of elements per row is predetermined, as in structured sparsity. These broader column indexes are anticipated to improve runtime, albeit with an increase in storage requirements.
- Alg-3S-FC-FI (Full column with Fast Indexing): The optimized Alg-3S-FC can be further enhanced by incorporating the custom instructions proposed in [91], which integrate/fuse the address generation phase into both scalar and vector load instructions. This acceleration of indexing calculations in hardware is orthogonal to all approaches presented in this chapter and can be equally applied to `vindexmac` extensions as well.

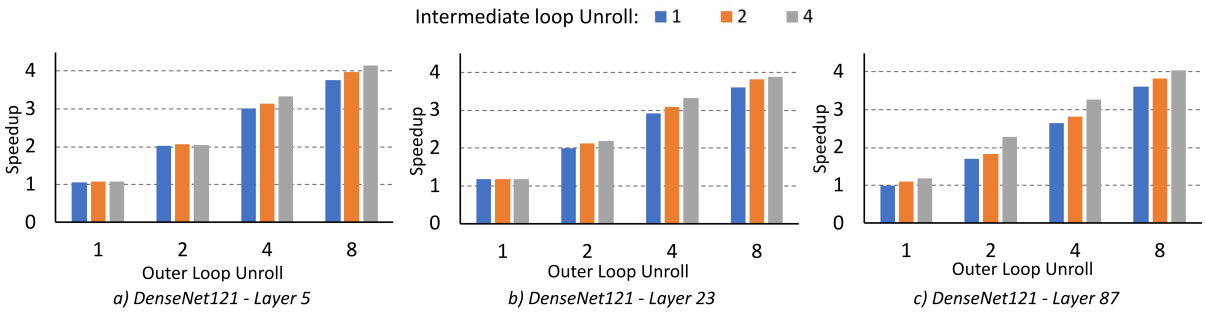


Figure 3.10: The performance of the optimized sparse-dense matrix multiplication, enhanced with the proposed `vindexmac` instruction (Algorithm 3.6). Results are provided for the execution of three CNN layers of Densenet121 [52], assuming a 1:4 sparsity pattern: (a) Layer 5; (b) Layer 23; and (c) Layer 87. The reported cases examine the combined effect of the unrolling of the intermediate and outer loops of Algorithm 3.6. The speedup is reported relative to the performance of the rolled variant.

- SPA: SPA is the most widespread algorithm that handles sparse matrix multiplication when both arrays A and B are substantially sparse [38, 79, 73]. SPA is included for completeness even if its main target is sparse-sparse matrix multiplications with high degrees of sparsity.

As shown in Figure 3.9, both Alg-3S-FC and Alg-3S-FC-FI yield performance improvements, though not substantial over Alg-3S. This is primarily because the indexing arithmetic required by Alg-3S – necessitated by the compact column indexes of the structured sparse format – can be shared. Combined with reduced name dependencies under loop unrolling, this has only a minor effect on performance when executed on an Out-of-Order (OoO) superscalar core. More importantly, however, the utilization of full columns (as in Alg-3S-FC and Alg-3S-FC-FI) results in a significant storage overhead of 14.7%, 26.5%, and 23.5% for the sparse matrix in Layers 5, 23, and 87, respectively, which is an undesired side effect. The limited performance of the SPA implementation is attributed to its use of vector-indexed memory operations to manage non-zero elements. These operations adversely affect the overall performance; they lead to a high number of cache misses, even with a relatively small number of vector instructions.

Given (a) the target of structured sparsity, (b) the smaller column indexes, and (c) the requirement to maintain full RISC-V ISA compliance, we select Alg-3S as the reference baseline, henceforth called 'SpMM' in the experiments. More specifically, we will shortly describe the best setup as SpMM(16,8), referring to an inner-loop unrolling of 16 and an outer-loop unrolling of 8.

It is important to note that if we were to select Alg-3S-FC-FI as the baseline, which employs full columns and the custom instructions proposed by [91], the performance speedups with the proposed methodology would only be marginally smaller (as indicated by the marginal performance improvement of Alg-3S-FC-FI over Alg-3S in Figure 3.9). The speedups achieved by the proposed approach when compared to Alg-3S-FC-FI will be quantified experimentally in the following sub-section.

3.4.3 Identifying the best configuration when using the new `vindexmac` instruction

Sparse-dense matrix multiplication using the new `vindexmac` instruction has been evaluated for various configurations that execute the same three layers of DenseNet121 [52] with 1:4 structured sparsity. The proposed kernel evaluated is the one shown in Algorithm 3.6, which extends the row-based matrix

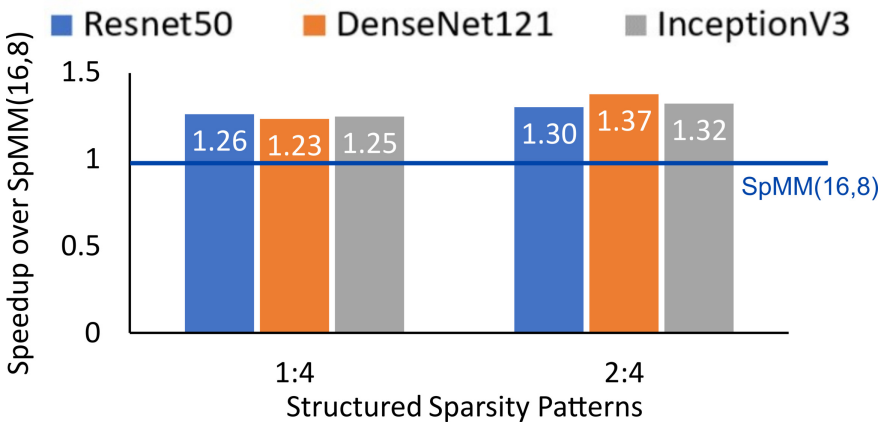


Figure 3.11: The speedup achieved by the proposed approach ('Proposed(8,4)') in the *total* execution times of the three examined CNNs. Results are shown for (a) 1:4, and (b) 2:4 structured sparsity, normalized to 'SpMM(16,8)' of the respective sparsity.

multiplication with a mixed placement of data in the vector and scalar register files and the new instruction that operates on local data and reduces memory traffic. Algorithm 3.6 consists of three distinct loops (inner, intermediate, and outer) that can be arbitrary unrolled. To identify the best configuration, we choose to always fully unroll the inner loop, and then investigate the effect of separately unrolling the other two loops. In all examined cases, the tile of matrix B is composed of L=16 rows that occupy half of the vector register file. The value of L and the sparsity pattern $N{:}M$ determine the number of iterations of the intermediate loop and the effective unrolling of the inner loop.

The achieved speedup for each examined configuration is shown in Figure 3.10. To have a consistent view of the achieved speedup, the results of Figure 3.6 and Figure 3.10 have been normalized to the same basis, i.e., the runtime of the rolled version of Algorithm 3.3-S, which uses instructions from the current definition of the RISC-V ISA. From the speedups reported in Figure 3.10, we can observe that the unrolling of the outer and intermediate loops substantially improves the performance. In all cases, the speedup is maximized when the outer loop unrolling factor is 8 and the intermediate loop has been unrolled by a factor of 4. The outer-loop unrolling corresponds to the number of rows of sparse matrix A that are computed in parallel. Since a tile of matrix B occupies half of the vector register file, only the other half of the vector register file is available to facilitate higher degrees of loop unrolling. Thus, an unroll factor of 8 is the maximum possible, based on the number of available vector registers. An intermediate loop-unrolling factor of 4 corresponds to a full unroll for the examined configuration, i.e., $VectorLength$=16 elements (see Table 3.1), L=16 and $N{:}M$=1:4.

In the rest of the experimental results, we use the best configuration for the proposed approach, as identified in Figure 3.10: intermediate-loop unrolling of 4 and outer-loop unrolling of 8. We define this setup as Proposed(8,4).

3.4.4 Comparisons

The best identified configurations derived for SpMM (in Section 3.4.2) and for the proposed approach (in Section 3.4.3) are hereby compared when executing the three complete CNN models. Figure 3.11 illustrates the achieved speedup of the proposed approach ('Proposed(8,4)') over 'SpMM(16,8)' in the three

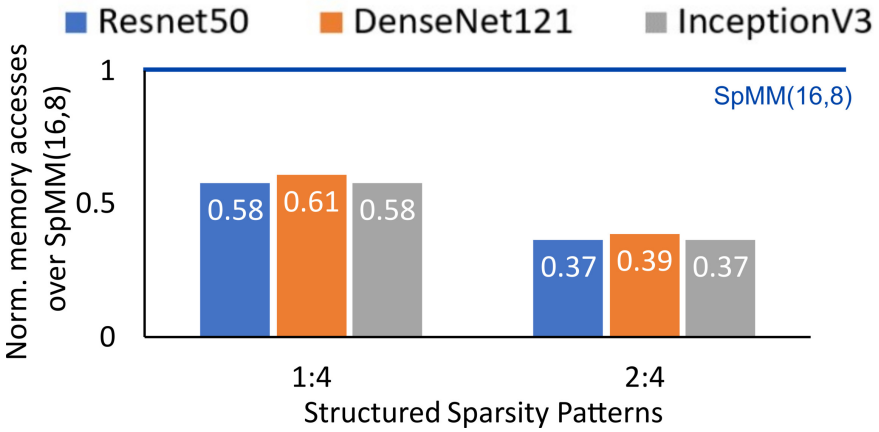


Figure 3.12: The normalized number of *total* memory accesses observed when using the proposed approach ('Proposed(8,4)') in the three examined CNNs, for (a) 1:4, and (b) 2:4 structured sparsity. The normalization is with respect to 'SpMM(16,8)' of the respective sparsity.

evaluated CNNs, assuming 1:4 and 2:4 sparsity patterns. Clearly, the performance of 'Proposed(8,4)' is substantially better in all cases. Across all three CNNs, the average speedup achieved for 1:4 sparsity is 1.25×, which increases to 1.33× for 2:4 sparsity. As expected, the average speedups of the proposed approach when compared against Alg-3S-FC-FI – with the same unroll factors for both the inner and outer loops – are only marginally smaller: 1.20× for 1:4 sparsity and 1.28× for 2:4 sparsity.

It should be stressed that this extra 25%–33% improvement offered by the proposed new `vindexmac` instruction is *extra* speedup, *over and above* the already highly optimized row-based kernel introduced in this paper. The *overall* improvement corresponds to above 4× speedup, as compared to baseline approaches performing sparse-dense matrix multiplication. This result highlights the combined improvement offered by mixed-data placement in both the scalar and vector register files and the reduced memory traffic achieved by the new instruction.

In general, the above-mentioned improvements in performance when using the 'Proposed(8,4)' approach are reaped from (a) the reduction of vector operations per iteration, and (b) the elimination of unnecessary vector loads from memory and their transformation into indirect reads from the vector register file (as facilitated by the `vindexmac` instruction). By pre-loading tiles of matrix B into the vector register file, the proposed approach exploits data locality very effectively, thereby lowering the memory traffic. The results presented in Figure 3.12 quantifies this reduction in total memory accesses when using the proposed `vindexmac` instruction. The presented results are normalized to the number of memory accesses observed with 'SpMM(16,8)' of the respective sparsity. As can be seen, the total memory accesses decrease markedly. For sparsity 1:4, the memory accesses are reduced by 42%, on average, while the average reduction for 2:4 sparsity is 63%.

3.4.5 Performance scaling: Increasing hardware vector length or employing multiple cores

To further explore the scalability of the proposed approach built around the new `vindexmac` instruction, we experiment with different hardware configurations. Specifically, we investigate the effect on

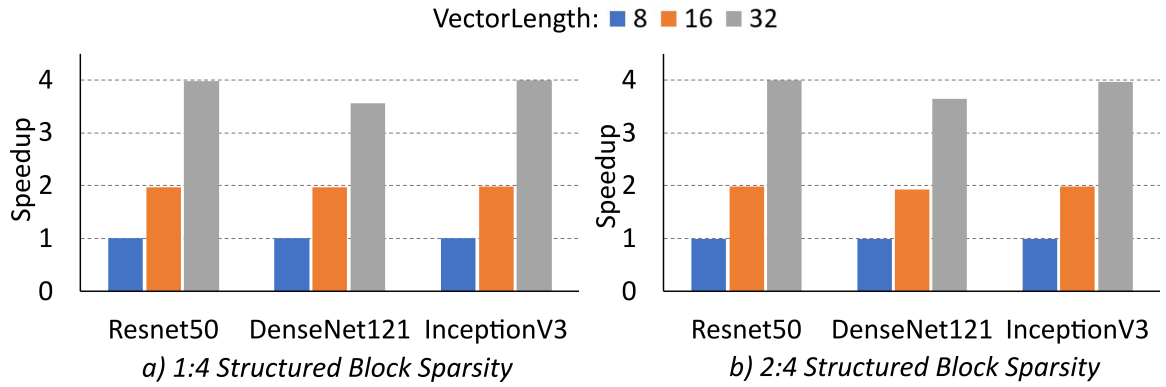


Figure 3.13: The effect of hardware vector length on the proposed approach in the three examined CNNs, for (a) 1:4 and (b) 2:4 structured sparsity. The normalization of the results is with respect to the $VectorLength$=8 configuration of the respective sparsity.

performance when scaling the available parallelism of the vector unit. Recall that the basic configuration (as presented in Table 3.1) utilizes 512b vector registers to operate across 16 parallel lanes, assuming 32-bit words. Hence, in the basic configuration, each vector register has 16 vector elements, i.e., $VectorLength$=16. In addition to the basic hardware configuration, we hereby examine two more configurations: 256b vector registers with 8-lane parallelism ($VectorLength$=8), and 1024b vector registers with 32-lane parallelism ($VectorLength$=32).

The execution time measured for the three CNNs – assuming the same two sparsity patterns – is shown in Figure 3.13. The execution time in each case is normalized to the execution time of the $VectorLength$=8 configuration of the respective sparsity. As expected, when the vector length increases, the reaped speedup increases proportionally. Since the longer vector length is accompanied by more execution lanes, it means that more columns of the output matrix C are being processed in parallel at any given time. This, in turn, leads to significantly lower execution times. As can be seen in Figure 3.13, in most cases, the achieved speedup over $VectorLength$=8 is near-perfect, i.e., speedups approaching 2× with $VectorLength$=16 and speedups approaching 4× with $VectorLength$=32. This desired behavior is aided by the L2 cache, which is big enough in the current processor configuration to accommodate the bigger volume of data as the vector length increases. Naturally, the performance improvement would saturate if the working data size were to approach the L2 cache capacity.

Besides the scaling of the hardware vector length, additional parallelism can be extracted by utilizing a *multicore* setup. In such environment, every CNN layer would be executed across multiple cores. The data assigned to each core corresponds to a vertical segment of VL columns of matrix B and the whole structured-sparse matrix A. This partitioning is shown in Section 3.3.3.

Each core consists of the scalar unit and the decoupled vector unit, as presented in Section 3.4.1. All cores share access to the DDR main memory [133] via a cache-coherent interconnect (specifically, the ARM AMBA Coherent Hub Interface (CHI) [6]) that transfers data at the cache-line level.

The speedup achieved when executing the examined CNNs in a multicore setup, as compared to a single-core implementation, is shown in Figure 3.14. We tested various multicore CPU sizes, doubling the number of cores in each experiment. The obtained results reveal that performance improves significantly with the number of cores. The reaped speedup gains continue up to 8 cores. Beyond 8 cores,

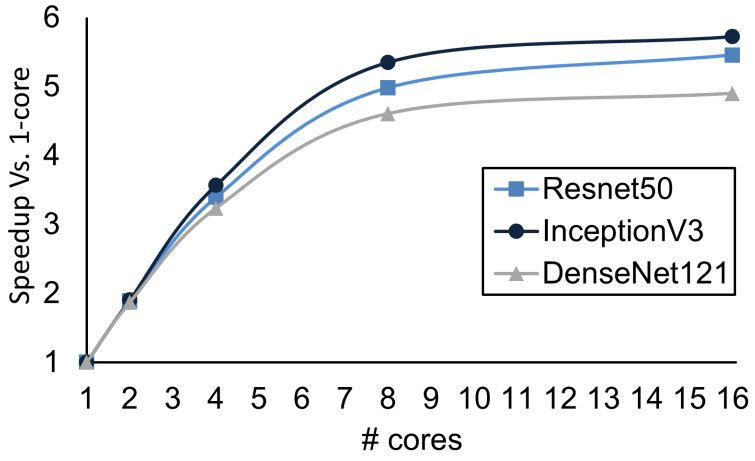


Figure 3.14: The speedup of the proposed approach ('Proposed(8,4)') for CNNs with 1:4 sparsity when executed in a multicore environment, as the number of cores increases. Execution time is normalized to the one achieved by a single core.

the performance saturates. This is attributed to the limitations of the memory sub-system. As more cores request memory access simultaneously, the memory bandwidth is insufficient to handle the escalating demand; i.e., the bandwidth inevitably saturates. This saturation causes contention among cores for memory resources, which limits the algorithm's scalability and prevents further speedup.

This memory bandwidth saturation is also evident in Figure 3.15, which compares the speedup of the proposed approach (Proposed(8,4)) relative to a baseline multicore setup with unmodified ISA (SpMM(16,8)) with increasing numbers of cores. As can be observed, the reaped speedup starts to decrease beyond 8 cores and tends toward 1 (i.e., performance parity) as the cores increase to 16 and beyond. In other words, the proposed approach is most effective in systems where the bottleneck is *computation* and not communication, as is typically the case in multicore setups with *many* cores. The key takeaway is that the proposed new `vindexmac` instruction allows for the use of more power-efficient designs with fewer cores by facilitating better memory bandwidth utilization per core.

3.5 Related Work

The hardware acceleration of efficient matrix multiplication – also involving sparse matrices – has been the focus of extensive research activity within the computer architecture community. Optimizations to sparse matrix multiplication predominantly aim to reduce the number of unnecessary operations involving zero-value elements. As a first step, data storing formats like Compressed Sparse Row (CSR) [12] and Compressed Sparse Column (CSC) [73] help reduce the memory footprint and data movement costs.

Vector processors [33, 21, 142, 20, 101, 94] have been utilized extensively to accelerate *dense* matrix multiplication due to their SIMD capabilities. Within the context of *sparse* data, various vectorized matrix multiplication implementations have also been proposed [38, 12, 79, 73]. The use of custom vector instructions has also been proposed as a means to accelerate sparse computations [91]. These new RISC-V vector data transfer instructions integrate/fuse the memory address generation operation with the standard load operation to reduce energy consumption [91].

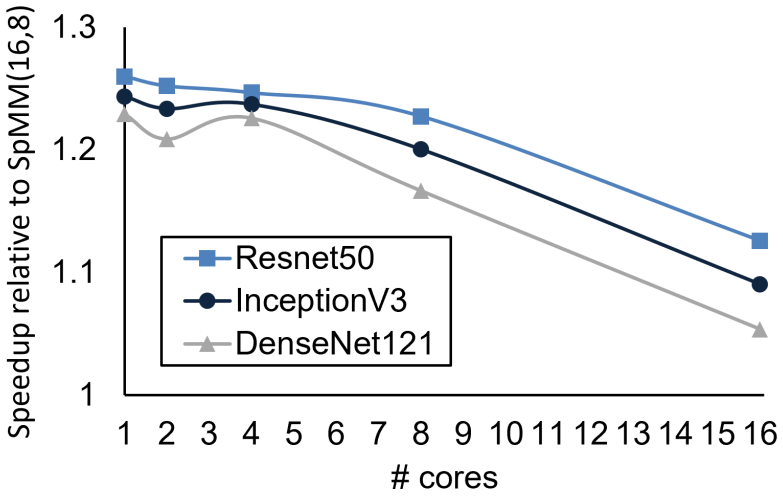


Figure 3.15: The speedup of the proposed approach ('Proposed(8,4)') relative to a baseline multicore setup with unmodified ISA ('SpMM(16,8)') with increasing numbers of cores, for CNNs with 1:4 sparsity.

Recent research has primarily focused on *specialized* vector-like hardware architectures to tackle the challenges involved when processing sparse data access patterns, such as those encountered in scientific computing, AI, and data analytics. The Tensor Marshaling Unit (TMU) [127] architecture utilizes a programmable dataflow engine to efficiently handle sparse tensor operations, thereby decoupling data marshaling from computation. A vector-based hardware implementation that aims to improve sparse computations is the Vector Indexed Architecture (VIA) [102]. VIA leverages a specialized scratchpad memory to improve sparse matrix operation efficiency by optimizing memory traffic and by handling the index-matching process that is central to sparse computations. This approach yields performance gains in sparse matrix-vector multiplication, matrix addition, and matrix multiplication.

Collectively, these architectures represent significant strides in decoupling data access from computation within vector processors. The ultimate goal is to mitigate memory access inefficiencies and enhance parallelism when processing sparse data, which, in turn, can be classified as either unstructured [34], or structured [95, 167]. Alongside the row-wise approach [134, 84, 97, 143] to matrix multiplication, which aligns well with the capabilities of vector processors, structured sparsity within vector processors presents an intriguing area for further exploration.

Outside the vector processing domain, there has been extensive research pertaining to the general challenges involved with the processing of sparse data. The SpChar framework [122] utilizes decision trees to characterize workload patterns across various sparse kernels, like Sparse Matrix-Vector Multiplication (SpMV), Sparse General Matrix-Matrix Multiplication (SpGEMM), and Sparse Matrix Addition (SpADD) on ARM architectures. This approach provides insights into the influence on performance of key architectural characteristics, such as cache size, memory latency, and branch prediction. Gupta et al. [41] highlight the unique requirements of DNN-based recommendation systems. Unlike traditional DNN workloads, recommendation systems are characterized by irregular memory access patterns – due to sparse data – and high memory capacity needs, which are the consequence of utilizing large embed-

ding tables. Finally, FPGA-based accelerators [99] have shown promise in sparse computing, due to their balance of power efficiency and performance, despite potential bandwidth limitations.

3.6 Conclusion

Vector processors can efficiently handle the abundant data-level parallelism available in modern ML applications. The scalability of ML models calls for appropriate model pruning that reduces their memory footprint and makes them amenable to applications at the edge. Pruning can benefit various neural network layers, such as fully connected, convolutional, recurrent, and normalization. Furthermore, pruning can benefit attention mechanisms. In this context, we aimed to seamlessly integrate the simplicity of structured sparsity with mainstream vector architectures enhanced with a proposed new instruction.

Initially, we reorganized and optimized the state-of-the-art row-based matrix multiplication algorithm for structured-sparse data for the current definition of the RISC-V ISA. We adopted a hybrid placement for the non-zero data elements and their corresponding column indexes across the vector and the scalar register files. This approach reduced register name dependencies and simplified loop unrolling, thereby significantly improving the runtime.

Subsequently, this optimized approach was further enhanced with the proposed new `vindexmac` instruction, which operates on local data that is pre-loaded into the vector register file. The use of this new instruction reduces the total number of instructions executed and eliminates unnecessary vector loads. Most importantly, the new instruction can be implemented with negligible hardware cost.

Chapter 4

Vectorized FlashAttention

Attention is one of the most demanding dense kernels in modern transformer models and a major source of both computation and memory traffic. As sequence lengths grow, the cost of attention increases rapidly, making it a central performance bottleneck in transformer inference and training. This makes efficient attention execution an important target for vector architectures, especially when the goal is to retain high throughput without the hardware cost of more specialized accelerators.

Attention works by assigning different levels of importance to different parts of the input, depending on how useful they are for the current computation. Instead of treating all parts equally, the model computes relevance scores and uses them to emphasize important information while suppressing less useful context. However, this ability comes with high computational cost. The standard attention mechanism has quadratic complexity, meaning that both execution time and memory usage grow rapidly with sequence length [13]. This creates a major performance bottleneck and limits the efficient use of transformers on long inputs.

To address this challenge, several approaches have been proposed to reduce the cost of attention. These include sparse [23], linear [65], and low-rank [158] attention methods, which aim to improve efficiency while preserving accuracy. In addition, specialized hardware accelerators [43, 56, 5, 157] have been developed to speed up key parts of the attention computation, such as matrix operations [161] and softmax evaluation [137, 157].

FlashAttention [29, 28, 110], originally developed for GPUs, addresses the computational bottlenecks of standard attention efficiently. By performing the attention computation in tiles instead of materializing the full attention matrix, it reduces memory requirements and improves both memory efficiency and computational performance.

This chapter adapts FlashAttention to the RVV ISA by developing a fully vectorized implementation that preserves the algorithmic advantages of tiled attention while remaining compatible with practical vector processors. The objective is not only to accelerate attention, but to do so within the constraints of a standard vector ISA, without relying on custom hardware support.

A key challenge in this setting is the softmax computation embedded in attention. In addition to the matrix multiplications required for score calculation and weighted accumulation, attention also requires maximum tracking, exponentiation, accumulation, and normalization. These stages are reduction-heavy and involve non-linear operations that are not directly supported by the baseline RVV ISA. For this reason,

this chapter also introduces a simplified method for evaluating the exponential function using standard floating-point arithmetic, tailored specifically to the dataflow of FlashAttention and suitable for efficient vector execution without requiring custom hardware support.

The chapter further develops a tiling strategy that preserves the core idea of FlashAttention, namely processing attention in small, memory-efficient blocks, while adapting it to vector-length-agnostic execution and the register constraints of practical RVV processors. In this way, the proposed approach remains fully compatible with the standard RVV programming model while improving data reuse, reducing memory traffic, and exposing the regular data-parallel structure required for efficient vector execution.

Overall, the chapter makes three main contributions. First, it presents a fully vectorized implementation of the FlashAttention algorithm, expressed entirely through vector instructions and without relying on scalar operations in the core computation. Second, it introduces a simplified method for vector exponentiation using standard floating-point arithmetic, avoiding custom hardware instructions or ISA extensions while preserving the accuracy of representative LLM applications. Third, it develops a vector-aware tiling strategy that adapts FlashAttention to the constraints of RVV processors, improving locality and fitting naturally within the vector register file.

4.1 Baseline Attention and FlashAttention algorithm

The attention mechanism allows models to assign varying levels of importance to elements within an input sequence, making it possible to capture long-range dependencies and contextual associations. It operates through the interaction of query, key, and value vectors, all of which are generated from the input embeddings [155]. In practice, the attention mechanism operates across multiple heads in parallel, known as multi-head attention, allowing the model to capture complex relationships more effectively [155]. In the following, without loss of generality, we will limit our discussion to a single-head attention.

4.1.1 Baseline Attention with Lazy Softmax Division

Given a query vector $\vec{q}$ and sets of key and value vectors defined as $K = \vec{k}_1, \ldots, \vec{k}_N$ and $V = \vec{v}_1, \ldots, \vec{v}_N$, the attention mechanism is computed as follows:

$$s_i = \text{dot}(\vec{q}, \vec{k}_i) \qquad f_i = \frac{e^{s_i}}{\sum_j e^{s_j}} \qquad \text{Attn}(\vec{q}, \text{K}, \text{V}) = \sum_i f_i \vec{v}_i \tag{4.1}$$

Here, each attention score s_i measures the similarity between the query vector and the ith key vector through a dot product. These scores are then passed through a softmax function to determine the relevance of each token to the query. This step involves computing the exponential of each score and normalizing by the sum of all exponentials, resulting in a probability distribution over the input tokens. The final attention output is obtained as a weighted sum of the value vectors, where each weight reflects the relative importance of the corresponding token.

However, computing exponentials of large scores can cause numerical instability, potentially leading to overflow. To address this, a numerically stable variant known as the safe softmax is used. This method subtracts the maximum score from all individual scores before exponentiation, preserving the behavior

Algorithm 4.1 Attention with lazy softmax division

for each query $\vec{q}$ **do**
  for $i = 1 : N$ **do**
    $s_i \leftarrow \text{dot}(\vec{q}, \vec{k}_i)$
    $m_i \leftarrow \max(m_{i-1}, s_i)$
  end for
  $\ell_0 \leftarrow 0$
  for $i = 1 : N$ **do**
    $\vec{o}_i \leftarrow \vec{o}_{i-1} + e^{s_i - m_N} \cdot \vec{v}_i$
    $\ell_i \leftarrow \ell_{i-1} + e^{s_i - m_N}$
  end for
  $\text{attn}(\vec{q}, K, V) \leftarrow \vec{o}_N / \ell_N$
end for

of softmax while preventing overflow:

$$f_i = \frac{e^{s_i - \max}}{\sum_j e^{s_j - \max}} \quad \text{(safe softmax)} \tag{4.2}$$

Instead of performing exponentiation, normalization, and weighted summation as separate steps, lazy softmax architectures [56, 53] take a different approach. They accumulate both the weighted sum of the value vectors and the sum of exponentials in parallel, delaying the division step until the end:

$$s_i = \text{dot}(\vec{q}, \vec{k}_i) \quad f_i = e^{s_i - \max} \quad \text{Attn}(\vec{q}, \text{K}, \text{V}) = \frac{\sum_i f_i \vec{v}_i}{\sum_j e^{s_j - \max}} \tag{4.3}$$

Algorithm 4.1 outlines the computation of attention using the lazy softmax approach. The process begins by calculating the dot product between the query and each key vector, while simultaneously tracking the maximum score for numerical stability. Then, the output vector $\vec{o}_i$ is built incrementally by summing the products of each value vector and its corresponding exponentiated, shifted score. At the same time, the running total of exponentials, denoted as ℓ_i, is updated. The final attention result is produced by dividing the accumulated weighted sum by the total sum of exponentials ℓ_N.

The attention computation in Algorithm 4.1 has a major bottleneck, as output calculation can only start after determining the maximum attention score, limiting efficiency for large sequence lengths. Sequence length N represents the context size. When a model like a Transformer interprets a word, it considers the surrounding words to determine its meaning. The sequence length specifies how many words the model can look at simultaneously.

FlashAttention [29], inspired by online softmax computation [90], addresses this by merging the two inner loops of Algorithm 4.1 into a single loop that computes all necessary variables online.

4.1.2 FlashAttention Algorithm

FlashAttention [29] transforms attention computation into an online process. The main difference between FlashAttention and the baseline attention mechanism (Algorithm 4.1) is that FlashAttention performs all required computations within a single inner loop. This eliminates the need to precompute the

Algorithm 4.2 FlashAttention with delayed softmax division

1: **for each** query $\vec{q}$ **do**
2: **for** $i = 1 : N$ **do**
3: $s_i \leftarrow \text{dot}(\vec{q}, \vec{k}_i)$
4: $m_i \leftarrow \max(m_{i-1}, s_i)$
5: $\ell_i \leftarrow \ell_{i-1} e^{m_{i-1}-m_i} + e^{s_i-m_i}$
6: $\vec{o}_i \leftarrow \vec{o}_{i-1} e^{m_{i-1}-m_i} + e^{s_i-m_i}\vec{v}_i$
7: **end for**
8: $\text{attn}(\vec{q}, K, V) \leftarrow \vec{o}_N/\ell_N$
9: **end for**

Algorithm 4.3 FlashAttention applied at blocks of key matrix

1: **for each** query $\vec{q}$ **do**
2: **for** $i = b : b : N$ **do**
3: $\vec{s} \leftarrow \vec{q} \cdot K^T[:,\ (i-b) : i-1]$
4: $m_i \leftarrow \max(m_{i-b}, \vec{s_i})$
5: $\ell_i \leftarrow \ell_{i-b} e^{m_{i-b}-m_i} + \sum_{j=0}^{b-1} e^{s_{i-j}-m_i}$
6: $\vec{o}_i \leftarrow \vec{o}_{i-b} e^{m_{i-b}-m_i} + \sum_{j=0}^{b-1} e^{s_{i-j}-m_i} V[i-j-1,\ :]$
7: **end for**
8: $\text{attn}(\vec{q}, K, V) \leftarrow \vec{o}_N/\ell_N$
9: **end for**

maximum attention score and avoids storing alignment scores in memory [81]. In effect, it addresses the scalability limitations of Alg.4.1 as sequence length increases.

The FlashAttention kernel is presented in Algorithm 4.2 following a vector-oriented description of the corresponding algorithm. Specifically, Algorithm 4.2 refers to the improved version of FlashAttention [28], *aka*, FlashAttention-2. The next version, i.e., FlashAttention-3 [123] follows exactly the same algorithmic structure and interleaves block□wise GEMM and softmax computation that is specific for GPU implementations. In each iteration of FlashAttention, the dot product between the query vector and the key vectors produces a similarity score s_i. The maximum attention score up to the current step m_i, is also identified. The multiplication by $e^{m_{i-1}-m_i}$ in the computation of ℓ_i in line 5 of Algorithm 4.2, adjusts the previously accumulated sum of exponentiated scores to the new maximum attention score. In line 6, The output vector o_i is updated by adding the new value vector V, weighted by its softmax importance, to the previous output vector o_{i-1}, which is similarly adjusted by the scaling factor $e^{m_{i-1}-m_i}$. Finally, in line 8, the attention of a query vector is computed by dividing the output o_N by the sum of all exponentiated attention scores ℓ_N.

As shown in Algorithm 4.2, parallelization can be applied either to the outer or the inner loop. Parallelizing the outer loop allows for processing multiple queries for the same column of K. However, in this case, all state needed per query vector such the maximum attention score m_i and the running sum of exponents ℓ_i should be also stored. Parallelizing the inner loop matches better to the vectorization of FlashAttention that is the focus of this chapter.

In this case, instead of computing a scalar alignment score s_i, we compute a vector of alignment scores $\vec{s}$ by performing a dot product across multiple columns of K and the same query vector $\vec{q}$. In this way, the maximum score identification, the computation of sum of exponents, and output accumulation

should also adapt to operate on this vector of attention scores. This adaptation can be easily derived by unrolling the recursive computation of m_i, ℓ_i and o_i. For instance, by unrolling once the running sum of exponents ℓ_i we get that:

$$\begin{aligned}\ell_i &= \ell_{i-1}e^{m_{i-1}-m_i} + e^{s_i-m_i} \\ &= \left(\ell_{i-2}e^{m_{i-2}-m_{i-1}} + e^{s_{i-1}-m_{i-1}}\right)e^{m_{i-1}-m_i} + e^{s_i-m_i} \\ &= \ell_{i-2}e^{m_{i-2}-m_i} + e^{s_{i-1}-m_i} + e^{s_i-m_i}\end{aligned} \tag{4.4}$$

Repeating the same procedure b times leads to the following lookahead form for computing the sum of exponentials for b consecutive attention scores.

$$\begin{aligned}\ell_i &= \ell_{i-b}e^{m_{i-b}-m_i} + e^{s_{i-b+1}-m_i} + \ldots + e^{s_i-m_i} \\ &= \ell_{i-b}e^{m_{i-b}-m_i} + \sum_{j=0}^{b-1} e^{s_{i-j}-m_i}\end{aligned} \tag{4.5}$$

From (4.5), it can be easily seen that ℓ_i can be computed by the sum of exponentials of b attention scores $s_i, s_{i-1}, \ldots s_{i-b}$ reduced by the current maximum score m_i. Their sum is then accumulated to the adjusted ℓ_{i-b} sum. This adjustment depends only on the current maximum m_i and the previous block maximum m_{i-b}. Likewise, the output accumulation $\vec{o}_i$ can be performed at the vector level including the multiplication with the value vectors and without requiring extra intermediate adjustments to the maximum value. This vector-level lookahead mechanism in the FlashAttention algorithm is illustrated in Algorithm 4.3.

The application of blocking for the query, key, and value matrices, introduced in the original FlashAttention proposals [29, 28], will be examined in Section 4.3 in the context of the vectorized implementation of FlashAttention.

4.2 Vectorization of FlashAttention

The proposed vectorization of FlashAttention using the RISCV ISA follows the block-level execution described in Algorithm 4.3.

The execution of FlashAttention of one query vector is split in five parts: In the first part, the vector of attention scores $\vec{s}$ is computed using a vector×matrix row-based vectorized multiplication. The vector corresponds to the query vector and the matrix is formed by b consecutive columns of the Key matrix. In the second part, the maximum attention in $\vec{s}$ is identified in parallel and the corresponding exponentials are computed in a fully-vectorized form using $\vec{s}$ and the current and the previous maximum attention scores. The third part refers to the computation of the sum of exponents ℓ_i the output vector $\vec{o}_i$. The fourth part corrects the maximum values for the previous sum of exponent ℓ_{i-b} and the output vector o_{i-b} and the last part computes for the current query vector the attention vector by dividing the accumulated output with the sum of exponents of all attention scores.

The execution of each part using vectorized operations that correspond closely to typical RISCV Vector ISA is shown in Algorithm 4.4. In the presented algorithm we assume that the vector processor

Algorithm 4.4 Vectorized FlashAttention for RISCV Vector ISA

1: **for each** query $\vec{q}$ **do**
2: vload v_q, $\vec{q}$
3: **for** $i = b : b : N$ **do**
4: $v_s \leftarrow 0$
5: **for** $j = 0 : d - 1$ **do**
6: vload v_k, $K^T[j,\ (i-b) :\ i-1]$
7: vrgather v_a, v_q, j ▷ Broadcast jth element of $\vec{q}$
8: vmacc v_s, v_k, v_a ▷ Multiply-Accumulate
9: **end for**
10: vredmax v_{max}, v_s, v_{oldmax} ▷ Obtain current maximum value
11: vrgather v_{max}, v_{max}, 0
12: v_b = vexp(v_{oldmax} - v_{max}) ▷ Calculate $e^{m_{i-b}-m_i}$
13: v_c = vexp(v_s - v_{max}) ▷ Calculate $e^{\vec{s}-m_i}$
14: vredsum v_{sum}, v_c ▷ Calculate current sum of exponents
15: vrgather v_{sum}, v_{sum}, 0 ▷ Broadcast sum of exponents
16: vmsneq v_{mask}, v_{oldmax}, v_{max} ▷ Create mask to perform lookahead
17: **for** $j = 0 : b - 1$ **do**
18: vload v_v, $V[i-j-1,\ :]$ ▷ Load row of V
19: vrgather v_d, v_c, $(b-1-j)$
20: vmacc v_{out}, v_v, v_d
21: **end for**
22: vmul v_{oldsum}, v_{oldsum}, v_b, v_{mask} ▷ Masked multiplication
23: vadd v_{sum}, v_{oldsum}, v_{sum} ▷ Calculate l_i
24: vmv v_{oldsum}, v_{sum} ▷ Move l_{i-b} to l_i
25: vmul v_{oldout}, v_{oldout}, v_b, v_{mask} ▷ Masked multiplication
26: vadd v_{out}, v_{oldout}, v_{out} ▷ Calculate o_i
27: vmv v_{oldmax}, v_{max} ▷ Move m_{i-b} to m_i
28: **end for**
29: vdiv v_{out}, v_{out}, v_{sum} ▷ Perform $\vec{o_N}/\ell_N$
30: vstore v_{out}, $\vec{o}_N$ ▷ Store $\vec{o}_N$
31: **end for**

operates on an arbitrary large vector length that matches the head dimension of the attention model, denoted as d. The more realistic scenario, where vector length is smaller than the head dimension is discussed in Section 4.3.

To compute the vector of alignment scores, we utilize the row-wise matrix multiplication [134], which is highly efficient for vector-matrix multiplication. This operation requires each element of the vector v_q to be multiplied with the corresponding row vector v_k from the transposed key matrix K^T. To accomplish this, during each iteration of the loop in line 5 of Algorithm 4.4, we must isolate individual elements of v_q. For this purpose, we employ the `vrgather` instruction, which enables indexing into v_q and broadcasting a selected element across all elements of the vector v_a. The alignment score vector is then computed using a multiply-and-accumulate instruction, as shown in line 8 of Algorithm 4.4, completing the first phase of FlashAttention's execution for a single query vector. Lines 5–9 in Algorithm 4.4 correspond to line 3 of Algorithm 4.3.

It is worth noting that, instead of the row-wise approach, a dot-product-based method could have been

used. However, such an approach would generate multiple scalar values that would need to be rearranged into a single vector. This process would require several additional shift and merge operations, resulting in a greater number of inefficient vector instructions. In contrast, the row-wise multiplication algorithm directly places each computed value in its correct position within the result vector, thus minimizing the number of vector instructions required and improving overall efficiency. In architectures where vrgather is expensive, the same operation could instead be implemented using scalar–vector multiply instructions, at the cost of additional data movement between scalar and vector register files.

In the second part, covering lines 10–13, the maximum value is computed across all elements of the vector vs and the previously calculated maximum that is stored in vector v$oldmax$. More precisely, line 10 of Algorithm 4.4 corresponds to line 4 of Algorithm 4.3. This is achieved using the `vredmax` instruction, which, in the RISC-V ISA, stores the resulting maximum in the first element of the vector vmax. To propagate this value across the entire vector, it is broadcast to all elements of vmax using the `vrgather` instruction, as shown in line 11 of Algorithm 4.4.

Moreover, since the RISC-V ISA lacks direct support for vector exponentiation instructions, the operations in lines 12 and 13 of Algorithm 4.4, which compute the terms $e^{m_{i-b}-m_i}$ and $e^{\vec{s}-m_i}$, respectively, are conceptually referred to as vexp for clarity. The implementation details of these operations will be discussed in the following subsection.

In the third part, the sum of exponents is computed using the `vredsum` instruction in line 14 of Algorithm 4.4. Then, in lines 17–21, the current output vector $\vec{o}_i$ is computed and stored in the vector v_{out} by applying the row-wise multiplication algorithm between the vector of exponentiated attention scores and the corresponding block of the value matrix. By the end of this part, v_{sum} contains the term $\sum_j e^{s_{i-j}-m_i}$, which is required in line 5 of Algorithm 4.3, and v_{out} holds the term $\sum_j e^{s_{i-j}-m_i} V[i-j-1,\ :]$, as defined in line 6 of Algorithm 4.3.

To complete the computation described in lines 5 and 6 of Algorithm 4.3, the previously computed sum of exponents ℓ_{i-b} and output vector o_{i-b} must be added to the newly computed terms, after applying a correction factor $e^{m_{i-b}-m_i}$. To simplify this correction, we use a conditional assignment. The correction is necessary whenever the current maximum value m_i differs from the previous maximum m_{i-b}. The result of this comparison is stored in line 16 in a vector mask. This mask controls the subsequent masked multiplication instructions that adjust the vectors v_{oldsum} and v_{oldout} based on the current maximum value, as shown in lines 22 and 25 of Algorithm 4.4, respectively. The corrected vectors are subsequently added to the current sum of exponents and the output vector $\vec{o}_i$, as illustrated in lines 23 and 26 of Algorithm 4.4.

In the RISC-V Vector ISA, masks are held in a dedicated mask register (typically `v0`), and predicated (masked) vector operations implicitly consume this register. While this design introduces a small but recurring bookkeeping overhead, it enables predication without control-flow divergence.

Finally, in the last part, the output vector $\vec{o}_i$ is normalized by dividing it by the final sum of exponents ℓ_i, as shown in line 29 of Algorithm 4.4.

4.2.1 Exponent approximation

To complete the FlashAttention kernel we need to approximate in lines 12 and 13 of Algorithm 4.4 the exponentials of the two maximum differences attention scores. The resulting exponentiated scores

should return in the same floatint-point format as the attention score differences. In RISC-V Vector ISA there is no direct instruction to compute exponentials in vector form. This absence means that software implementations must rely on approximation techniques to perform efficiently this computation at the vector level.

In the case of vectorized FlashAttention, a generic exponential function approximator is unnecessary and a custom approximation suffices, as we only need to compute attention score differences that are zero or negative. This stems from the fact that attention scores are always offset by the maximum score. Therefore, the first step is to appropriately quantize the attention score differences. Then, using the approach proposed by Blin [16], we compute the exponential of the quantized differences in a single step and directly convert the result into floating-point format.

Quantization of attention score differences

In general, converting a floating-point number to its fixed-point representation requires many integer bits to accommodate the wide range of floating-point arithmetic [69]. However, in this case, we can safely clip the attention score differences before quantization to a much smaller dynamic range, as it will only be used for the computation of e^x for negative values of x. In this range, e^x quickly converges to 0, even for small negative x values, i.e., $e^{-15} = 3.1 \cdot 10^{-7}$. Therefore, in this chapter, attention score difference Δ is clipped to [-15, 0]. Effectively, Δ is going to be multiplied later by a constant value that would move the range of the result to [-21.64, 0].

For quantizing the clipped attention score differences, we use 32-bit precision for simplicity, aligning with the processor word width, although fewer bits would have been sufficient. The quantized 32-bit fixed-point numbers are assumed to use 9 bits for the signed integer part and 23 bits for the fractional part. To do this, we employ scaling quantization of the clipped attention score differences in range $[-a, a)$ using $b = 32$ bits. The selected 9-bit of the signed integer part covers the range $[-2^8, 2^8)$ which implies $a = 2^8$. Overall, each attention score difference Δ is scaled by s and rounded. The scaling factor is defined as:

$$s = \frac{2^{b-1} - 1}{a}. \tag{4.6}$$

The quantized fixed-point value is then given by:

$$x_f = \text{int}(s \cdot \Delta). \tag{4.7}$$

Exponential of quantized attention score differences

To simplify computation, it is preferable to work with powers of two rather than natural exponents. Since $e^x = 2^{x \log_2 e}$, our goal is to compute $2^{x_f \log_2 e}$, where x_f is the quantized attention score difference computed in (4.7), and obtain the result directly in floating-point format without any additional conversion step.

Leaving aside for now the multiplication of x_f by $\log_2 e$, we note that since x_f is a fixed-point number with an integer part I_x and a fractional part F_x, we can express 2^{x_f} as follows:

$$2^{x_f} = 2^{I_x + F_x} = 2^{I_x} \cdot 2^{F_x} \tag{4.8}$$

Algorithm 4.5 Vectorized exponentiation of attention score differences in FlashAttention

1: vmfle $v_{mask},\ v_{exp},\ -15$	▷ Create mask for clipping
2: vmul $v_{exp},\ s \cdot \log_2(e)$	▷ Floating point multiplication
3: v_{exp} = int(v_{exp})	▷ Integer conversion
4: vadd $v_{exp},\ v_{exp},\ B \cdot 2^{W_M}$	▷ Integer addition
5: vadd $v_{exp},\ v_{zero},\ e^{-15}, v_{mask}$	▷ Replace the clipped values of x with e^{-15}

Since F_x, being a fractional part, lies between 0 and 1 by construction, we can safely approximate 2^{F_x} by $1 + F_x$. Substituting this approximation into (4.8), we obtain:

$$2^{x_f} \approx 2^{I_x} \cdot (1 + F_x) \tag{4.9}$$

With this approximation, we observe that the result of 2^{x_f} takes the form of a floating-point number, where I_x corresponds to the unbiased exponent and F_x to the mantissa.

To fit to the floating-point representation the exponent bias must be restored affecting only the integer part I_x of x_f and leaving the fraction part F_x unchanged. As noted by Blin [16], this can be achieved by adding to x_f bias B scaled by the fraction (mantissa) bit width W_M:

$$2^{x_f} \approx 2^{I_x - B} \cdot (1 + F_x) = x_f + B \cdot 2^{W_M} \tag{4.10}$$

This addition effectively incorporates the exponent bias back into I_x, thereby reconstructing the correct binary representation of the floating-point value of 2^x_f.

This implementation of vectorized exponentiation is presented in Algorithm 4.5. Let $\mathbf{v}_{\text{exp}}$ denote the input vector containing the values that need to be exponentiated. The computation begins by identifying elements in $\mathbf{v}_{\text{exp}}$ that are less than -15. For these elements, a mask is generated to indicate which values require clipping.

Following the mask generation, a floating-point multiplication is performed using the precomputed constant $s \cdot \log_2(e)$, effectively computing the product $s \cdot x_{\text{exp}}$. This result is then converted to an integer representation, as shown in line 2 of Algorithm 4.5. An integer addition is subsequently performed between the converted result and the constant $B \cdot 2^{W_M}$.

Finally, elements of $\mathbf{v}_{\text{exp}}$ that were marked for clipping are replaced. This is achieved via a masked vector operation that conditionally assigns the value e^{-15} to the corresponding positions, using a zero-initialized vector $\mathbf{v}_{\text{zero}}$ to facilitate the update. For the elements not requiring clipping, the addition will not have an effect due to masking. The final output vector approximates the exponential function e^x for the attention score differences in the context of FlashAttention.

The five operations described in Algorithm 4.5, i.e., masking operation, floating-point multiplication, integer conversion, and addition, are all supported by standard RISC-V vector instructions and allows us to compute exponential functions as needed in the context of Flashattention with low instruction count and without relying on any custom ISA extensions.

4.3 Tiling in Vectorized FlashAttention

Since a vector engine is equipped with a finite number of arithmetic logic units (ALUs) and can operate on only a limited amount of data concurrently, the size of the input arrays often exceeds the level of parallelism the hardware can natively support. This mismatch between data volume and computational capacity necessitates the use of additional techniques to manage and optimize performance. One such method is tiling, which partitions the data into smaller, manageable blocks that fit within the engine's parallel processing capabilities. By processing these tiles sequentially or in batches, the system ensures that the full attention output is computed correctly and efficiently.

4.3.1 Handling arbitrary head dimensions

The head dimension in attention refers to the size of the query, key, and value vectors used within each attention head. It determines how much information each head can capture independently, and the full model dimension is typically split across multiple heads to allow parallel attention to different representation subspaces.

The head dimension, denoted by d does not necessarily match the vector length supported by the underlying vector registers. In many cases, d exceeds the available hardware parallelism, which necessitates the application of tiling techniques to maintain computational efficiency. As shown in Algorithm 4.4, the two intermediate loops depend on the parameters d and b. While the parameter d is fixed by the attention mechanism and cannot be altered, the blocking parameter b is configurable and can be set arbitrarily. In this regard, b is set equal to the vector length (*vl*), thereby aligning the blocking strategy with the parallel processing capabilities of the target vector architecture.

When head dimension d is larger that the maximum supported vector length we need to introduce tiling in the vectorized FlashAttention. The tiled form of vectorized FlashAttention is presented in Algorithm 4.6, with the modifications relative to baseline vectorized FlashAttention (Algorithm 4.4) highlighted in magenta. Since head dimension exceed maximum vector length, the vectors loaded during computation do not encompass all elements within a full row of the Q, K, and V matrices. Instead, only a portion of each row is loaded in a given iteration, and the corresponding partial computation is performed. Subsequent iterations then load incrementally the remaining segments of each row, as shown in lines 5, 7 and 21 of Algorithm 4.6. Consequently, all intermediate results must be temporarily stored and subsequently reloaded to ensure correctness. Specifically, intermediate load and store operations involving the output vector occur at lines 25 and 28 of Algorithm 4.6, as well as at lines 36 and 38, in order to correctly compute the division of the output vector $\vec{o}_N$ by the sum of exponents ℓ_N. It is important to note that if the dimension d is only slightly larger than the vector length, the loops associated with the head dimension can be unrolled by utilizing additional vector registers. This approach reduces or eliminates the need to store partial results, thereby avoiding these specific loops.

4.3.2 Operating in parallel on multiple queries

Algorithm 4.6 adopts a dataflow strategy that processes the entire K and V matrices for each query of matrix Q. Although this method yields one row of the attention output at a time, an alternative approach

Algorithm 4.6 Vectorized FlashAttention-2 with d greater than vector length

for each query $\vec{q}$ **do**
  for $i = vl : vl : N$ **do**
    $v_s \leftarrow 0$
    for $h = 0 : d/vl - 1$ **do**
      vload v_q, $\vec{q}\,[h \cdot vl : \ (h+1) \cdot vl - 1]$
      for $j = 0 : vl - 1$ **do**
        vload v_k, $K^T[h \cdot vl + j, \ (i - vl) : \ i - 1]$
        vrgather v_a, v_q, j ▷ Broadcast jth element of $\vec{q}$
        vmacc v_s, v_k, v_a ▷ Multiply-Accumulate
      end for
    end for
    vredmax v_{max}, v_s, v_{oldmax} ▷ Obtain current maximum value
    vrgather v_{max}, v_{max}, 0
    v_b = vexp(v_{oldmax} - v_{max}) ▷ Calculate $e^{m_{i-b}-m_i}$
    v_c = vexp(v_s - v_{max}) ▷ Calculate $e^{\vec{s}-m_i}$
    vredsum v_{sum}, v_c ▷ Calculate current sum of exponents
    vrgather v_{sum}, v_{sum}, 0 ▷ Broadcast sum of exponents
    vmsneq v_{mask}, v_{oldmax}, v_{max} ▷ Create mask to perform lookahead
    for $h = 0 : d/vl - 1$ **do**
      for $j = 0 : vl - 1$ **do**
        vload v_v, $V[i - j - 1, \ h \cdot vl : \ (h+1) \cdot vl - 1]$ ▷ Load row of V
        vrgather v_d, v_c, $(vl - 1 - j)$
        vmacc v_{out}, v_v, v_d
      end for
      vload v_{oldout}, $\vec{o}_N\ [h \cdot vl : \ (h+1) \cdot vl - 1]$
      vmul v_{oldout}, v_{oldout}, v_b, v_{mask} ▷ Masked multiplication
      vadd v_{out}, v_{oldout}, v_{out} ▷ Calculate o_i
      vstore v_{out}, $\vec{o}_N\ [h \cdot vl : \ (h+1) \cdot vl - 1]$
    end for
    vmul v_{oldsum}, v_{oldsum}, v_b, v_{mask} ▷ Masked multiplication
    vadd v_{sum}, v_{oldsum}, v_{sum} ▷ Calculate l_i
    vmv v_{oldsum}, v_{sum} ▷ Move l_{i-b} to l_i
    vmv v_{oldmax}, v_{max} ▷ Move m_{i-b} to m_i
  end for
  for $h = 0 : d/vl - 1$ **do**
    vload v_{out}, $\vec{o}_N\ [h \cdot vl : \ (h+1) \cdot vl - 1]$
    vdiv v_{out}, v_{out}, v_{sum} ▷ Perform $\vec{o_N}/\ell_N$
    vstore v_{out}, $\vec{o}_N\ [h \cdot vl : \ (h+1) \cdot vl - 1]$ ▷ Store $\vec{o}_N$
  end for
end for

is possible. By processing multiple rows from matrix Q together with corresponding sub-blocks of the K and V matrices in parallel, the residency time of K and V in the cache is extended, achieving better temporal locality. This results in better performance as it will be demonstrated in Section 4.4.

The strategy of processing multiple rows of matrix Q is illustrated in Figure 4.1, where matrix S includes the attention scores computed gradually across all query and key vectors. As shown, blocking is applied to the Q, K, and V matrices. The block size B_r corresponds to the number of consecutive rows

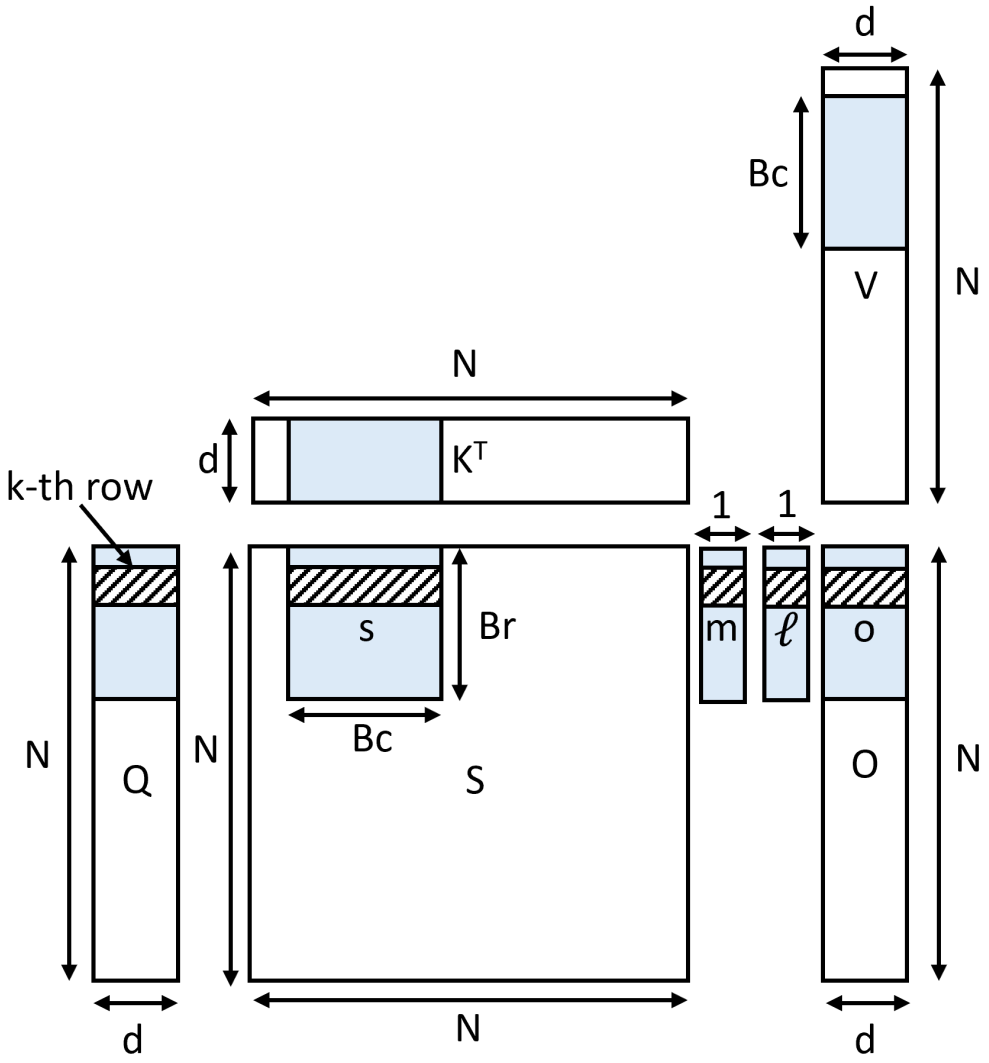


Figure 4.1: The diagram above illustrates how Attention is computed in blocks. The black-colored blocks represent the tiled implementation of Attention mechanism. The black-colored blocks correspond to the k-th row of the attention output. N denotes the sequence length, d denotes the head dimension and Br and Bc are the block sizes that can be controlled.

of Q processed for each block of the K and V matrices, while B_c denotes the number of columns of the K matrix processed in parallel. This partitioning enables the division of the Q matrix into N/B_r blocks, and the K and V matrices into N/B_c blocks. The maximum values and the sums of the exponentiated elements computed for each row are stored, respectively, in the vectors m and ℓ within each block of size B_r.

4.4 Experimental results

The experimental results aim to explore the design space of vectorized FlashAttention and identify the set of parameters that yield the best performance.

All experiments are conducted using a fully implemented decoupled vector unit integrated with an

Table 4.1: Simulated Processor Configuration

Scalar core	• RISC-V ISA (RV64GC), 8-way-issue out-of-order, 16-entry LSQ, 90 physical integer and 90 physical floating-point registers, 60-entry ROB • L1I cache: 1-cycle hit latency, 4-way, 64KB • L1D cache: 2-cycle hit latency, 4-way, 64KB • L1I and L1D have 64B cache line
Vector engine	• 1024-bit vector engine with 32-lane configuration (32-bit elements $\times$ 32 execution lanes) • The vector engine is connected directly to the L2 cache through 16 store queues and 16 load queues
L2 cache	• 8-way, 8-bank • 8-cycle hit latency, 512KB shared by both the big core and the vector engine • 64B cache line
Main Memory	DDR4-2400, 19.2 GB/s memory bandwidth

Table 4.2: The characteristics of contemporary LLM models available on HuggingFace [162]

LLM	Characteristics		
	Hidden Size	Num. Heads	Head dimension
DeepSeek / Qwen-1.5B-L1	1536	12	128
Google / Gemma2-2B-L1	2048	8	256

out-of-order superscalar processor, corresponding to model 1bDV in [142]. The use of a high-performance superscalar core alongside the vector engine is intended to ensure that the scalar portion of the processor does not become a limiting factor in evaluating vector engine performance. Essentially, this configuration allows the vector unit's performance to be measured in isolation, free from interference or bottlenecks introduced by the scalar side. The system is modeled using the gem5 simulator [14], and the key architectural parameters of the simulated processor are listed in Table 4.1

Algorithm 4.6 was executed on attention layers of state-of-the-art LLM architectures available on HuggingFace [162]. Table 4.2 reports the size of the hidden dimension, number of attention heads, and per-head dimensionality for the first attention layer of each model. In transformer architectures, the hidden size refers to the dimensionality of token embeddings and internal hidden states—that is, the number of features used to represent each token at any layer. A larger hidden size enables the model to learn and represent more complex and nuanced patterns. Conversely, the number of attention heads defines how many parallel attention mechanisms are applied to each token. Each head processes information in a distinct subspace of size `hidden size/num heads`, known as the head dimension, which serves as the primary focus of our evaluations. Notably, the head dimension remains consistent across all layers within each LLM.

4.4.1 Speedup achieved through vectorization

Figure 4.2 illustrates the performance improvements achieved through code vectorization for a fixed combination of head dimension and sequence length. To evaluate the impact of vectorization, we analyzed a

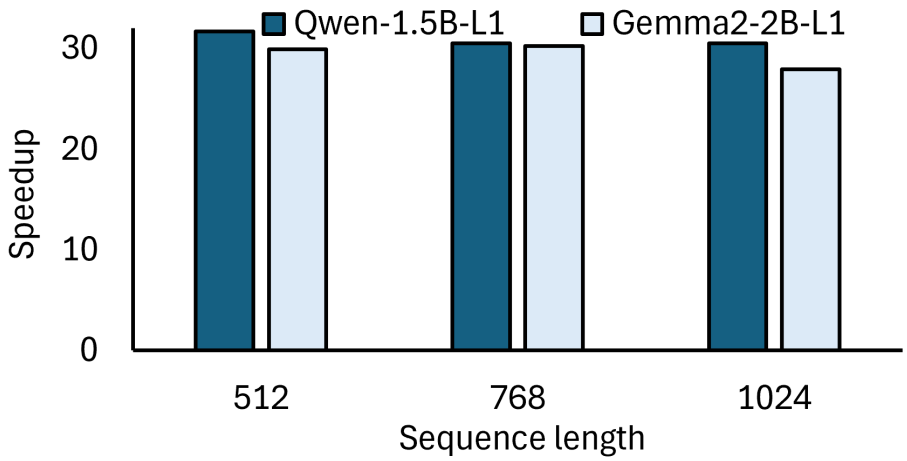


Figure 4.2: The speedup achieved through the vectorization of Algorithm 4.6 for $B_r = 1$ and vector length 32. All results are normalized to the corresponding scalar implementation with the same configuration (sequence length and head dimension).

range of sequence lengths with the parameter B_r set to 1 and the parameter B_c set equal to vector length. A vector length of 32 was used, and the reported metrics correspond to the head dimensions of layers from the Qwen-1.5B and Gemma2-2B LLMs. Additionally, we opt to unroll the loops related to head dimension for a factor of 4 to overcome the unnecessary vector memory operations due to the increased size of head dimension. All results presented in Figure 4.2 are normalized with respect to their corresponding scalar implementations, each having the same sequence length and head dimension as the respective vectorized implementation. As shown, performance approaches its theoretical maximum—aligned with the hardware-supported vector length of 32. However, as the head dimension and sequence length increase, a slight decline in the amount of speedup is observed. This degradation is attributed to the increment on size of all involved arrays, which leads to a greater number of branches and increased pressure on the memory subsystem. The same effect is noticed with the increment of the head dimension, which degrades slightly the performance. The impact of vector length will be examined subsection 4.4.3.

4.4.2 The effect of block size in the vectorized performance

To analyze the impact of the parameter B_r on the execution cycles of Algorithm 4.6, we evaluate its effect under various configurations of sequence length and head dimension, assuming a fixed vector length of 32. All implementations employ loop unrolling by a factor of four in the loops associated with the head dimension.

In Figure 4.3, the speedup achieved by using a B_r parameter greater than 1 is depicted, with all results normalized relative to the implementation employing a B_r parameter equal to 1. As shown in Figure 4.3, employing B_r values greater than one generally yields improved performance, often achieving speedup up to 1.4 compared to using $B_r = 1$.

However, there are notable exceptions, particularly in configurations with a sequence length of 512 and a head dimension of 128, which corresponds to the head dimension for a layer of Qwen-1.5B, as well as those with a sequence length of 1024 and a head dimension of 256, which is the head dimen-

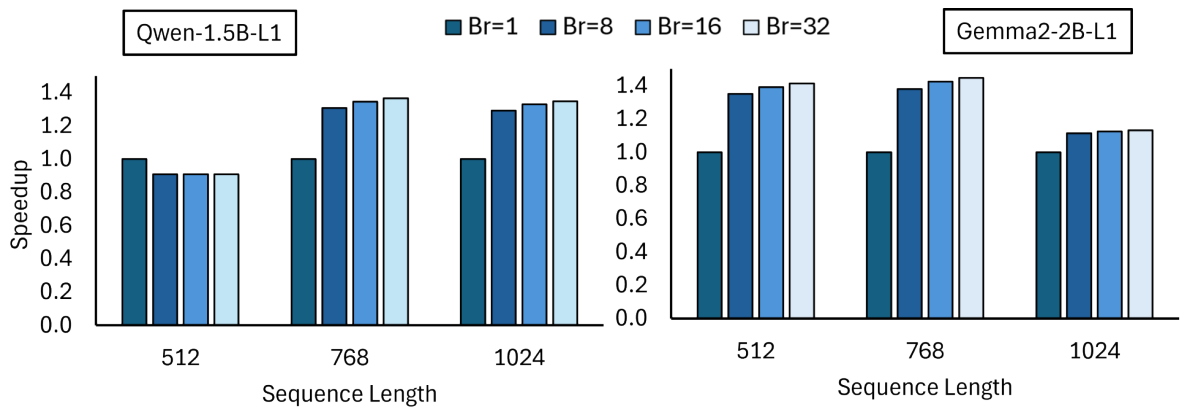


Figure 4.3: The speedup achieved by the algorithm presented in Algorithm 4.6 through the use of a B_r parameter greater than 1, compared to the implementation with $B_r = 1$, is evaluated across varying sequence lengths and the head dimensions of the Qwen-1.5B and Gemma2-2B LLMs, with a fixed vector length of 32.

sion for a layer of Gemma2-2B. In the latter scenario, the performance improvement is slightly reduced due to increased pressure on the memory subsystem resulting from the larger sequence length and head dimension.

Conversely, in configurations where the sequence length is 512 for the head dimension of the layer of Qwen-1.5B, employing $B_r = 1$ outperforms configurations with larger B_r values. This discrepancy arises because, for $B_r > 1$, the number of memory accesses within inner loops increases. Given the relatively small number of iterations in this setup, the increased memory traffic outweighs the benefits of block reuse, ultimately leading to reduced computational efficiency for the attention mechanism.

These improvements are primarily attributed to the reuse of K and V blocks across all rows within a given B_r block. Specifically, each subsequent block of the K and V arrays is accessed only after all rows within the current B_r block have been processed, thereby enhancing data locality and improving cache utilization. However, as the value of B_r increases beyond a certain point, the performance benefits begin to diminish due to reduced cache efficiency.

4.4.3 Scaling hardware vector length

To assess the impact of hardware vector length, we measured the execution time for increasing vector lengths of 32, 64, and 128 data elements, while keeping the blocking factor fixed at $B_r = 32$ and the loop unrolling factor equal to 4. For a vector length of 32, covering a head dimension of 256 as needed in 'Gemma2-2B' transformer necessitates 8 vector registers for each loaded vector of Query, Values, and Keys, not accounting for temporary results. This prevents full loop unrolling related to head dimensions and leads to unnecessary loads. Figure 4.4 depicts the performance gains achieved by increasing vector length relative to the baseline configuration with a vector length of 32. Performance improves with higher hardware vector lengths. However, the rate of improvement decreases when the vector length increases. This flattening in performance improvement mainly happens due to the memory system overloading with more frequent and vector load and store operations requesting more elements. Specifically, under conditions where sequence length equals 512 and head dimension equal to the head dimension of the

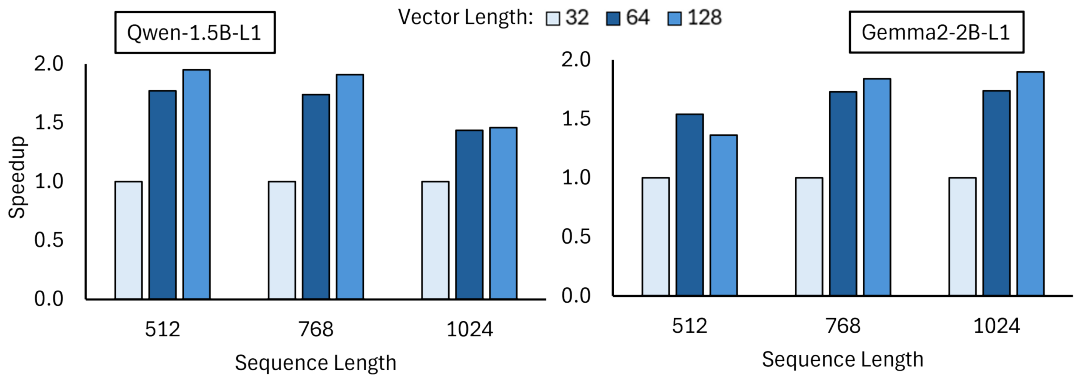


Figure 4.4: The speedup achieved by Algorithm 4.6 for three vector length configurations (32, 64, and 128), with the B_r parameter set to 32, is presented. All the results are normalized with respect to the configuration using a vector length of 32.

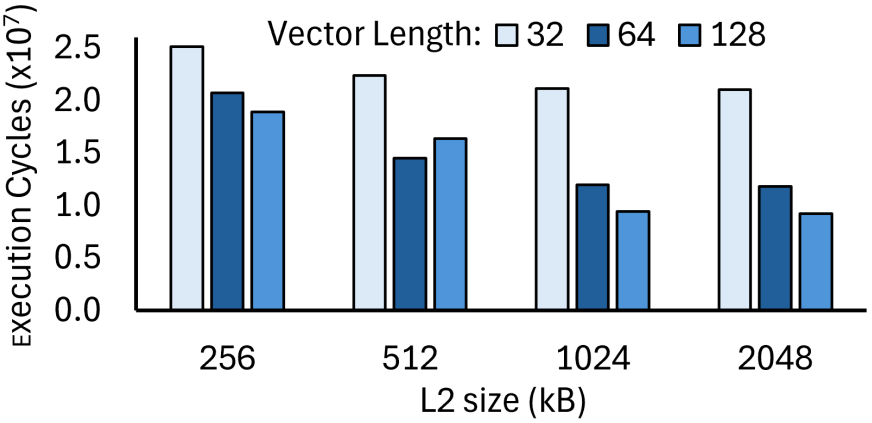


Figure 4.5: Execution cycles of the FlashAttention kernel as a function of L2 cache size (256 kB–2048 kB) for three vector lengths (32, 64, and 128), measured for sequence length 512 for Gemma2-2B-L1's.

layer of Gemma2-2B-L1, the performance worsens when the vector length is 128 compared to when it is 64.

Figure 4.5 provides additional insight into the anomalous behavior observed at the 512KB L2 configuration for Gemma2-2B-L1. As the L2 capacity increases from 256KB to 2048KB, execution cycles generally decrease and the benefit of longer vector lengths becomes more pronounced: vector length 128 typically yields the lowest cycle count, followed by vector length 64, while vector length 32 remains consistently slower. The notable exception occurs at L2 = 512KB, where vector length 64 outperforms vector length 128. This isolated case shows that the slowdown at 512KB is not due to wider vectors, but to a configuration-specific interaction with the cache. In this case, the array's stride and alignment interact unfavorably with the cache's set/bank indexing, causing many accesses to concentrate on a small subset of sets (and/or banks). This effect is less pronounced in smaller vector lengths that communicate less data with the cache for each access.

4.4.4 Comparison to state-of-the-art RISC-V Vector Function Evaluation Libraries

While vectorization and appropriate tiling are the primary contributors to the efficiency of the proposed vectorized FlashAttention kernel, the simplified evaluation of exponentials in the FlashAttention context also reduces execution time. To assess the contribution of this simplified exponent evaluation to the overall performance gains, we re-implemented the proposed approach but replaced the proposed exponent evaluation with the vectorized exponential evaluators offered by state-of-the-art function evaluation libraries designed specifically for the RISC-V vector ISA extension [112, 11]. Both libraries are industrial products operating on standard RISC-V vector ISA, as also done by the proposed approach, without including any custom hardware extensions as done in [157].

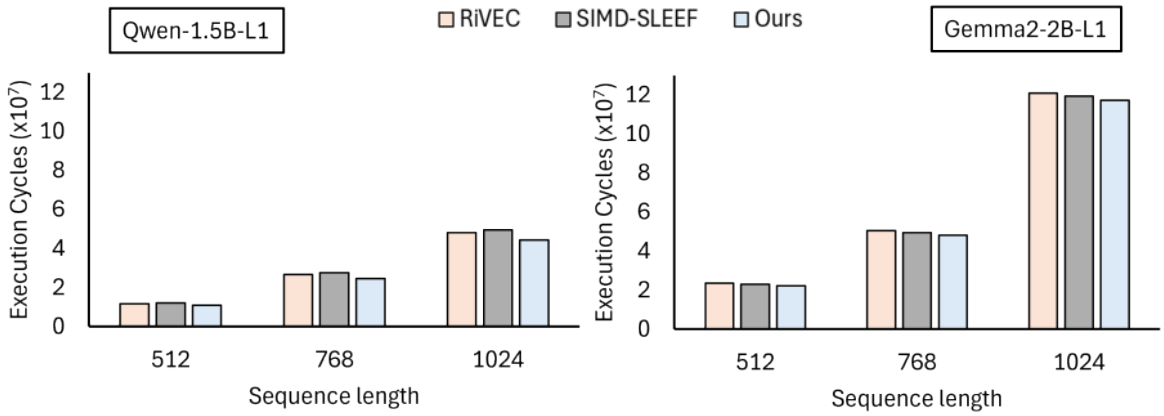


Figure 4.6: The speedup obtained by our approximate vector exponentiation relative to the vector exponent implementations in RiVEC [112] and the SIMD-SLEEF library for elementary-function evaluation [11].

Figure 4.6 reports the execution cycles measured in each examined scenario. For all experiments, we assume a hardware vector length of 32 and a blocking factor of $Br = 32$. Higher vector lengths exhibit exactly the same behavior as verified by additional experiments not shown in this paper.

Across all evaluated configurations and sequence lengths, the proposed simplified computation of exponentials in the context of FlashAttention achieves the highest performance. For Qwen-1.5B-L1, the proposed method achieves lower execution time over both RiVEC [112] and SIMD-SLEEF [11]. For the case of Gemma2-2B-L1, the three implementations are closer in performance; nevertheless, our approach remains the best-performing option across the tested sequence lengths. These results indicate that a domain-specific approximation of the exponential function, tailored to the non-positive inputs produced by the numerically stable softmax, is an effective strategy for reducing the cost of exponentiation within FlashAttention.

4.4.5 Accuracy of exponent approximation on LLM applications

Having explored the efficiency of the proposed vectorized FlashAttention, in this section we quantify the accuracy loss introduced by the clipping and quantization applied during the exponential approximation of the attention score differences.

Table 4.3: Performance of Google's FLAN-T5 LLM model on 10 benchmarks from the GLUE dataset [156]. Accuracy and F1-score are reported for all benchmarks, except for STS-B, where the Pearson Correlation Coefficient (PCC) is used, as it is not a binary classification task.

Benchmarks	Accuracy (%)		F1-score	
	EXP	EXP-ours	EXP	EXP-ours
STS-2	92.1	92.1	0.921	0.931
MNLI-m	87.5	87.5	0.794	0.800
MNLI-mm	84.2	84.2	0.845	0.833
QQP	93.1	93.1	0.930	0.930
QNLI	93.3	93.3	0.933	0.920
CoLA	72.0	72.0	0.830	0.840
MRPC	86.0	86.0	0.900	0.900
RTE	74.3	74.3	0.726	0.738
WNLI	62.0	62.8	0.690	0.710
STS-B	92.0	92.0	-	-

We executed ten representative NLP benchmarks from GLUE [156] dataset using Google's Flan-T5 [111] LLM in the FP-32 configuration. For comparison, we considered two implementations: (a) EXP, where the exponential function e^x is computed using the standard math library and (b) EXP-ours where the function of e^x is approximated using the introduced quantization and exponent approximation. To run inference with the LLM we utilized Microsoft's PromptBench workflow [169].

The results are shown in Table 4.3. We report the Accuracy and F1-score for all benchmarks with an exception for STS-B, where we report the Pearson Correlation Coefficient (PCC), as it is not a binary classification task. As it can be observed both implementations exhibit comparable performance in terms of inference accuracy and the F1-score metric. In some cases, EXP-ours method even outperforms their full-precision counterparts. While it does not indicate that the model is inherently superior, it confirms that the approximation used to perform vector exponentiation does not degrade performance [67].

To clarify better the observed behavior we include also a more focused experimental error analysis for the proposed approach. In the proposed vectorized computation of FlashAttention, arithmetic error may be introduced in two places: (a) when clipping and quantizing the range of the attention score differences; and (b) when applying Blin's approximation [16] in Eq. (4.10). To quantify the contribution of each error case, we executed inference for the benchmarks reported in Table 4.3 introducing only one source of error at a time. The average relative contribution of five indicative benchmarks is shown in Table 4.4.

Table 4.4: Total absolute error contribution (%) per error source for five Benchmarks from the GLUE dataset [156].

Error Source	**Benchmark**				
	MNLI-m	MNLI-mm	QQP	QNLI	MRPC
Clipping	0.029	0.039	0.043	0.04	0.085
$2^x \approx 1 + x$	99.971	99.961	99.957	99.96	99.915

From the percentages reported in Table 4.4, it is evident that Blin's approximation [16], originally derived from [96], which sets $2^x \approx 1 + x$ for $0 < x < 1$, is the dominant source of error. The absolute

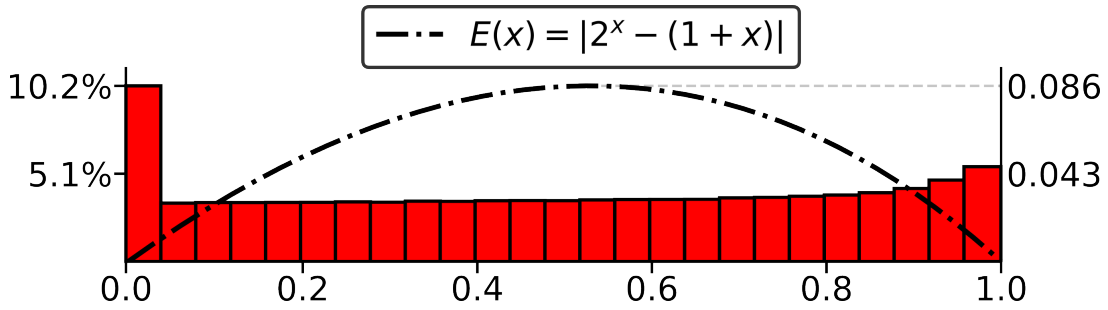


Figure 4.7: The absolute error introduced by Blin's approximation over the input range (0,1), together with the histogram showing how often inputs subject to this approximation occur across all transformer layers and examined benchmarks.

error of this approximation, $E(x) = |2^x - (1+x)|$, over the input range $(0, 1)$ is shown in Figure 4.7. As can be observed, the absolute error is maximized near the midpoint of the interval. Importantly, this error does not accumulate: even two consecutive applications at the point of maximum error result in a total error of only 0.086. Next, we examine how often the inputs to the exponential function that are affected by this approximation during FlashAttention execution fall within the range where the error is largest. To this end, we recorded the values of x whenever the approximation was applied across all LLMs, tasks, and inference steps. The resulting histogram is also shown in Figure 4.7, plotted alongside $E(x)$. The data indicate that most inputs cluster near 0 or 1, where the approximation error is effectively zero. Overall, more than 70% of the cases result in an absolute error below 0.077, which does not meaningfully affect overall LLM inference accuracy.

4.5 Related Work

The standard attention mechanism, while highly effective, is computationally expensive due to its quadratic complexity with respect to sequence length [13]. This complexity becomes a significant bottleneck in Transformer models, particularly when processing long sequences, as commonly encountered in modern NLP tasks such as document summarization and code generation.

To address these challenges, various strategies have been proposed to improve the efficiency of attention mechanisms. A major line of work focuses on approximating the full attention matrix using sparse [23], linear [65], or low-rank [158] attention methods, which aim to strike a balance between computational cost and accuracy. Additionally, techniques like structured sparse pruning [22] have been applied to further reduce complexity by eliminating less significant connections within the attention layers.

Another research direction involves hardware-level acceleration. Custom accelerators [43, 56] have been developed to optimize core components of attention computation, including matrix operations [161] and softmax evaluation [137]. Recent work [157] also targets the Softmax bottleneck by adding lightweight hardware for exponentiation. VEXP integrates a low-cost BF16 exponential unit into a RISC-V core FPU and exposes it via custom (packed-SIMD) instructions. These hardware efforts aim to enhance speed and scalability, especially for long input sequences. In particular, attention accelerators often store key and value vectors locally in SRAM while streaming query vectors to compute attention scores [43, 66, 86],

reducing memory traffic. However, as sequence lengths grow, these vectors must be fetched from slower DRAM, leading to potential performance degradation.

To reduce reliance on memory bandwidth and improve parallelism, in-memory attention computation has also been explored [132]. Moreover, recent designs aim to decouple computational resources from sequence length by accumulating partial softmax results for each column of the attention scores [56, 161]. This allows for accurate softmax evaluation without buffering the entire score matrix, thus avoiding memory spills.

The softmax operation itself presents notable performance challenges due to its dependence on reduction operations, such as max-value determination and exponential summation, which limit parallelism. FlashAttention [29, 28, 110] addresses these inefficiencies by computing attention in smaller tiles, eliminating the need to store the full attention matrix. Designed for GPUs, FlashAttention recomputes scores on the fly and improves both memory and computational efficiency. FlashAttention-2 further optimizes this by avoiding the need for unified softmax hardware; instead, it performs exponentiation and normalization steps separately. Multiple hardware-based techniques have been explored to implement these non-linear operations efficiently, including piecewise-linear approximations with range reduction [68], logarithmic quantization [159], and other functional approximations [119], as well as software vector implementations on emerging ISAs such as RVV.

Complementary to hardware-oriented softmax approximations, prior work has explored software vector implementations of nonlinear functions on RVV. RiVEC provides a gem5-based RISC-V vector simulator and benchmark suite that includes baseline vector exponentiation routines for evaluation [112]. Bavier et al. study RVV-specific vectorization of nonlinear functions (including exp), demonstrating the benefits of ISA-aware implementations [11]. Our work differs by using a domain-specific, low-cost exp approximation tailored to the non-positive softmax inputs in FlashAttention.

Beyond accelerating softmax itself, other optimization techniques aim to reduce latency and energy consumption, such as computation skipping based on token similarity [53, 131], and quantization strategies that minimize data transfer overhead and improve accelerator utilization [89].

4.6 Conclusion

The FlashAttention kernel streamlines attention computation by fusing softmax evaluation with the attention update, enabling a memory efficient formulation that avoids materializing the full attention matrix. Originally developed for GPUs, it computes attention in fixed size tiles, largely decoupling on chip working set requirements from the input sequence length. In this chapter, we adapt FlashAttention 2, an optimized successor to the original FlashAttention, to vector processors and present a fully vectorized design targeting the RISC V Vector ISA.

A key result of this study is that FlashAttention's compute in tiles principle transfers well to vector architectures when the kernel is reformulated to fully exploit data level parallelism. By expressing the entire attention computation, including the numerically stable softmax update, using vector instructions, we show that attention can achieve high compute efficiency while keeping intermediate state largely within the vector register file. This is especially important for vector processors, where performance

and energy efficiency depend on sustained vector utilization and reduced memory traffic. Our results show that suitable tiling improves data reuse and reduces bandwidth pressure, making fused attention well suited to vector execution.

Our analysis further shows that performance depends not only on the use of vectorization, but also on how work is partitioned and scheduled. Different row block sizes for the Q matrix expose trade offs among parallelism, reuse of K and V tiles, register pressure, and the reduction overheads required for softmax normalization. The best configurations maintain contiguous, vector friendly access patterns while balancing the number of live intermediate vectors needed for running softmax statistics and partial outputs. Evaluations across multiple hardware vector lengths highlight interactions between vector width, tile geometry, and kernel structure, suggesting that efficient vectorized attention requires coordinated choices across tiling strategy, vector length, and vector implementations of key primitives such as exponentiation and reductions.

Chapter 5

Vector Engines with Reduced Number of Physical Vector Registers

5.1 Introduction

Achieving high computational throughput in data-parallel workloads inherently demands large amounts of on-chip storage to feed wide execution pipelines. In modern vector architectures, this storage takes the form of the VRF. While keeping operands resident in massive vector registers is highly beneficial for data reuse in complex machine-learning kernels, the physical VRF typically becomes the dominant contributor to both silicon area and static power. This creates a fundamental tension in the design of low-cost edge processors, where strict resource budgets often force architects to cripple vector processing capabilities by aggressively shrinking datapath widths. Resolving this conflict requires a fundamental rethinking of vector state management, exploring how full architectural compliance and high parallel throughput can be sustained even when the physical capacity of the underlying register file is substantially reduced.

To understand the critical importance of hardware area in resource-constrained environments, it is necessary to examine the integration of vector capabilities into low-cost processors. Ultra-low-cost cores prioritize area efficiency and power consumption over peak performance, typically adopting a lightweight single-issue datapath with small caches. Nevertheless, the increasing demand for machine learning at the edge requires enhancing these cores with Vector Processing Units (VPUs) to achieve acceptable throughput.

A VPU can be integrated alongside the existing scalar core to share fetch, decode, and memory pipeline stages, as illustrated in Figure 5.1. Vector processing relies on parallel execution units fed directly by the VRF. The vector *length* denotes the bit-width of each vector register and determines the maximum degree of parallelism. However, adding this vector state storage introduces a massive hardware overhead. This problem is especially pronounced under the RVV ISA, which mandates 32 architectural vector registers and supports arbitrary vector-length multipliers (LMUL).

To evaluate this overhead, a RISC-V compliant vector unit was integrated within the Codasip L31 RISC-V core [25] using an organization similar to Figure 5.1 and implemented in 28 nm technology. The VPU uses a vector length of 256 bits and has 8 vector lanes supporting 8-, 16-, and 32-bit integer alongside `bfloat16` calculations. As shown in the area breakdown in Figure 5.2, the VRF consumes

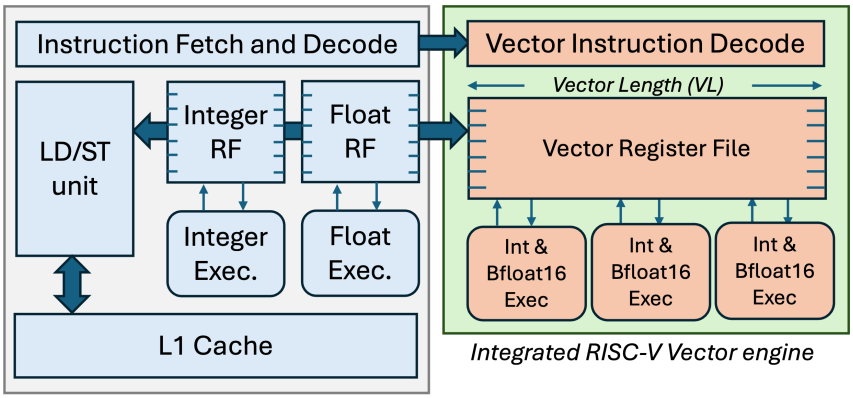


Figure 5.1: The organization of a scalar RISC-V CPU augmented with an integrated VPU. The scalar and vector datapaths share the same load/store memory port to the L1 cache.

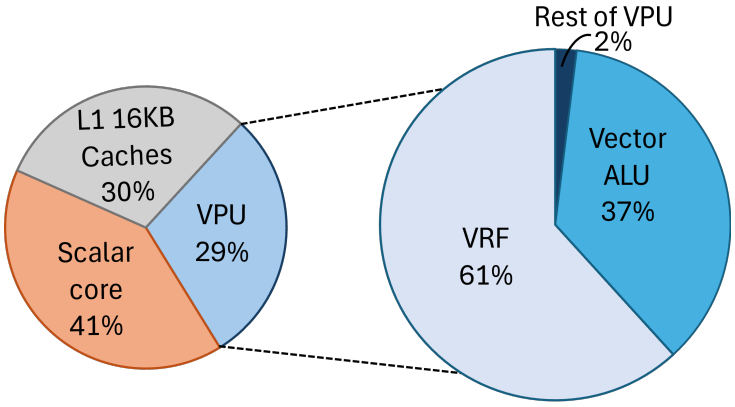


Figure 5.2: Percentage breakdown of the hardware area consumption of the main components of a RISC-V CPU with an integrated VPU. The VRF consumes 61% of the VPU area.

more than half (61%) of the VPU's total area.

Faced with this severe area bottleneck, architects have explored several compromises to reduce the physical cost of vector state. The most typical method is to use narrower vector registers and cut down the number of vector execution lanes. While effective in saving space, this approach significantly restricts data-level parallelism and directly degrades performance.

The central challenge, therefore, is to reduce the physical cost of vector state while preserving the full flexibility and performance potential of the RVV programming model. Rather than reducing the vector *length* (which degrades parallelism) or introducing complex renaming mechanisms, this chapter demonstrates a more effective approach. The proposed technique lowers the hardware area cost by decreasing the "height" of the VRF, meaning the total number of physical vector registers it stores. This is achieved while seamlessly maintaining full compatibility for the 32 vector register names and arbitrary length multiplier settings dictated by the RVV ISA.

5.2 Vector Register dispersion

5.2.1 Dispersion Mechanism

Determining vector operand location

Once a vector instruction enters the ID stage, its two source operands *and* its destination operand must be read from the cVRF. Note that the destination operand is also needed, because the RISC-V ISA has numerous instructions that utilize the destination register in their execution (e.g., `vmacc`, `vmadd`, etc.). Thus, the Register Dispersion mechanism first checks to see which (if any) of these three operands are currently located within the cVRF. This is achieved by accessing the Tag Array independently for each of the three operands. The process of checking for cVRF residency is performed serially for all three operands to accommodate for the cases where cVRF evictions are required, as will be explained shortly.

If there is a hit in the Tag Array, the operand is already within the cVRF, and the tag hit location indicates the corresponding location within the cVRF (i.e., the physical mapping). This physical location (index) is carried – through the ID/EX pipeline register – to the next pipeline stage, where the cVRF will be accessed.

On the other hand, if there is a miss in the Tag Array, then the missing operand must be fetched from the memory sub-system. The mechanism's Control Unit stalls the processor and proceeds differently, based on cVRF slot availability.

Replacement policy

If there is at least one free slot in the cVRF, then the control unit dynamically generates appropriate `load` micro-operations to load the operand from memory into the cVRF, in the location pointed to by the circular FIFO's tail pointer. Recall that the memory address of each architectural register is a priori fixed and known. Again, the physical location/index in the cVRF is carried to the next pipeline stage.

If there are no free slots within the cVRF, then an *eviction* must first be performed, before loading the missing operand. In its current incarnation, the proposed mechanism employs a FIFO replacement policy, due to its simplicity. By leveraging the existing circular FIFO implementation of the cVRF, a FIFO replacement policy is achieved by simply evicting the register that is located at the head pointer of the cVRF. On the contrary, a Least Recently Used (LRU) policy would require additional counters. Despite its simplicity, the FIFO replacement policy will be demonstrated to achieve very high hit rates. After locating the evictee, the Control Unit first generates appropriate `store` micro-operations to store the evicted register to memory. The memory address is determined by the evictee's tag entry, i.e., its architectural register number. Subsequently, the Control Unit generates `load` micro-operations to load the missing operand from memory into the cVRF (in the location pointed to by the tail pointer). Same as before, this cVRF index is carried to the next pipeline stage.

Upon completion of the above-mentioned process, all three operands required by the vector instruction are located within the cVRF, and their indexes are stored within the ID/EX pipeline register. The vector instruction then proceeds to the EX pipeline stage. During this stage, the cVRF is accessed using the indexes from the previous (ID) stage, and the register values are sent to the VPU for instruction

execution.

In summary, the Register Dispersion mechanism employs a simple Control Unit and a split-type cVRF, with its Tag Array in the ID stage and its Data registers in the EX stage, to dynamically manage the usage of the vector registers.

Exception handling

Register Dispersion is transparent to the architectural RVV state: all 32 vector registers remain architecturally defined, while the cVRF acts as a microarchitectural cache of a subset of these registers. Therefore, exception and interrupt handling follows the standard RISC-V procedure (e.g., trap redirection via `mtvec` and trap state recorded in `mepc`, `mcause`, and `mtval`) [25]. To preserve precise architectural state at trap boundaries, we ensure that any modified (dirty) cVRF entries are written back to their dedicated backing-store locations and the corresponding cVRF tags are invalidated upon trap entry (interrupt or synchronous exception). After this drain/flush step, the complete architectural vector context resides in the memory, and the interrupt handler can save/restore vector state using the conventional software mechanisms, without requiring ISA-visible changes due to cVRF. Upon return from the handler, execution proceeds normally and cVRF state is repopulated on demand.

5.2.2 Supporting LMUL under Register Dispersion

The goal of this section is to explain how grouped vector operands are supported when only a limited number of physical vector registers are kept in the cVRF. This is a key challenge in register dispersion, where a single architectural instruction may require multiple grouped registers, while only a small subset of the architectural vector state is resident in the cVRF at any given time.

This challenge follows directly from the semantics of the RVV length multiplier. When LMUL is greater than one, a logical vector operand spans multiple architectural vector registers. As a result, the execution of a single vector instruction requires access to a larger number of source and destination register groups. In a conventional full-size VRF, these grouped operands are simply read and written through the normal register-file organization. In a dispersed-register design, however, the required groups may not all be present locally. Consequently, LMUL support must be integrated with the cVRF residency, replacement, and refill mechanisms.

The key observation is that the difficulty introduced by LMUL is not merely that more architectural state is involved, but that this wider state must be supported under constrained local storage capacity. If several grouped operands are required by the current instruction, the cVRF may be unable to hold them all simultaneously. Therefore, the execution of the instruction must be coordinated in a way that reuses the available cVRF slots across multiple internal steps while preserving the architectural behavior of the original RVV instruction.

To achieve this, register dispersion supports LMUL through an instruction-sequencing mechanism tailored to the compact register-file organization. Rather than requiring all grouped operands to be present in the cVRF at once, the control logic breaks the execution of a wide logical vector instruction into a sequence of narrower internal steps. In each step, only the subset of source and destination register groups needed for the current portion of the computation is required to reside in the cVRF. Once a step

completes, the next subset of grouped operands may be brought in, allowing the same limited pool of physical registers to be reused across the full execution of the instruction.

This sequencing mechanism is particularly well suited to register dispersion, since it allows LMUL support to be maintained without increasing the local VRF to the point where the area-saving benefits of the cVRF would be lost. At the same time, it preserves full RVV compatibility, because the decomposition into internal sequencing steps is entirely microarchitectural and remains invisible to software. Existing vectorized kernels therefore execute correctly without modification, even when instructions operate on grouped registers.

An important difference from the generic LMUL support described earlier is that, under register dispersion, each sequencing step must additionally interact with the cVRF management logic. Before a step can execute, the control path must determine whether the required source and destination register groups are already resident in the cVRF. If they are not, the corresponding groups must be fetched from the backing memory hierarchy and placed into the available cVRF slots. When no free slot is available, one of the currently resident registers must be selected for eviction according to the replacement policy. Thus, the execution of grouped vector operands under register dispersion becomes tightly coupled to register residency and replacement behavior.

For this reason, the cost of supporting LMUL in the proposed design depends not only on the sequencing logic itself, but also on the pressure that grouped operands place on the compact register file. Larger LMUL values increase the number of architectural register groups touched by a single instruction and therefore raise the likelihood that operand groups will need to be loaded, replaced, or revisited during execution. Nevertheless, by sequencing the grouped operation over a small number of reusable cVRF slots, the proposed design is able to preserve support for RVV register grouping while retaining the substantial area and energy savings of a compact local register file.

In this sense, LMUL is not treated here as an isolated feature, but as a representative stress case for the register-dispersion mechanism. If grouped vector operands can be supported correctly and efficiently despite the reduced physical register capacity, then the same compact storage organization can also sustain a broad range of practical RVV workloads without sacrificing ISA compliance.

5.3 Experimental Evaluation

The experimental evaluation has three main objectives: (a) demonstrate that a reduction in the number of registers stored within the vector register file has no effect on application performance; (b) assess the proposed mechanism on full applications, including LLM inference and CNN workloads, to verify that the kernel-level trends translate to end-to-end performance; (c) validate that the integration of LMUL support within the cVRF enables actual performance improvements in some applications with wide vector operations, by leveraging instruction sequencing to efficiently utilize limited datapath and register resources; and (d) show that the use of the proposed light-weight cVRF reaps substantial hardware area/power savings, as compared to a full-size VRF that stores all the ISA-defined architectural vector registers.

Table 5.1: Processor configuration

Scalar core	• Codasip L31 RISC-V embedded CPU (RV32IMFCB) [25] • Single-issue, 3 pipeline stages, 200 MHz clock frequency • 32 32-bit scalar and 32 32-bit floating-point registers • L1I cache: 1-cycle hit latency, 2-way, 16 KB • L1D cache: 1-cycle hit latency, 2-way, 16 KB • Main memory: 1–5-cycle latency @200MHz, 2 MB
VPU	• 256-bit vector engine with 8-lane configuration (32-bit elements $\times$ 8 execution lanes) • Full VRF (32-Vregs) / cVRF (3/4/5/6/7/8/16-Vregs)

5.3.1 Experimental Setup

The commercially-available L31 low-cost 3-stage RISC-V microprocessor [25] is augmented with a VPU and the proposed Register Dispersion mechanism were fully implemented in CodAL, an architecture description language, which facilitates the design of processors. The micro-architectural parameters of the processor are summarized in Table 5.1.

For the *performance* evaluations, the Codasip Studio simulator was used, which can *cycle-accurately* simulate processors designed in the CodAL language while executing benchmark applications, and all the performance metrics were derived using this simulator. All benchmarks were compiled using `clang` v17.0.2 targeting the RISC-V Vector ISA. The applications employed in the experimental results and their characteristics are depicted in Table 5.2. The applications examined include benchmarks taken from the RiVEC benchmark suite [112] and Ara's application collection [112]. Machine learning kernels from Natural Language Processing (NLP), CNNs, and transformers employed in LLMs are also included. More specifically, the 'DenseNet121 Layer105' and 'ResNet50 Layer112' benchmarks correspond to layer 105 of DenseNet121 [52] and layer 10 of ResNet50 [44], respectively. The execution of those CNN layers is mapped to an equivalent matrix multiplication after employing the im2col transformation on the corresponding input features and weights [154]. Additionally, a self-attention layer of the Bert application [32] – implemented in a vectorized form using the FlashAttention-2 technique [28] – is also included. Unless otherwise stated, the performance results are obtained using an LMUL configuration of 1.

For the *hardware cost* analysis, the RTL implementation of the low-cost processor with its VPU and the Register Dispersion technique were generated using Codasip Studio. The RTL was then synthesized using the Cadence digital implementation flow and a 28 nm standard-cell library for the hardware cost evaluation. For the FPGA resource analysis, we evaluated the same RTL on a Xilinx Virtex-7 FPGA (xc7vx690tffg1761-2) using Vivado 2025.2, and we report LUT/FF/DSP/BRAM utilization.

Table 5.2: Applications characteristics

Application	Domain	Characteristics
PathFinder	Grid Traversal	Rows:32 Columns:32
Jacobi-2D	Engineering	Problem size:128 steps:10
Somier	Physics Simulation	Problem size:32 steps:2
GemV	NLP	(512 x 512) x 512
Dropout	ML	Vector Length:131072 Scale:0.5
fconv2d-7x7	CNN	256 x 256 filter size:7
DenseNet121 Layer105	CNN	(32 x 1152)x(1152 x 64)
ResNet50 Layer10	CNN	(128 x 256)x(256 x 784)
FlashAttention-2	Transformer	Seq. Length:200 Hidden Dim.:64 Block size row:1 Block size column:128

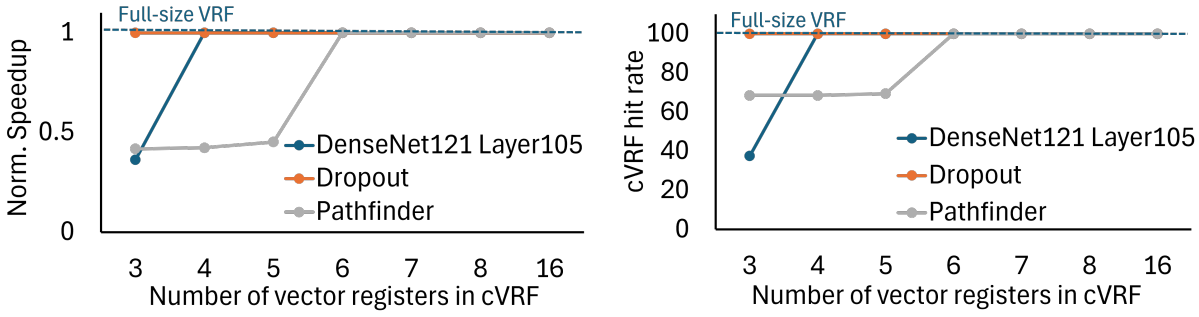
5.3.2 Performance results

Speedup achieved with a full-sized VRF

The addition of a VPU to the scalar CPU core can reap significant performance uplifts when executing workloads with high degree of DLP. The second column of Table 5.3 shows the performance speedups achieved in the examined vectorized benchmark applications when using a VPU, as compared to scalar (i.e., CPU-only) execution. In this experiment, the VPU uses a full-size VRF that can store all 32 architectural vector registers, and the vector length is set to 256 bits to align with the common cache line size of 32 bytes used in many low-cost processors [163, 140]. As can be seen, the speedups range from around 4.3× to 8×, thereby corroborating the benefit of augmenting the CPU with a VPU.

Table 5.3: Speedups achieved when using a VPU with a full-size VRF and 256-bit vector length over scalar execution. The VRF utilization percentages are also shown.

Applications	Speedup	#Active Vector Registers	VRF util. (%)
PathFinder	7.99×	6	18%
Jacobi-2D	6.48×	7	21%
Somier	7.82×	14	44%
GemV	6.89×	9	28%
Dropout	4.3×	3	9%
fconv2d-7x7	7.74×	15	47%
DenseNet121 Layer105	7.82×	4	12%
ResNet50 Layer10	7.63×	4	12%
FlashAttention-2	7.91×	32	100%



(a) Achieved performance (b) Hit rates

Figure 5.3: (a) The performance achieved by various cVRF sizes, and (b) the hit rates in cVRFs of various sizes, when executing three indicative benchmark applications. All results are normalized to the performance of a full-size VRF.

Nevertheless, this dramatic improvement in performance comes at a significant hardware cost. As shown in Figure 5.2, the addition of a VPU – with its full-size VRF – consumes area comparable to the area occupied by the scalar core. Such area cost is often prohibitive for ultra low-cost processors that are resource-constrained. To decrease the area overhead, we focus on the VRF of the VPU, which occupies 61% of the entire VPU area (see Figure 5.2).

Reducing the number of vector registers does not hurt performance

The founding premise of the proposed Register Dispersion mechanism is the observation that most applications tend to use very few architectural registers during execution. This behavior is obvious in the third column of Table 5.3, which lists the number of architectural vector registers used by the evaluated benchmark applications. As can be seen, in most cases, the number of used registers is much lower than the 32 defined by the ISA. The use of very few vector registers is also reflected in the VRF utilization percentages shown in the fourth column of Table 5.3. Special reference should be made to the 'FlashAttention-2' benchmark. Even though said application uses all 32 vector registers throughout its entire execution, it only operates on very few registers at any given time, as will be demonstrated later. In other words, each execution phase works on a small subset of the 32 registers.

Given this salient attribute shared by most of the examined applications, it is worthwhile to investigate how many registers a cut-down VRF – i.e., the proposed cVRF – must be able to store to reach the same performance levels as when using a full-size VRF.

Toward this end, and based on the numbers of used vector registers reported in Table 5.3, we investigate the use of cVRFs of various sizes. Specifically, we investigate cVRFs that can store from 3 to 16 vector registers. To aid visual clarity (and since all applications show similar trends), the performance results for an indicative subset of the examined benchmarks are illustrated in Figure 5.3(a). All results are normalized to the performance achieved when using a full-size VRF that can store all 32 architectural vector registers defined by the ISA. This normalization reference is the best possible performance that can be achieved. The vector length is set to 256 bits in all configurations. As shown in Figure 5.3(a), the performance of the smaller cVRFs is markedly worse than the full-size VRF, because they cannot fit all

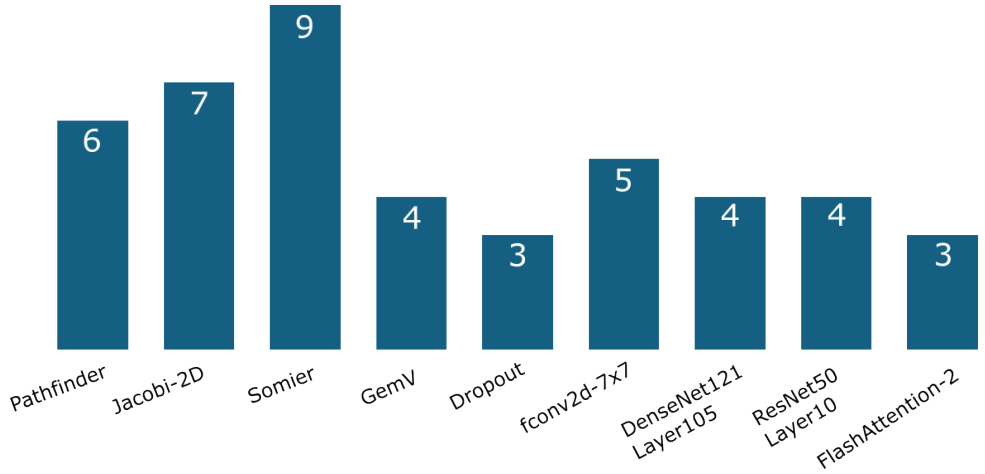


Figure 5.4: The minimum number of vector registers required to achieve a hit rate above 95% in the compact VRF.

the registers accessed by the applications. However, as the size of the cVRF increases, the performance improves and eventually – when the size of the cVRF is 8 – reaches the performance of the full-size VRF. Hence, this experiment demonstrates that *storing 8 vector registers* in the cVRF is *sufficient to achieve full performance* (i.e., no degradation) *for a wide range of vector applications*.

Figure 5.3(b) shows the hit rates in the various examined cVRFs for the same indicative subset of benchmarks. As the number of stored vector registers approaches the maximum number used by each application, the hit rates increase. As is evident, the bigger cVRFs (of sizes 6 and above) have almost 100% hit rate. On the other hand, the small cVRFs of sizes 3 to 5 suffer from many misses in the 'Pathfinder' benchmark, which uses more registers that can fit within the cVRF.

To further support our assertion that storing eight vector registers within the cVRF is sufficient in achieving peak performance, we determined the minimum number of registers required to achieve a hit rate exceeding 95% within the cVRF for all examined applications. The results are illustrated in Figure 5.4. Similar to the previously mentioned behavior of the 'FlashAttention-2' benchmark, even though some applications require more than eight vector registers throughout their entire execution, not all the vector registers are utilized *simultaneously* at any given time. This contributes positively to the hit rate within the cVRF. This is precisely the reason why the 'FlashAttention-2' application exhibits a very high hit rate even with a cVRF of only 3 vector registers, even though it uses all 32 vector registers over its entire execution. Again, the results in Figure 5.4 demonstrate that a cVRF with eight registers is adequate for most cases.

5.3.3 Performance comparison under equal-area VRF organizations

As explained in Section 5.1, another approach to reducing the hardware area of the VRF is to decrease the vector *length* of each vector in the VRF, instead of the number of stored registers. We hereby demonstrate that this alternative approach leads to much lower application performance, as compared to the performance reaped with Register Dispersion. For a fair comparison, we compare two different VRF implementations that have *equal area*: (1) a cVRF that can store 8 vector registers, each of 256-bit vector

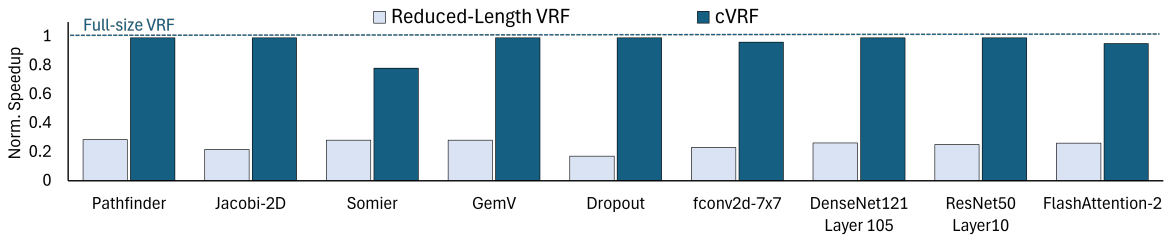


Figure 5.5: Performance comparison between Register Dispersion with a cVRF of 8 256-bit vectors versus an *equal-area* VRF with 32 vectors of reduced 64-bit length. The results are normalized to the performance achieved when using a VPU with a full-size VRF (i.e., 32 vectors of 256-bit length).

length; and (2) a VRF that can store all 32 architectural vector registers, but with a reduced vector length of 64 bits. In other words, the first implementation corresponds to the proposed Register Dispersion approach of reducing the VRF "height" by a factor of 4, whereas the second implementation corresponds to the approach of reducing the "width" (i.e., vector length) of the full-size VRF by a factor of 4.

The obtained results are depicted in Figure 5.5 and are normalized to the performance achieved when using a VPU with a full-size VRF (i.e., 32 vectors of 256-bit length). As can be seen, the Register Dispersion setup achieves near-identical performance to the system with a full-size VRF in *most* benchmark applications. With the 'Somier' and 'fconv2d' benchmarks, the performance is lower, since these applications access more than 8 architectural vector registers. Nevertheless, the performance of 'fconv2d' is not affected considerably. This can be attributed to the strategic grouping and unrolling of vector register names within the application's code.

Interestingly, the performance results for all six benchmark applications from the ML-related domains (i.e., NLP, general ML, CNN, and Transformer) are excellent under Register Dispersion. As can be seen in the right six sets of bars in Figure 5.5, the cVRF approach comes very close to matching the performance of a full-size VRF with only a quarter of its size. This behavior clearly indicates that the proposed mechanism is, indeed, a promising approach to accelerating ML applications on ultra-low-cost processors on the edge.

Compared to Register Dispersion, the configuration with reduced vector length performs much worse across all benchmarks. The reason for this huge disparity in achieved performance is the severely diminished data-level parallelism when decreasing the vector *length*.

We also extend our comparisons from representative kernels to complete LLM and CNN application workloads: (i) *llama2.c* inference [63] using the TinyLlama "stories260K" model [64], (ii) the MLPerf Tiny image classification benchmark (MLPerf Tiny [10]) on the CIFAR-10 dataset [70], and two CNN models from MCUNet [80]: (iii) *mcvww0*, trained for the Visual Wake Words task [24], and (iv) *mcin0*, trained on ImageNet [31]. For *llama2.c* inference, we use a deterministic, pre-tokenized prompt to avoid tokenizer and I/O overheads in the measured region. Specifically, we execute single-batch decoding with a fixed prompt of 16 tokens and generate 48 additional tokens, for a total context of 64 tokens. For the CNN workloads, we use the native input resolutions of each benchmark: MCUNet *mcvww0* processes 64×64 images, MCUNet *mcin0* processes 48×48 images, and the MLPerf Tiny CIFAR-10 benchmark uses 32×32 images.

Figure 5.6 reports the *normalized speedup* for these runs. Results are normalized to the *full-size VRF*

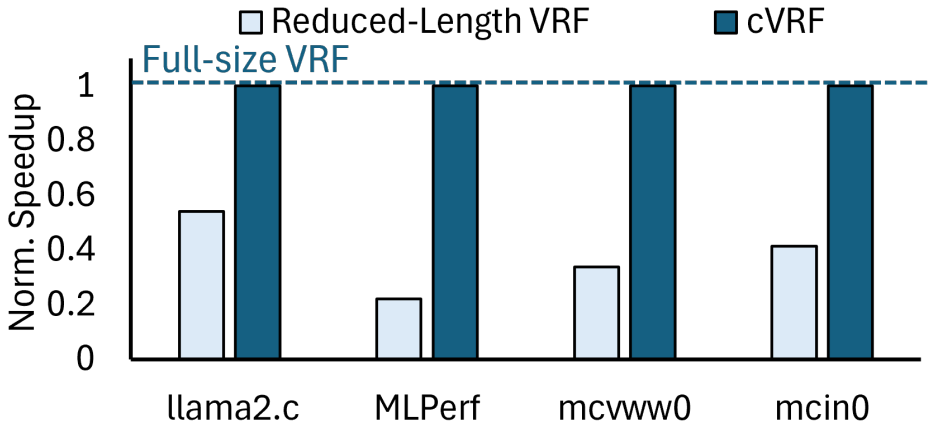


Figure 5.6: Normalized speedup for LLM and CNN workloads, normalized to the full-size VRF baseline (shown as the horizontal reference line). We compare an equal-area 32-register VRF with reduced vector length, and the proposed cVRF with 8 physical vector-register slots.

implementation (32 architectural vector registers), which is shown as the horizontal reference line at 1.0. Overall, the proposed cVRF achieves normalized speedups close to the full-size VRF across all four examined applications, whereas the reduced-length VRF exhibits a significant performance loss.

Two key observations emerge from Figure 5.6. First, the cVRF configuration tracks the full-size VRF baseline closely across all four applications, indicating that the reduced physical register capacity is sufficient for the dominant execution phases of these workloads. Second, the equal-area reduced-vector-length baseline shows a consistently larger gap, highlighting that, under an equal hardware budget, preserving vector width and using Register Dispersion to manage a smaller set of physical registers is more effective than narrowing the vector datapath. Together, these results support the conclusion that a compact VRF organization can retain near-baseline performance for representative LLM/CNN workloads while reducing VRF hardware cost.

5.3.4 Performance scaling with increasing vector length multiplier and software unrolling

To further examine how performance scales in the presence of a reduced-size cVRF, we evaluate two complementary mechanisms that affect vector operand width and register pressure: (i) *vector length multiplication* (LMUL), which groups multiple physical vector registers into a single logical operand, and (ii) *software loop unrolling*, which can reduce loop overhead and improve reuse, but may increase the number of simultaneously live vector registers.

Impact of increasing LMUL

Figure 5.7 illustrates the impact of various LMUL configurations on application performance for various number of available physical vector registers. The figure also examines performance scaling when the register file is fully populated. Each sub-figure corresponds to a specific application: (a) DenseNet121, Layer 105; (b) Dropout; and (c) Pathfinder. The y-axis represents the performance speedup relative to the baseline configuration – i.e., LMUL=1 with three available vector registers – while the x-axis shows the number of available physical vector registers.

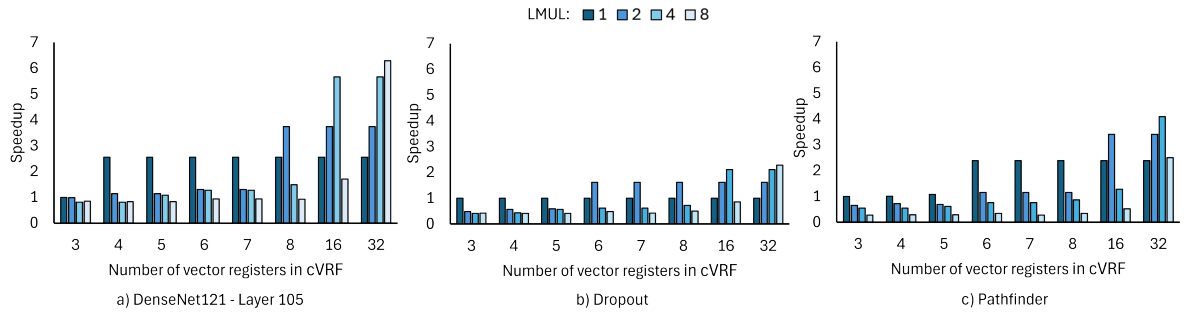


Figure 5.7: Performance speedup for LMUL configurations of 1, 2, 4 and 8 evaluated on three applications: (a) DenseNet121 Layer 105, (b) Dropout, and (c) Pathfinder. The results are normalized to the configuration of LMUL=1 with three available vector registers.

For DenseNet121 Layer 105, which requires four logical vector registers, an interesting trend emerges. When eight physical registers are available, this corresponds to four logical registers under an LMUL configuration of 2. At this point, the actual benefit of using LMUL becomes evident, as the configuration with LMUL=2 achieves superior performance compared to LMUL=1 and equals the maximum possible performance of the LMUL configuration of 2. However, when the available physical registers are fewer than eight, the maximum performance of the LMUL=1 configuration outperforms LMUL=2. This behavior is primarily due to two factors. First, **register replacement overhead**, whereby – with fewer than eight registers – LMUL=2 cannot allocate the required number of logical registers without frequent register spilling and reloading. This, in turn, leads to additional data movement. The second factor is **operand requirement**, since the application requires eight logical operands under LMUL=2, compared to four under LMUL=1. This higher operand requirement increases register pressure and the frequency of register interchanges when the register file is constrained.

For the Dropout application, similar behavior is observed, but the performance crossover point occurs when six vector registers are available. Dropout requires three logical vector registers, which translates into six physical registers under LMUL=2. When six registers are available, LMUL=2 can operate without excessive register swapping, allowing it to achieve comparable, or slightly superior performance relative to the peak performance of LMUL=1 and to reach the maximum performance achievable for LMUL=2. With fewer than six registers, however, the same overhead issues as in DenseNet prevail.

For Pathfinder, the results show that the LMUL=2 configuration never exceeds the peak performance achieved with LMUL=1 for any of the evaluated register counts. This behavior arises because Pathfinder requires six logical vector registers for execution. When LMUL=2 is applied, this requirement translates into twelve physical registers, which exceeds the capacity of all tested configurations (in which the maximum available was eight registers). Consequently, LMUL=2 experiences substantial register pressure and frequent data swapping, preventing it not only from surpassing the maximum performance of LMUL=1 when fewer than 16 physical vector registers are available. This illustrates the critical relationship between an application's logical operand requirements and the size of the physical register file.

Across the three workloads, LMUL=4 is particularly sensitive to limited register availability, because it multiplies the physical register demand by $4\times$ per logical vector register. As a result, when the cVRF

provides only a small number of registers (e.g., up to eight), LMUL=4 frequently cannot keep the full working set resident, leading to frequent evictions, spill/fill traffic, and reduced speedup. Once sufficient registers are available to accommodate the application's logical operands (e.g., at 16 registers for DenseNet121 and for Dropout), LMUL=4 begins to realize its intended benefit and its performance increases sharply, approaching the corresponding full-size VRF limit. For Pathfinder, however, the working set remains comparatively large, so LMUL=4 requires a larger cVRF to avoid thrashing; consequently, its gains appear later and remain below the best-performing configuration in the tested range.

LMUL=8 imposes the highest physical-register demand ($8\times$ per logical vector register) and therefore exhibits the strongest sensitivity to cVRF capacity. Under limited register counts, the enlarged operand footprint cannot be kept resident, leading to frequent evictions and spill/fill activity that suppresses speedup relative to smaller LMUL settings. Once register constraints are sufficiently relaxed, however, LMUL=8 becomes advantageous: for DenseNet121 and Dropout it delivers the best performance at the largest evaluated register counts, approaching the full-size VRF limit. In contrast, Pathfinder remains constrained by its comparatively high operand requirements; within the evaluated register budgets, LMUL=8 does not realize the same level of improvement because the required working set cannot be fully accommodated without substantial swapping.

When register constraints are removed (i.e., when 32 physical vector registers are available in Figure 5.7), larger *LMUL* values tend to improve performance, because operating on wider vectors reduces the number of iterations required to complete vectorized operations. However, an important observation is that performance does not scale linearly with increases in LMUL, or with the number of registers utilized: for example, doubling LMUL from 1 to 2 does not yield a twofold speedup, LMUL=4 does not achieve a fourfold improvement, and even LMUL=8 falls short of an eightfold speedup despite operating on wider vectors. The primary reason for this sub-linear scaling is that the vector ALU datapath remains constant regardless of the LMUL configuration. While LMUL determines how many physical registers are grouped together to form a single logical vector register, the computational throughput of the ALU itself does not change. Consequently, increasing LMUL reduces the number of executed instructions but does not proportionally increase the execution rate of each instruction. This architectural limitation caps the achievable speedup, even when the register file is no longer a bottleneck.

The key takeaway is that LMUL can deliver performance improvements even with a reduced-size register file, provided that the number of logical operands required by the application can be satisfied by the available physical registers without excessive spilling. This underscores the flexibility and adaptability of a reduced-size vector register file. Under the full-size VRF configuration, as shown in Figure 5.7, the true performance potential of LMUL becomes evident when the register constraints are removed. Although performance gains are observed across all LMUL configurations under these ideal conditions, the magnitude of improvement remains inherently limited by the datapath capabilities of the underlying hardware. Consequently, while LMUL is effective in enhancing performance in either constrained or unconstrained register environments, its benefits are ultimately bounded by the computational throughput of the execution units.

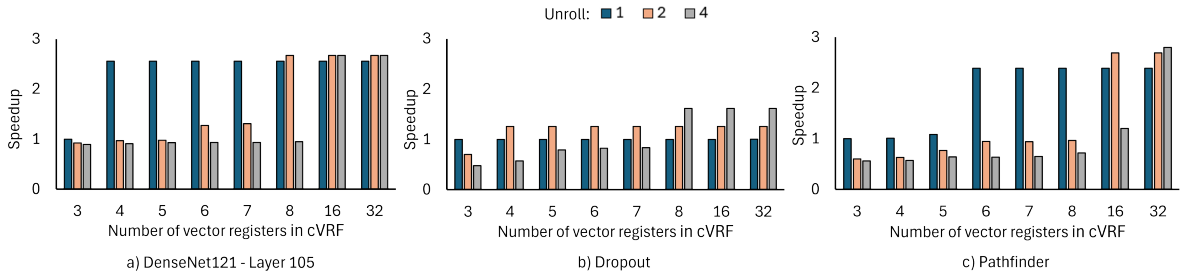


Figure 5.8: Performance speedup for different loop unroll factors (1, 2, and 4) evaluated on three applications: (a) DenseNet121 Layer 105, (b) Dropout, and (c) Pathfinder. The results are normalized to the baseline configuration (unroll=1) with three available vector registers in the cVRF.

Impact of loop unrolling

As shown in Figure 5.8, we evaluate the effect of software loop unrolling (unroll factors of 1, 2, and 4) across different numbers of available physical vector registers for DenseNet121 Layer 105, Dropout, and Pathfinder. The figure reports performance speedup normalized to the baseline configuration (unroll=1) with three available vector registers. A key trend is that aggressive unrolling is not universally beneficial under constrained register availability: with small cVRF sizes, higher unroll factors increase the number of simultaneously live vector values, which elevates register pressure and can trigger more frequent evictions and spill/fill activity, thereby limiting or reducing speedup relative to unroll=1. As the number of available vector registers increases, the negative impact of register pressure diminishes and unrolling becomes advantageous by reducing loop overhead and improving overlap, so unroll=2 and particularly unroll=4 achieve their highest speedups at larger register counts where the expanded working set can remain resident. Overall, Figure 5.8 highlights that unrolling is an effective complementary knob to LMUL, but its benefit is strongly workload- and capacity-dependent.

Quantifying worst-case stall scenarios under cVRF misses

A potential concern is the stall cost when operands are not resident in the cVRF and must be serviced through the Register Dispersion miss path. Each vector instruction may reference up to three operands (e.g., two sources and one destination). For each operand, the Tag Array is consulted to determine residency. Importantly, Tag Array lookup is performed in parallel in the decode stage and therefore does not add explicit stall cycles; stalls arise only when an operand is missing and must be fetched (and possibly when an eviction must be written back).

In the worst-case where all three operands miss and free slots exist in the cVRF, the processor performs three Tag Array lookups and three memory fetches, followed by three installs into the cVRF. The precise stall time depends on the memory hierarchy and whether transactions can overlap; therefore, it is not a fixed ISA-level constant. To provide a concrete upper-bound estimate consistent with our evaluation platform (Table 5.1), we assume conservative serialized handling in which each miss triggers a memory fetch that incurs the maximum memory latency of 5 cycles before the next operand service begins, **without accounting for cache-hit probability or overlap among memory transactions**. Under this assumption, the additional stall budget is dominated by the three fetches, i.e., approximately

Table 5.4: Number of instruction operand misses in the cVRF for three applications—(a) DenseNet121 Layer 105, (b) Dropout, and (c) Pathfinder—measured with an **8-slot** cVRF, evaluated under LMUL settings 1, 2, 4, and 8.

No. of instr. operands missing in cVRF (out of a total of 3)	DenseNet121 Layer105				Dropout				Pathfinder			
	LMUL				LMUL				LMUL			
	1	2	4	8	1	2	4	8	1	2	4	8
1	4	8	532865	135632	2	4	8	135632	6	52701	67815	135632
2	0	0	128	55800	0	0	0	55800	0	6200	18598	55800
3	0	0	76160	37976	0	0	0	37976	0	9300	15887	37976

Table 5.5: Number of instruction operand misses in the cVRF for three applications—(a) DenseNet121 Layer 105, (b) Dropout, and (c) Pathfinder—measured with an **8-slot** cVRF, evaluated under unroll factors 1, 2, and 4.

No. of instr. operands missing in cVRF (out of a total of 3)	DenseNet121 Layer105			Dropout			Pathfinder		
	Unroll			Unroll			Unroll		
	1	2	4	1	2	4	1	2	4
1	4	8	532865	2	4	8	6	52701	67815
2	0	0	128	0	0	0	0	6200	18598
3	0	0	76160	0	0	0	0	9300	15887

$3 \times 5 = 15$ cycles.

A more severe case occurs when the cVRF has no free entries and each missing operand requires replacing an existing entry. In this scenario, each operand service includes both a write-back (store) of the evicted entry and a fetch (load) of the required architectural register, resulting in three stores and three loads. Using the same conservative serialized assumption and the maximum 5-cycle memory latency per transaction (Table 5.1), the stall budget is approximately $(3\,\text{stores}+3\,\text{loads})\times 5 = 30$ cycles. This "double worst-case" is intentionally pessimistic, but it provides a concrete bound for maximal contention.

The Tag Array lookup is performed in parallel within the decode stage and is assumed to complete within the decode cycle; therefore, it does not contribute additional stall cycles. At the cycle level, stall time is dominated by miss servicing through the memory subsystem (and by write-backs when replacement is required), while increasing the Tag Array/cVRF size primarily affects performance by reducing the miss rate. This trend is quantified in Table 5.4 and Table 5.5, which report vector operand-miss counts for the three representative applications using an 8-slot cVRF under varying LMUL settings and varying loop-unroll factors, respectively. Importantly, the unroll factors and LMUL settings are directly comparable from the cVRF perspective: both increase the number of simultaneously live architectural vector registers that must be kept resident to sustain execution, thereby stressing the same limited set of physical cVRF slots; thus, matching values (e.g., 1/2/4) represent comparable pressure on the cVRF working set. For the evaluated workloads, the working sets largely fit within the reduced cVRF for modest settings, resulting in few misses. As register pressure increases (e.g., under more aggressive LMUL or unrolling), miss counts rise substantially, increasing the likelihood of miss-induced stalls and degrading performance. Finally, for Dropout, one of the frequently used operands is $\mathrm{v0}$, which we keep outside

the cVRF (always locally available); therefore, accesses to v0 do not incur cVRF misses and are not counted in the operand-miss statistics. Overall, Tables 5.4 and 5.5 quantify the key trade-off: higher register pressure increases the operand-miss rate—and thus the likelihood of miss-induced stalls—while enabling a smaller, lower-cost cVRF.

5.3.5 The effect of cache associativity on the dispersion mechanism

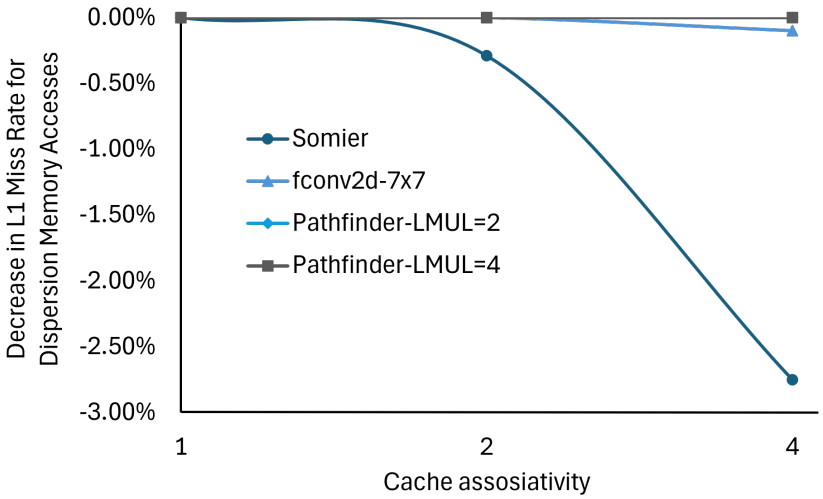


Figure 5.9: Reduction in L1 cache miss rate for memory accesses associated with the dispersion mechanism in the Somier, fconv2d-7x7, Pathfinder-LMUL=2, and Pathfinder-LMUL=4 applications.

Figure 5.9 presents the L1 miss rate of memory operations originating exclusively from the cVRF. In other words, the reported miss rate does not represent the total L1 data cache miss rate of the application, but rather the subset of misses generated when vector registers that cannot fit into the 8-slot cVRF are evicted to memory (cache or main memory) and subsequently reloaded. The total size of the L1 data cache remains unchanged throughout this evaluation; *only the associativity changes*.

To capture the impact of cache associativity (number of ways in each set) on the dispersion mechanism's miss rate, we evaluate four workloads that require more than 8 physical registers: Somier, fconv2d-7x7, Pathfinder-LMUL=2, and Pathfinder-LMUL=4. The two Pathfinder configurations allow us to examine how different LMUL settings influence the dispersion behavior. The Pathfinder-LMUL=1 case is omitted, as its requirement of six vector registers is less than the cVRF's capacity of eight registers, which means that no evictions occur.

The results show that, for both Pathfinder configurations (LMUL=2 and LMUL=4), the miss rate remains almost unchanged with varying associativity. This behavior stems from the fact that these workloads exhibit a broad variety of vector accesses per iteration, reducing the likelihood of repeatedly accessing the same registers and thus making associativity less influential.

In contrast, fconv2d-7x7 demonstrates a small decrease in the miss rate when the number of ways increases to four, while Somier shows a more pronounced reduction. This indicates that workloads with higher temporal locality in their evicted vector registers benefit more from increased associativity.

Overall, these results highlight that the dispersion mechanism's miss rate is workload-dependent,

with increased associativity providing benefits primarily when register eviction patterns repeatedly target a small working set. Nevertheless, one way to ensure that register values residing in the cache are never evicted to main memory is to tinker with the cache replacement policy. For example, assuming a Least-Recently Used (LRU) cache replacement policy, any way occupied by a register value could be *pinned* to the corresponding set's Most-Recently Used (MRU) position. This will ascertain that the register value will not be evicted if/when the set becomes full. Incoming new data to the same set will keep evicting the other ways, leaving the register value unaffected.

5.3.6 Hardware cost analysis

To assess the hardware area and power savings achieved by the Register Dispersion mechanism, we compare a CPU utilizing a compact Vector Register File (cVRF) capable of storing 8 vector registers with a vector length of 256 bits to an equivalent CPU employing a full-size VRF capable of storing all 32 vector registers. Both evaluated configurations operate at a clock frequency of 200 MHz, which falls within the typical range observed in low-cost processors [138]. It should be noted that register Dispersion logic (e.g., Tag Array checks) is not on the timing-critical path and it does not degrade the achievable clock frequency.

ASIC Area Savings

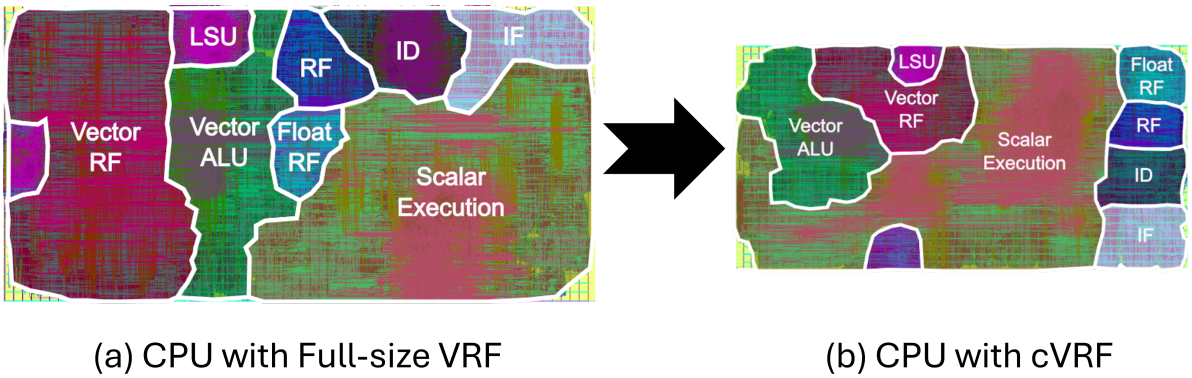


Figure 5.10: The physical layout of the two versions of the examined CPU. Highlighted regions include the Instruction Fetch (IF) logic, the Instruction Decode (ID) logic, the Load-Store Unit (LSU) of the scalar core, the scalar execution units, and the integer and floating-point register files. The VPU consists of the regions of the VRF and the Vector ALU. Compacting the VRF also altered the relative placement of the CPU modules.

The physical layout of both implementations at 28 nm is illustrated in Figure 5.10. The layout on the left corresponds to the CPU with a VPU that utilizes a full-size VRF, whereas the layout on the right side corresponds to the same CPU, but with its VPU designed using a cVRF. Both layouts focus only on the logic area of the CPU, excluding the L1 instruction and data caches. For both scalar and vector register files, we assume a flip-flop-based standard-cell-based implementation. Moving to latch-based designs, as done in [19], is possible and would equally benefit both the full-size VRF and the cVRF implementations.

Both layout photos highlight the main blocks of the scalar core and the vector engine. The scalar

Table 5.6: Comparisons with representative vector/SIMD designs featuring short vector lengths (VL $\leq$ 1024 bits).

ASIC	VL (bits)	#VRegs	#Lanes	Element widths				Tech	Freq. (MHz)	Area (mm^2)	Area @12nm (mm^2)	Area eff @12nm (VL/mm^2)
				64	32	16	8					
Proposed cVRF*	256	8	8	–	✓	✓	✓	28 nm	200	0.14	0.059	4339
Ahromaa [15]*	128	–	4	–	✓	–	–	65 nm	600	0.53	0.061	2098
AVA[72]	1024	64	8	✓	–	–	–	22 nm	1000	1.98	0.943	1086
Saturn [164]	512	32	4	✓	✓	✓	✓	16 nm	600	0.41	0.273	1875
RISC-V^2 [101]	128	32	4	–	✓	–	–	45 nm	1000	0.61	0.078	1641
Spatz [19]	512	32	4	✓	✓	✓	✓	12 nm	1000	0.58	0.58	883
Ariane-4VL [142]*	512	32	4	–	✓	✓	✓	12 nm	1000	0.15	0.15	3413
M-VU [126]	512	32	4	–	✓	✓	✓	65 nm	210	3.931	0.332	1542

*Area includes the scalar core in addition to the vector unit.

execution logic occupies the majority of the area, since it includes both an integer ALU with a 32-bit multiplier and divider and a complete single-precision floating point unit. The scalar CPU also includes two separate register files; one is used for integer operands, and the other for floating-point operands.

By reducing the number of stored registers from 32 to 8, the area occupied by the VRF decreases by 3.5×. The reduction does not reach 4×, because the proposed approach incurs the extra cost of the cVRF tag array and the Register Dispersion control unit. The reduction in the total VPU area (i.e., the VRF and the Vector ALU) is around 53%. This translates to a 23% area reduction in the *total processor area* (i.e., *entire* CPU+VPU setup). The additional area required to support the LMUL functionality is integrated within the Instruction Decode (ID) stage of the layout and results in a total area increase of less than 1%, as compared to an implementation without vector length multiplier support. The reported area savings are accompanied with a reduction in routing congestion, as a consequence of the VRF's wide read and write ports.

Given that our gains come primarily from reducing register-file footprint, we position our design within the broader space of vector architectures. We compare our RVV-based design against other representative state-of-the-art vector implementations spanning different design philosophies and cost–performance targets, with a focus on vector/SIMD designs with short vector lengths (VLEN $\leq$ 1024 bits).

Table 5.6 summarizes the main characteristics of the designs under comparison and shows that the proposed architecture achieves its low-cost objective, as demonstrated by the high area efficiency enabled by the use of the cVRF and the register dispersion mechanism. The table emphasizes differences in vector length, the number of vector registers, and vector length configurability, together with ASIC implementation metrics that exclude cache area. Note that, since our vector support is integrated into the scalar core, the reported area for our implementation includes the scalar core. Similarly, all competing designs whose reported areas also include the scalar core are marked with an asterisk in Table 5.6. For all other designs, only the vector-related area is reported.

The area efficiency metric reported in the last column of Table 5.6 expresses how much vector length, and therefore data level parallelism, each processor can support relative to the silicon area required for its implementation. Higher values correspond to more efficient use of area. To compute this metric, the area of each design is scaled to a common 12 nm technology node using DeepScale [120]. The results indicate that the proposed design minimizes the area usage for its supported vector length by reducing the size of the area-intensive VRF. In contrast, the other designs focus on high-performance operating

points with higher clock frequencies. Consequently, they trade area efficiency for increased throughput, which differs from the low-cost objective of the proposed architecture.

Application-level average power savings

Each examined application was executed on the RTL of both CPUs under comparison to estimate the corresponding switching activity. Non-ML applications were driven by uniformly random inputs, while ML applications were executed on relevant datasets. Subsequently, the average power consumption of each CPU while executing each application was estimated on the final extracted netlist after layout, including all interconnect parasitics. The comparison results of the per-application average power consumption are presented in Figure 5.11. The CPU utilizing a cVRF with register dispersion exhibits average power savings of 10%, as compared to the CPU with a full-size VRF.

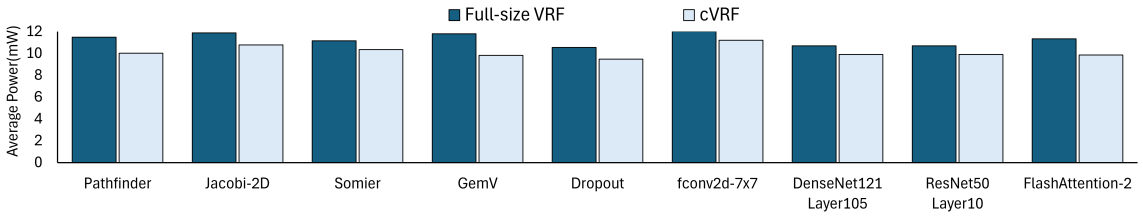


Figure 5.11: The average power consumption for all examined applications when using a CPU with a full-size VRF and when using the same CPU with the proposed cVRF.

The effect of vector length multiplication on power efficiency

To further investigate the relationship between the LMUL configurations and power efficiency, three representative applications – DenseNet121 - Layer 105, Dropout, and Pathfinder – were selected from the set of applications used in the earlier average power evaluation. The results are presented in Figure 5.12, where the left figure corresponds to the baseline configuration with a full-size VRF, and the right figure corresponds to a cVRF containing eight physical vector registers.

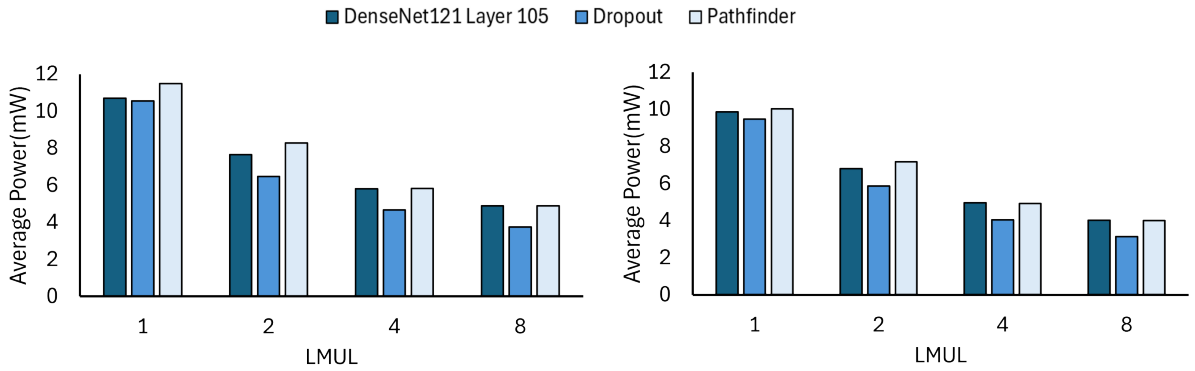


Figure 5.12: The effect of vector length multiplication on average power consumption when using a CPU with a full-size VRF and when using the same CPU with the proposed cVRF.

In both configurations, increasing the LMUL value beyond 1 consistently reduces the average power consumption, with the lowest power typically observed at LMUL=8. This effect arises from the reduced

number of dynamic instructions issued, since larger LMUL values enable the processor to operate on a greater number of data elements per instruction. Regardless, the decrease in average power consumption does not scale proportionally with the reduction in the total instruction count (e.g., the reductions from LMUL=2 to LMUL=4 and from LMUL=4 to LMUL=8 show diminishing returns compared to the transition from LMUL=1 to LMUL=2). This discrepancy is primarily due to two factors. First, larger LMUL values tend to increase the number of memory accesses because wider operand groups must be fetched, which raises the dynamic power consumption in the cache hierarchy and memory subsystem. Second, when the vector datapath width is smaller than the operand width dictated by the LMUL setting, each vector instruction must be executed over multiple steps. This increases the switching activity within the vector execution unit, thereby raising the per-instruction power cost.

Consequently, while an ideal linear scaling would suggest that average power should drop by approximately $2\times$ for LMUL=2, $4\times$ for LMUL=4, and $8\times$ for LMUL=8, the measured reductions are noticeably smaller in practice. These results indicate that LMUL tuning can indeed be an effective strategy for lowering power consumption, but the achievable benefits are limited by the additional memory activity and the constraints imposed by the vector datapath width.

In summary, the proposed mechanism is very effective in reducing the hardware cost of the VPU, as compared to using a full-size VRF. Most importantly, these hardware savings are achieved with no/minimal negative impact on performance.

FPGA implementations

To complement the analysis of ASIC implementations, we implemented the proposed design using a cVRF with 8 physical registers on a Virtex-7 FPGA and compared it against a design with a full-sized VRF. Both designs operate at 75 MHz. Table 5.7 reports the LUT, FF, DSP, and BRAM utilization for the two implementations.

The two designs use an identical number of BRAMs for cache implementation, as well as the same number of memory LUTs consumed by scalar CPU subsystems, specifically the branch target buffer, the instruction fetch buffer, and the integer and floating-point scalar register files. These cache and memory structures are unchanged between the two designs, which explains why their resource usage is identical in both cases.

The same also holds for the DSP usage, i.e., DSP utilization is identical in both designs. Of these, 24 DSPs are consumed by the vector lanes, indicating that each lane uses three DSPs (i.e., 8 lanes $\times$ 3 DSPs/lane). The remaining DSPs are associated with the scalar core and therefore do not change with the VRF organization.

Table 5.7: FPGA resource breakdown for our Virtex-7 FPGA for the cVRF and full-size VRF implementations.

Design	LUT		FF	DSP	BRAM		Maximum VL (bits)
	Logic	Memory			Cache	VRF	
cVRF	55878	256	17652	30	43	12	18432
Full-size VRF	55151	256	17022	30	43	12	4608

Although the proposed design operates with only one quarter of the vector registers compared to a vector processor with a full-sized VRF, both designs use the same number of BRAMs to implement the VRF. This behavior can be explained as follows. The VRFs are mapped to RAMB36E1 BRAMs configured in simple dual-port (SDP) mode with a capacity of 512×72 bits. Under this configuration, a 256-bit vector register that must be read entirely in parallel is partitioned across $\lceil 256/72 \rceil = 4$ BRAMs. Since SDP mode provides only one read port per physical BRAM instance, supporting the three independent read ports required by the RVV ISA necessitates replication by a factor of three. As a result, the VRF implementation requires $4 \times 3 = 12$ BRAMs, as reported in Table 5.7.

Nevertheless, this behavior highlights an opportunity for better BRAM efficiency with cVRF. By reducing the number of physical registers, the same BRAM space can be redistributed into *more storage per register* without increasing the required per-cycle read bandwidth. This property is exploited by designs that support a *logical* vector length exceeding the available hardware parallelism (e.g., designs that advertise very wide VL, but execute it in multiple steps [72]). In such cases, portions of the vector register may be *time-multiplexed* ("wrapped around") into the same BRAM, because the entire vector is not accessed in parallel. Under this execution model, a cVRF with fewer registers can become advantageous: for a fixed BRAM budget, reducing the number of resident registers increases the BRAM capacity available per register, thereby increasing the *achievable* (resident) vector length, as shown in Table 5.7.

5.4 Related Work

The elemental concept of reduced-size register file has also been exploited in other computing domains [9, 1]. The targeted domains so far have been superscalar processors [9] and Graphics Processing Units (GPU) [1]. Specifically, the design proposed in [9] aims to reduce the latency and energy requirements of the register file of a superscalar core for similar IPC performance. By using a *dedicated two-level* and banked register file organization, the proposed approach reduces the register-file size and the port requirements. On the contrary, our work uses a smaller, *single-level* register file, and it relies on the *existing* levels of the cache/memory sub-system to manage the spill-over registers. Similarly, the Bicameral Cache design [114] proposes a split scalar/vector cache organization with tailored line sizes and memory-side prefetching, eliminating interference between scalar and vector accesses. While the Bicameral Cache introduces a specialized additional cache structure to enhance data locality, our approach remains simpler by leveraging the unmodified existing cache/memory hierarchy.

The design in [1] reduces the dynamic energy consumption of GPUs by utilizing energy-efficient small caches between the register file and the execution units. By using a hierarchical register-file architecture, certain registers are cached within existing operand buffering structures that are located closer to the execution pipelines of the GPU. Once again, this approach is founded on a two-level organization, as opposed to our work that uses all the levels of the existing cache/memory hierarchy.

Low-cost processors and vector processing units have been the subject of significant research, particularly in the domain of embedded systems and the Internet-of-Things (IoT) [58, 88], where resource constraints and energy efficiency are critical. The significance of the hardware cost of the VRF, in particular, was recognized in [19]. Specifically, the Spatz vector processor presented in [19] demonstrates

how compact vector units with minimal VRFs can deliver high efficiency while leveraging the RVV Extension. In contrast to the cVRF approach proposed in this paper, Spatz uses a full 32-vector VRF and lowers its hardware cost though the use of optimized latch-based Standard Cell Memory (SCM). With superior area and energy efficiency, the Spatz architecture is well-suited for edge and IoT applications. Nevertheless, this approach is seen as orthogonal to our technique, and – as mentioned in Section 5.3.6 – it could be used to further lower the hardware cost of the cVRF. Building on Spatz, TROOP [109] applies micro-architectural optimizations – such as decoupled load/store interfaces, improved vector chaining, and shadow buffers – to achieve near-roofline performance for memory-intensive kernels. While TROOP focuses on maximizing bandwidth utilization and latency-hiding within the VPU, our work targets VRF cost reduction for low-cost processors. The work in [135] proposed the integration of vector units within a low-cost scalar processor, but no hardware optimizations were made pertaining to the VPU.

Other studies have concentrated on optimizing memory-access mechanisms for vector processors. For instance, EARTH [40] proposes shifting-based data reorganization techniques to efficiently manage constant-stride and segmented memory access patterns in RISC-V vector architectures, thereby eliminating the need for large segment buffers and reducing both area and power consumption. In contrast, the work presented in this manuscript focuses on reducing the VRF size and integrating vector length multiplication.

SIMD units are often preferred over vector processors, especially for embedded systems, due to their simpler design. SIMD processing has also been used for multimedia applications on mobile devices [7, 103]. The Sparrow architecture [17] leverages the scalar integer register file for SIMD operations and uses a reduced number of SIMD instructions to achieve area savings.

5.5 Conclusion

This chapter demonstrated that Register Dispersion, implemented through a compact Vector Register File (cVRF), is an effective microarchitectural technique for reconciling the high throughput demands of vectorized ML workloads with the strict area and power constraints of low-cost processors. By exploiting the observation that many workloads use only a small live set of vector registers at any given time, the proposed design substantially reduces VRF area and energy without sacrificing ISA compliance or vector width. In this way, the chapter showed that reducing the number of physical vector registers can be a practical means of preserving the benefits of vector execution in cost-sensitive RVV systems.

The chapter also extended this compact-register organization to support RVV vector register grouping through LMUL. This is achieved through a lightweight instruction-sequencing mechanism that allows grouped vector operands to be executed correctly even when only a limited number of physical register slots are available locally. The results showed that this added flexibility can be supported with very small hardware overhead, while also revealing the associated trade-off: larger LMUL settings increase register pressure and may introduce additional spill/fill traffic when the cVRF capacity is limited.

Taken together, the two parts of this chapter establish both the core value and the practical maturity of Register Dispersion. The first is that a cache-like cVRF can preserve most of the performance of a conventional VRF while greatly reducing storage cost. The second is that this organization can be

extended to support full RVV functionality, including grouped registers, and can sustain not only isolated kernels but also complete end-to-end ML workloads such as CNN and transformer-based LLM inference. The reported ASIC area and power results, together with FPGA resource measurements, further confirm that the proposed approach is practical across both custom and reconfigurable implementations.

The results also highlight several important design trade-offs that merit further exploration. While a FIFO-based replacement policy proves sufficient for achieving high hit rates in many workloads, more adaptive policies could further reduce miss rates under high register pressure, particularly when aggressive LMUL settings or loop unrolling are employed. Similarly, although instruction sequencing enables LMUL support with minimal complexity, it introduces serialization that can limit performance scaling when the register working set exceeds cVRF capacity. These trade-offs suggest opportunities for selectively increasing cVRF capacity or co-designing compiler heuristics that account for physical register availability when selecting LMUL and unrolling factors.

Chapter 6

Conclusions and Future Work

6.1 Summary

This thesis set out to solve a fundamental tension in modern computing: how to deliver the immense data-parallel throughput required by machine-learning workloads without exceeding the strict area and power budgets of low-cost processors. The findings presented in this work demonstrate that true energy efficiency cannot be achieved through isolated optimizations. Instead, it requires a holistic, full-stack co-design approach spanning the underlying microarchitecture, the instruction set, and the algorithmic kernel mappings.

By systematically addressing the most expensive aspects of vector execution, this thesis makes several core contributions that fundamentally alter the design tradeoffs for energy-efficient vector processors:

Redefining the Cost of Vector Register file Historically, the VRF has been an unavoidable barrier to entry for low-cost vector processing due to its massive area and energy footprint. The introduction of **Register Dispersion** fundamentally challenges the assumption that a monolithic VRF is required. By observing the temporal locality of vector registers in machine-learning kernels and transforming the VRF into a demand-driven, cache-like physical structure, this work achieves a 53% reduction in total Vector Processing Unit area and significant power savings. The true impact of this contribution is that it decouples physical silicon cost from architectural vector length, making highly expressive vector processing viable for strictly constrained embedded environments without breaking ISA compliance.

Streamlining Complex Vector Data Relocation Data permutation is a notorious bottleneck in real-world vectorized workloads, often leading to multi-cycle stalls or bloated silicon area. The development of a **Unified Vector Permutation Unit** solves this by mapping both input- and output-driven permutation patterns onto a single crossbar-based structure. The impact here is a significant simplification of the execution backend. By eliminating the need for isolated, bespoke datapaths, this thesis turns complex within-vector data movement into a predictable, low-overhead primitive.

Accelerating Sparse Linear Algebra For structured-sparse applications, conventional vector execution often struggles with indirect memory accesses and poor data reuse. This thesis proves that carefully

architected data placement can overcome these limitations. By keeping sparse indices on the scalar side and utilizing the minimal IndexMAC fused instruction, the proposed approach drastically reduces memory traffic and maximizes dense-operand reuse within the VRF. This demonstrates a highly impactful design philosophy: minimal, tightly targeted ISA extensions combined with smart loop unrolling can deliver accelerator-like performance for sparse CNNs while fully preserving the programmability of the baseline vector processor.

Enabling Efficient Transformer Execution Finally, this work proves that advanced large language models do not strictly require specialized, fixed-function tensor hardware. The fully RVV-compliant vectorization of **FlashAttention-2** demonstrates that attention mechanisms can be mapped highly efficiently to a standard vector pipeline. By introducing vector-length-agnostic tiling and a lightweight exponential approximation for the softmax bottleneck, this contribution heavily reduces control overhead and compute latency. The impact is a proven, scalable blueprint for executing complex Transformer models on modest, cost-sensitive edge processors.

6.2 Future work

Building upon the architectural principles and optimizations established in this thesis, several compelling avenues for future research emerge to further scale the capabilities of energy-efficient vector processors.

Native Support for Microscaling and Sub-Byte Data Formats As machine learning models continue to grow, future research must address the shift toward micro-scaled and sub-byte data formats to further increase computational density and alleviate memory bandwidth bottlenecks. Emerging industry standards, such as the Microscaling Formats (MX) including MXFP6, FP4, and block floating-point representations, group multiple low-precision values under a shared scaling factor. Vector architectures are uniquely positioned to exploit these block-based formats since they inherently operate on contiguous sets of data elements. Future investigations should focus on modifying vector execution pipelines to natively decode and process shared micro-exponents across parallel lanes, enabling high-throughput mixed-precision accumulation without incurring costly conversion overheads. Furthermore, integrating sub-byte data types introduces new opportunities to amplify the benefits of the architectural mechanisms proposed in this work. For instance, densely packing micro-scaled operands could effectively multiply the logical capacity of a dispersed vector register file, allowing highly complex neural networks to remain resident on-chip even in severely area-constrained edge devices.

Vector Architectures for Privacy-Preserving Computing and Homomorphic Encryption Beyond standard machine-learning workloads, emerging privacy-preserving paradigms such as Fully Homomorphic Encryption (FHE) represent a compelling application domain for vector architectures. FHE schemes (e.g., CKKS and BGV) allow computation directly on encrypted ciphertexts, but they are severely bottlenecked by operations on massive polynomials of high degree ($N \geq 2^{14}$). These workloads rely heavily on modular arithmetic, Number Theoretic Transforms (NTT), and automorphism permutations, all of which exhibit abundant data-level parallelism. Future research could explore how the unified permutation unit

and configurable execution pipelines developed in this work can be specialized for NTT butterfly networks, modular reduction, and key-switching permutations. Furthermore, adapting Register Dispersion to handle the memory footprint of massive cryptographic polynomials without exhausting on-chip storage could significantly narrow the performance and energy overheads of privacy-preserving edge inference.

Event-Driven Vector Processing for Spiking Neural Networks Another promising direction lies in bridging vector processing with neuromorphic computing through the efficient emulation of Spiking Neural Networks (SNNs). Unlike conventional neural networks that process dense continuous values, SNNs rely on sparse, temporal, and binary event-driven activations. Future research can investigate how programmable vector processors can be optimized for the unique dynamics of biological models, such as Leaky Integrate-and-Fire neurons. This involves extending vector ISAs and datapaths to natively support fine-grained temporal masking, sub-word bit manipulation, and zero-overhead conditional updates for membrane potentials. By coupling structured sparsity techniques with low-cost indexing mechanisms akin to IndexMAC, a compact vector processor could efficiently simulate large populations of spiking neurons with high temporal resolution and minimal energy consumption.

Ultra-Low-Cost Vector Engines on Native Sensors and Plastic Substrates From a physical implementation standpoint, extending energy-efficient vector processing to non-traditional, extremely low-cost technologies such as flexible electronics and plastic substrates opens new frontiers for ubiquitous computing. Printed and organic semiconductor technologies enable disposable smart packaging, conformable healthcare patches, and environmental sensors, but they operate under extreme constraints of low carrier mobility, low gate density, and ultra-tight power budgets. Future research could explore the design of ultra-minimalist vector processing engines specifically tailored to flexible substrates. In such severely area-constrained environments, techniques like Register Dispersion become foundational enablers. By combining near-threshold circuit techniques with highly parameterized, minimal-gate-count vector lanes, programmable data-parallel intelligence could be embedded directly onto flexible substrates at the outermost boundaries of the sensing edge.

Bibliography

[1] Mojtaba Abaie Shoushtary, Jose Maria Arnau, Jordi Tubella Murgadas, and Antonio Gonzalez. "Lightweight Register File Caching in Collector Units for GPUs". In: *Workshop on General Purpose Processing Using GPU*. ACM, 2023, pp. 27–33.

[2] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, et al. "Phi-4 technical report". In: *arXiv preprint arXiv:2412.08905* (2024).

[3] Tutu Ajayi et al. "Toward an open-source digital flow: First learnings from the openroad project". In: *Design Automation Conference (DAC)*. 2019.

[4] Guillermo Alaejos et al. "Micro-kernels for portable and efficient matrix multiplication in deep learning". In: *The Journal of Supercomputing* 79.7 (2023), pp. 8124–8147.

[5] Kosmas Alexandridis, Vasileios Titopoulos, and Giorgos Dimitrakopoulos. "Low-Cost FlashAttention with Fused Exponential and Multiplication Hardware Operators". In: *arXiv preprint arXiv:2505.14314* (2025).

[6] *AMBA 5 CHI Architecture Specification*. ARM, 2018.

[7] Hossein Amiri and Asadollah Shahbahrami. "High performance implementation of 2-D convolution using AVX2". In: *Intern. Symposium on Computer Architecture and Digital Systems (CADS)*. 2017, pp. 1–4.

[8] Krste Asanovic. *Vector microprocessors*. University of California, Berkeley, 1998.

[9] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi. "Reducing the complexity of the register file in dynamic superscalar processors". In: *IEEE Intern. Symp. on Microarchitecture (MICRO)*. 2001, pp. 237–248.

[10] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. "Mlperf tiny benchmark". In: *arXiv preprint arXiv:2106.07597* (2021).

[11] Eric Bavier, Nicholas Knight, Hugues de Lassus Saint-Geniès, and Eric Love. "Vectorized nonlinear functions with the RISC-V vector extension". In: *IEEE Symposium on Computer Arithmetic (ARITH)*. 2023, pp. 127–130.

[12] Nathan Bell, Steven Dalton, and Luke N Olson. "Exposing fine-grained parallelism in algebraic multigrid methods". In: *SIAM Journal on Scientific Computing* 34.4 (2012), pp. C123–C152.

[13] Iz Beltagy, Matthew E Peters, and Arman Cohan. “Longformer: The long-document transformer”. In: *arXiv preprint arXiv:2004.05150* (2020).

[14] Nathan Binkert et al. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (2011).

[15] N. Bleier, A. Wezelis, L. Varshney, and R. Kumar. “Programmable Olfactory Computing”. In: *IEEE Micro* 44.04 (2024), pp. 88–96.

[16] James F Blinn. “Floating-point tricks”. In: *IEEE Computer Graphics and Applications* 17.4 (1997), pp. 80–84.

[17] Marc Solé Bonet and Leonidas Kosmidis. “SPARROW: A Low-Cost Hardware/Software Co-designed SIMD Microarchitecture for AI Operations in Space Processors”. In: *Design, Automation & Test in Europe (DATE)*. 2022, pp. 1139–1142.

[18] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. “Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age”. In: *IEEE Transactions on robotics* 32.6 (2017), pp. 1309–1332.

[19] Matheus Cavalcante, Matteo Perotti, Samuel Riedel, and Luca Benini. “Spatz: Clustering Compact RISC-V-Based Vector Units to Maximize Computing Efficiency”. In: *arXiv preprint arXiv:2309.10137* (2023).

[20] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. “Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI”. In: *IEEE Trans. on VLSI Systems* 28.2 (2019), pp. 530–543.

[21] Chen Chen et al. “Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product”. In: *ACM/IEEE Intern. Symp. on Comp. Arch. (ISCA)*. 2020, pp. 52–64.

[22] Zhaodong Chen, Zheng Qu, Yuying Quan, Liu Liu, Yufei Ding, and Yuan Xie. “Dynamic N:M fine-grained structured sparse attention mechanism”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2023, pp. 369–379.

[23] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. “Generating long sequences with sparse transformers”. In: *arXiv preprint arXiv:1904.10509* (2019).

[24] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. “Visual wake words dataset”. In: *arXiv preprint arXiv:1906.05721* (2019).

[25] Codasip. *L31 RISC-V embedded processor*. 2022. URL: `https://codasip.com/products/low-power-embedded-risc-v-processors/l31`.

[26] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. “Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities”. In: *arXiv preprint arXiv:2507.06261* (2025).

[27] Shengkun Cui, Archit Patke, Hung Nguyen, Aditya Ranjan, Ziheng Chen, Phuong Cao, Gregory Bauer, Brett Bode, Catello Di Martino, Saurabh Jha, et al. “Story of Two GPUs: Characterizing the Resilience of Hopper H100 and Ampere A100 GPUs”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. 2025, pp. 1145–1164.

[28] Tri Dao. “Flashattention-2: Faster attention with better parallelism and work partitioning”. In: *arXiv preprint arXiv:2307.08691* (2023).

[29] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. “Flashattention: Fast and memory-efficient exact attention with IO-awareness”. In: *Advances in neural information processing systems* 35 (2022), pp. 16344–16359.

[30] César Debeunne and Damien Vivet. “A review of visual-LiDAR fusion based simultaneous localization and mapping”. In: *sensors* 20.7 (2020), p. 2068.

[31] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. “Imagenet: A large-scale hierarchical image database”. In: *IEEE conference on computer vision and pattern recognition*. 2009, pp. 248–255.

[32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *North American Chapter of the Association for Computational Linguistics*. 2019.

[33] R. Espasa et al. “Tarantula: a vector extension to the alpha architecture”. In: *Intern. Sympos. on Comp. Arch. (ISCA)*. 2002, pp. 281–292.

[34] Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. “Rigging the lottery: Making all tickets winners”. In: *Inter. Conf. on Machine Learning*. 2020, pp. 2943–2952.

[35] Zitao Fang. “LEM: A Configurable RISC-V Vector Unit Based on Parameterized Microcode Expander”. In: (2022).

[36] Daniele Folegnani and Antonio González. “Energy-effective issue logic”. In: *Proceedings of the 28th annual international symposium on Computer architecture*. 2001, pp. 230–239.

[37] Joe Gebis and David Patterson. “Embracing and extending 20th-century instruction set architectures”. In: *Computer* 40.4 (2007), pp. 68–75.

[38] John R Gilbert, Cleve Moler, and Robert Schreiber. “Sparse matrices in MATLAB: Design and implementation”. In: *SIAM journal on matrix analysis and applications* 13.1 (1992), pp. 333–356.

[39] Felix Gremse, Andreas Höfter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. “GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging”. In: *SIAM Journal on Scientific Computing* 37.1 (2015), pp. C54–C71.

[40] Hongyi Guan, Yichuan Gao, Chenlu Miao, Haoyang Wu, Hang Zhu, Mingfeng Lin, and Huayue Liang. “Efficient Architecture for RISC-V Vector Memory Access”. In: *arXiv preprint arXiv:2504.08334* (2025).

[41] Udit Gupta et al. “ The Architectural Implications of Facebook’s DNN-Based Personalized Recommendation ”. In: *IEEE Inter. Symp. on High Performance Computer Architecture (HPCA)*. 2020, pp. 488–501.

[42] Fred G Gustavson. “Two fast algorithms for sparse matrices: Multiplication and permuted transposition”. In: *ACM Trans. on Mathematical Software (TOMS)* 4.3 (1978), pp. 250–269.

[43] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. “A3: Accelerating Attention Mechanisms in Neural Networks with Approximation”. In: *IEEE Intern. Symp. on High-Performance Computer Architecture (HPCA)*. 2020, pp. 328–341.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep residual learning for image recognition”. In: *IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*. 2016.

[45] Raymond Heald, Ken Shin, Vinita Reddy, I-Feng Kao, Masood Khan, William L Lynch, Gary Lauterbach, and Joe Petolino. “64-KByte sum-addressed-memory cache with 1.6-ns cycle and 2.6-ns latency”. In: *IEEE Journal of Solid-State Circuits* 33.11 (1998), pp. 1682–1689.

[46] John L Hennessy and David A Patterson. “A new golden age for computer architecture”. In: *Communications of the ACM* 62.2 (2019), pp. 48–60.

[47] Moisés Hernández, Ginés D Guerrero, José M Cecilia, José M García, Alberto Inuggi, Saad Jbabdi, Timothy EJ Behrens, and Stamatios N Sotiropoulos. “Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using GPUs”. In: *PloS one* 8.4 (2013), e61892.

[48] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. “Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks”. In: *The Journal of Machine Learning Research* 22.1 (2021), pp. 10882–11005.

[49] Mark Horowitz. “1.1 computing’s energy problem (and what we can do about it)”. In: *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*. Ieee. 2014, pp. 10–14.

[50] Kaixi Hou, Hao Wang, and Wu-chun Feng. “A framework for the automatic vectorization of parallel sort on x86-based processors”. In: *IEEE Trans. on Parallel and Distributed Systems* 29.5 (2018), pp. 958–972.

[51] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).

[52] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. “Densely connected convolutional networks”. In: *IEEE Conf. on Comp. Vision and Pattern Recogn. (CVPR)*. 2017, pp. 4700–4708.

[53] Ning-Chi Huang et al. “ELSA: Exploiting Layer-wise N:M Sparsity for Vision Transformer Acceleration”. In: *IEEE Conf. on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2024, pp. 8006–8015.

[54] Dan Andrei Iliescu and Francesco Petrogalli. *ARM Scalable Vector Extension and application to Machine Learning*. Tech. rep. ARM whitepaper, 2017.

[55] Intel. *Intel Intrinsics Guide*. 2024.

[56] Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. "MnnFast: a fast and scalable system architecture for memory-augmented neural networks". In: *Intern. Symp. on Computer Architecture (ISCA)*. 2019, pp. 250–263.

[57] Peng Jian, Lihan Hu, and Shihui Song. "Exposing and Exploiting Fine-Grained Block Structures for Fast and Accurate Sparse Training". In: *Advances in Neural Information Processing Systems*. 2022.

[58] Matthew Johns and Tom J Kazmierski. "A minimal RISC-V vector processor for embedded systems". In: *2020 Forum for Specification and Design Languages (FDL)*. IEEE. 2020, pp. 1–4.

[59] Robert F Jones Jr and Earl E Swartzlander Jr. "Parallel counter implementation". In: *Journal of VLSI Signal Processing Systems* 7.3 (1994), pp. 223–232.

[60] Marc Jorda, Pedro Valero-Lara, and Antonio J Pena. "cuConv: CUDA implementation of convolution for CNN inference". In: *Cluster Computing* 25.2 (2022), pp. 1459–1473.

[61] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings". In: *International symposium on computer architecture*. 2023, pp. 1–14.

[62] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit". In: *International symposium on computer architecture*. 2017, pp. 1–12.

[63] Andrej Karpathy. *Llama 2 LLM architecture inference in C*. 2023. URL: `https://github.com/karpathy/llama2.c`.

[64] Andrej Karpathy. *tinyllamas: stories260K*. 2023. URL: `https://huggingface.co/karpathy/tinyllamas/tree/main/stories260K`.

[65] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. "Transformers are rnns: Fast autoregressive transformers with linear attention". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5156–5165.

[66] Ben Keller, Rangharajan Venkatesan, Steve Dai, Stephen G Tell, Brian Zimmer, Charbel Sakr, William J Dally, C Thomas Gray, and Brucek Khailany. "A 95.6-TOPS/W Deep Learning Inference Accelerator With Per-Vector Scaled 4-bit Quantization in 5 nm". en. In: *IEEE Journal of Solid-State Circuits* 58.4 (2023), pp. 1129–1141.

[67] Sehoon Kim et al. "I-bert: Integer-only bert quantization". In: *International conference on machine learning*. PMLR. 2021, pp. 5506–5518.

[68] Nazim Altar Koca, Anh Tuan Do, and Chip-Hong Chang. “Hardware-efficient Softmax Approximation for Self-Attention Networks”. In: *Intern. Symp. on Circuits and Systems (ISCAS)*. 2023, pp. 1–5.

[69] Jack Koenig et al. “A hardware accelerator for computing an exact dot product”. In: *IEEE Symposium on Computer Arithmetic (ARITH)*. 2017, pp. 114–121.

[70] Alex Krizhevsky, Geoffrey Hinton, et al. *Learning multiple layers of features from tiny images*. 2009.

[71] Chris Lattner. “LLVM and Clang: Advancing compiler technology”. In: *Free and Open Source Devs’ European Meeting (FOSDEM)* (2011).

[72] Cristobal Ramirez Lazo, Enrico Reggiani, Carlos Rojas Morales, Roger Figueras Bague, Luis A. Villa Vargas, Marco A. Ramirez Salinas, Mateo Valero Cortes, Osman Sabri Unsal, and Adrian Cristal. “Adaptable Register File Organization for Vector Processors”. In: *IEEE Inter. Symp. on High-Performance Computer Architecture (HPCA)* (2021), pp. 786–799.

[73] Valentin Le Fèvre and Marc Casas. “Efficient Execution of SpGEMM on Long Vector Architectures”. In: *Intern. Symp. on High-Performance Parallel and Distributed Computing (HPDC)*. 2023, pp. 101–113.

[74] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (2002), pp. 2278–2324.

[75] Yunsup Lee. *Decoupled vector-fetch architecture with a scalarizing compiler*. University of California, Berkeley, 2016.

[76] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. “Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators”. In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–38.

[77] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and K Asanovic. “The hwacha microarchitecture manual, version 3.8”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-263* (2015).

[78] Huimin Li, Nele Mentens, and Stjepan Picek. “Maximizing the potential of custom RISC-V vector extensions for speeding up SHA-3 Hash functions”. In: *Design, Automation & Test in Europe Conference (DATE)*.

[79] Jiayu Li, Fugang Wang, Takuya Araki, and Judy Qiu. “Generalized sparse matrix-matrix multiplication for vector engines and graph applications”. In: *IEEE Workshop on Memory Centric High Perf. Comp.* 2019, pp. 33–42.

[80] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. “Memory-efficient Patch-based Inference for Tiny Deep Learning”. In: *Advances in Neural Information Processing Systems*. Vol. 34. 2021, pp. 2346–2358.

[81] Hao Liu and Pieter Abbeel. “Blockwise parallel transformers for large context models”. In: *Advances in neural information processing systems* 36 (2023), pp. 8828–8844.

[82] Zheng Liu, Meng Hao, Weizhe Zhang, Gangzhao Lu, Xueyang Tian, Siyu Yang, Mingdong Xie, Jie Dai, Chenyu Yuan, Desheng Wang, et al. “Optimizing depthwise separable convolution on DCU: Z. Liu et al.” In: *CCF transactions on high performance computing* 6.6 (2024), pp. 646–664.

[83] Zhi-Gang Liu, Paul N Whatmough, Yuhao Zhu, and Matthew Mattina. “S2TA: Exploiting structured sparsity for energy-efficient mobile CNN acceleration”. In: *IEEE Inter. Symp. on High-Performance Comp. Arch. (HPCA)*. 2022, pp. 573–586.

[84] Zhigang Liu, Matthew Mattina, Paul Whatmough, and Jesse Garrett BEU. *Processor for sparse matrix computation*. US Patent 11,392,376. 2022.

[85] Jason Lowe-Power et al. *The gem5 Simulator: Version 20.0+*. arXiv cs.AR 2007.03152. 2020.

[86] Siyuan Lu, Meiqi Wang, Shuang Liang, Jun Lin, and Zhongfeng Wang. “Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer”. In: *IEEE Intern. System-on-Chip Conference (SOCC)*. 2020, pp. 84–89.

[87] William L Lynch, Gary Lauterbach, and Joseph I Chamdani. “Low load latency through sum-addressed memory (SAM)”. In: *IEEE/ACM Intern. Symp. on Computer Architecture (ISCA)*. 1998, pp. 369–379.

[88] Malith Maheepala, Matthew A. Joordens, and Abbas Z. Kouzani. “Low Power Processors and Image Sensors for Vision-Based IoT Devices: A Review”. In: *IEEE Sensors Journal* 21.2 (2021), pp. 1172–1186.

[89] Alberto Marchisio, Davide Dura, Maurizio Capra, Maurizio Martina, Guido Masera, and Muhammad Shafique. “SwiftTron: An efficient hardware accelerator for quantized transformers”. In: *Intern. Joint Conference on Neural Networks (IJCNN)*. 2023, pp. 1–9.

[90] Maxim Milakov and Natalia Gimelshein. “Online normalizer calculation for softmax”. In: *arXiv preprint arXiv:1805.02867* (2018).

[91] Jung Gyu Min, Dongyun Kam, Younghoon Byun, Gunho Park, and Youngjoo Lee. “Energy-Efficient RISC-V-Based Vector Processor for Cache-Aware Structurally-Pruned Transformers”. In: *IEEE/ACM Intern. Symp. on Low Power Electronics and Design (ISLPED)*. 2023.

[92] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. “Large language models: A survey”. In: *arXiv preprint arXiv:2402.06196* (2024).

[93] Francesco Minervini and O Palomar. “Vitruvius: And area-efficient RISC-V decoupled vector accelerator for high performance computing”. In: *RISC-V Summit* (2021).

[94] Francesco Minervini, Oscar Palomar, Osman Unsal, Enrico Reggiani, Josue Quiroga, Joan Marimon, Carlos Rojas, Roger Figueras, Abraham Ruiz, Alberto Gonzalez, et al. “Vitruvius+: an area-efficient RISC-V decoupled vector coprocessor for high performance computing applications”. In: *ACM Trans. on Architecture and Code Optimization* 20.2 (2023), pp. 1–25.

[95] Asit Mishra et al. “Accelerating sparse deep neural networks”. In: *arXiv preprint arXiv:2104.08378* (2021).

[96] John N Mitchell. “Computer multiplication and division using binary logarithms”. In: *IRE Trans. on Electr. Computers* 4 (1962), pp. 512–517.

[97] Amir Morad, Leonid Yavits, and Ran Ginosar. “Efficient Dense and Sparse Matrix Multiplication on GP-SIMD”. In: *Intern. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2014.

[98] Arslan Munir, Joonho Kong, and Mahmood Azhar Qureshi. *Accelerators for convolutional neural networks*. John Wiley & Sons, 2023.

[99] Tan Nguyen et al. “FPGA-based HPC accelerators: An evaluation on performance and energy efficiency”. In: *Concurrency and Computation: Practice and Experience* 34.20 (2022), e6570.

[100] Zhixin Pan and Prabhat Mishra. “Hardware acceleration of explainable machine learning using tensor processing units”. In: *arXiv preprint arXiv:2103.11927* (2021).

[101] Kariofyllis Patsidis, Chrysostomos Nicopoulos, Georgios Ch. Sirakoulis, and Giorgos Dimitrakopoulos. “RISC-V^2: A Scalable RISC-V Vector Processor”. In: *IEEE Intern. Symp. on Circuits and Systems (ISCAS)*. 2020.

[102] Julian Pavon et al. “VIA: A Smart Scratchpad for Vector Units with Application to Sparse Matrix Computations”. In: *IEEE Intern. Symp. on High-Performance Computer Archit. (HPCA)*. 2021, pp. 921–934.

[103] Alex Peleg and Uri Weiser. “MMX technology extension to the Intel architecture”. In: *IEEE micro* 16.4 (2002), pp. 42–50.

[104] Matteo Perotti, Matheus Cavalcante, Renzo Andri, Lukas Cavigelli, and Luca Benini. “Ara2: Exploring Single-and Multi-Core Vector Processing with an Efficient RVV 1.0 Compliant Open-Source Processor”. In: *IEEE Trans. on Computers* (2024).

[105] Matteo Perotti, Matheus Cavalcante, Nils Wistoff, Renzo Andri, Lukas Cavigelli, and Luca Benini. “A “new ara” for vector computing: An open source highly efficient RISC-V v 1.0 vector processor design”. In: *IEEE Intern. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*. 2022, pp. 43–51.

[106] Michael Platzer and Peter Puschner. “Vicuna: A timing-predictable RISC-V vector coprocessor for scalable parallel computation”. In: *Euromicro conference on real-time systems (ECRTS)*. 2021.

[107] Aman Priyanshu, Yash Maurya, and Zuofei Hong. “AI governance and accountability: An analysis of anthropic’s claude”. In: *arXiv preprint arXiv:2407.01557* (2024).

[108] Navaneeth Kunhi Purayil, Matteo Perotti, Tim Fischer, and Luca Benini. *AraXL: A Physically Scalable, Ultra-Wide RISC-V Vector Processor Design for Fast and Efficient Computation on Long Vectors*. 2025. arXiv: 2501.10301 [cs.AR]. URL: https://arxiv.org/abs/2501.10301.

[109] Navaneeth Kunhi Purayil, Diyou Shen, Matteo Perotti, and Luca Benini. “TROOP: At-the-Roofline Performance for Vector Processors on Low Operational Intensity Workloads”. In: *arXiv preprint arXiv:2508.03900* (2025).

[110] Markus N Rabe and Charles Staats. "Self-attention does not need $O(n^2)$ memory". In: *arXiv preprint arXiv:2112.05682* (2021).

[111] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67.

[112] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. "A RISC-V simulator and benchmark suite for designing and evaluating vector architectures". In: *ACM Trans. on Architecture and Code Optimization (TACO)* 17.4 (2020), pp. 1–30.

[113] B Ramakrishna Rau and Joseph A Fisher. "Instruction-level parallel processing: History, overview, and perspective". In: *The journal of Supercomputing* 7.1 (1993), pp. 9–50.

[114] Susana Rebolledo, Borja Perez, Jose Luis Bosque, and Peter Hsu. "The Bicameral Cache: a split cache for vector architectures". In: *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2024, pp. 728–736.

[115] Gang Ren, Peng Wu, and David Padua. "Optimizing data permutations for SIMD devices". In: *ACM SIGPLAN Notices* 41.6 (2006), pp. 118–131.

[116] *RISC-V vector extension (Version 1.0)*. `https://github.com/riscv/riscv-v-spec`. Accessed: 03-13-2026.

[117] Jeff Rupley, Brad Burgess, Brian Grayson, and Gerald D Zuraski. "Samsung m3 processor". In: *IEEE Micro* 39.2 (2019), pp. 37–44.

[118] *RVV benchmark suite*. URL: `https://github.com/camel-cdr/rvv-bench`.

[119] Mohammad Erfan Sadeghi, Arash Fayyazi, Seyedarmin Azizi, and Massoud Pedram. "Peano-vit: Power-efficient approximations of non-linearities in vision transformers". In: *ACM/IEEE Intern. Symposium on Low Power Electronics and Design (ISLPED)*. 2024, pp. 1–6.

[120] Satyabrata Sarangi and Bevan Baas. "DeepScaleTool: A tool for the accurate estimation of technology scaling in the deep-submicron era". In: *Intern. Symp. on Circuits and Systems (ISCAS)*. 2021.

[121] Kiran Seshadri, Berkin Akin, James Laudon, Ravi Narayanaswami, and Amir Yazdanbakhsh. "An evaluation of edge tpu accelerators for convolutional neural networks". In: *International Symposium on Workload Characterization (IISWC)*. IEEE. 2022, pp. 79–91.

[122] Francesco Sgherzi, Marco Siracusa, Ivan Fernandez, Adrià Armejach, and Miquel Moret'o. "SpChar: Characterizing the Sparse Puzzle via Decision Trees". In: *arXiv preprint arXiv:2304.06944* (2023).

[123] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. "Flashattention-3: Fast and accurate attention with asynchrony and low-precision". In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 68658–68685.

[124] Kota Shiba, Mitsuji Okada, Atsutake Kosuge, Mototsugu Hamada, and Tadahiro Kuroda. "Polyomino: A 3D-SRAM-Centric Accelerator for Randomly Pruned Matrix Multiplication With Simple Reordering Algorithm and Efficient Compression Format in 180-nm CMOS". In: *IEEE Trans. on Circuits and Systems I* (2023).

[125] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. "Openai gpt-5 system card". In: *arXiv preprint arXiv:2601.03267* (2025).

[126] Mattia Sinigaglia, Amirhossein Kiamarzi, Marco Bertuletti, Luigi Ghionda, Mattia Orlandi, Riccardo Tedeschi, Aurora Di Giampietro, Yvan Tortorella, Luca Bertaccini, Simone Benatti, et al. "Maestro: A 302 GFLOPS/W and 19.8 GFLOPS RISC-V Vector-Tensor Architecture for Wearable Ultrasound Edge Computing". In: *arXiv preprint arXiv:2503.04581* (2025).

[127] Marco Siracusa et al. "A Tensor Marshaling Unit for Sparse Tensor Algebra on General-Purpose Processors". In: *IEEE/ACM Intern. Symp. on Microarchitecture*. 2023, pp. 1332–1346.

[128] Alan Smith and Vamsi Krishna Alla. "AMD instinct™ MI300X: a generative AI accelerator and platform architecture". In: *IEEE Micro* (2025).

[129] Steven Smith. *Digital signal processing: a practical guide for engineers and scientists*. Newnes, 2003.

[130] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. "Knights landing: Second-generation intel xeon phi product". In: *Ieee micro* 36.2 (2016), pp. 34–46.

[131] Zhuoran Song, Chunyu Qi, Yuanzheng Yao, Peng Zhou, Yanyi Zi, Nan Wang, and Xiaoyao Liang. "TSAcc: An Efficient Tempo-Spatial Similarity Aware Accelerator for Attention Acceleration". In: *ACM/IEEE Design Automation Conference*. 2024.

[132] Shrihari Sridharan, Jacob R Stevens, Kaushik Roy, and Anand Raghunathan. "X-former: In-memory acceleration of transformers". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 31.8 (2023), pp. 1223–1233.

[133] Nitish Srivastava. *A Tutorial on the Gem5 Memory Model*. URL: `https://nitish2112.github.io/post/gem5-memory-model`.

[134] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product". In: *IEEE Int. Symp. on Microarchitecture*. 2020, pp. 766–780.

[135] Milan Stanic et al. "Towards low-power embedded vector processor". In: *ACM Intern. Conf. on Computing Frontiers*. 2016, pp. 339–342.

[136] Nigel Stephens et al. "The ARM scalable vector extension". In: *IEEE/ACM Intern. Symp. on Microarchitecture (Micro)* 37.2 (2017), pp. 26–39.

[137] Jacob R Stevens et al. "Softermax: Hardware/software co-design of an efficient softmax for transformers". In: *ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 469–474.

[138] STMicroelectronics. *STM32 Toolchain Specifications*. 2023. URL: `https://www.st.com/en/microcontrollers-microprocessors/stm32h743ii.html`.

[139] Tushar Prasanna Swaminathan, Christopher Silver, Thangarajah Akilan, and Jitendra Kumar. "Benchmarking deep learning models on nvidia jetson nano for real-time systems: An empirical investigation". In: *Procedia Computer Science* 260 (2025), pp. 906–913.

[140] Espressif Systems. *ESP32 - Technical Reference Manual*. Espressif whitepaper. 2024.

[141] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. "Rethinking the inception architecture for computer vision". In: *IEEE Conf. on Comp. Vision and Pattern Recogn. (CVPR)*. 2016, pp. 2818–2826.

[142] Tuan Ta, Khalid Al-Hawaj, Nick Cebry, Yanghui Ou, Eric Hall, Courtney Golden, and Christopher Batten. "Big.VLITTLE: On-demand data-parallel acceleration for mobile systems on chip". In: *IEEE/ACM Intern. Symp. on Microarchitecture (MICRO)*. 2022, pp. 181–198.

[143] Hikaru Takayashiki, Hotaka Yagi, Hiroki Nishimoto, and Naoki Yoshifuji. "A New Sparse GEneral Matrix-matrix Multiplication Method for Long Vector Architecture by Hierarchical Row Merging". In: *SC'23 Workshops - Intern. Conf. on High Perf. Comp., Network, Storage, and Analysis*. 2023, pp. 756–759.

[144] Hongbing Tan, Sheng Liu, Haiyan Chen, and Jianguo Wu. "Modeling and evaluation for gather/scatter operations in vector-simd architectures". In: *Application-specific systems, architectures and processors (asap)*. IEEE. 2017, pp. 143–148.

[145] Vasileios Titopoulos, George Alexakis, Kosmas Alexandridis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. "Register Dispersion: Reducing the Footprint of the Vector Register File in Vector Engines of Low-Cost RISC-V CPUs". In: *ACM International Conference on Computing Frontiers*. 2025, pp. 3–11.

[146] Vasileios Titopoulos, George Alexakis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. "Efficient Implementation of RISC-V Vector Permutation Instructions". In: *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. Vol. 1. 2025, pp. 1–6.

[147] Vasileios Titopoulos, George Alexakis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. "Vector Processing on Low-Cost RISC-V Cores via Reduced Number of Physical Vector Registers". In: *ACM Transactions on Parallel Computing* (2026).

[148] Vasileios Titopoulos, Kosmas Alexandridis, and Giorgos Dimitrakopoulos. "Vectorized FlashAttention with low-cost exponential computation in RISC-V vector processors". In: *The Journal of Supercomputing* 82.4 (2026), p. 189.

[149] Vasileios Titopoulos, Kosmas Alexandridis, Christodoulos Peltekis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. "IndexMAC: A custom RISC-V vector instruction to accelerate structured-sparse matrix multiplications". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2024, pp. 1–6.

[150] Vasileios Titopoulos, Kosmas Alexandridis, Christodoulos Peltekis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. “Optimizing Structured-Sparse Matrix Multiplication in RISC-V Vector Processors”. In: *IEEE Transactions on Computers* 74.4 (2025), pp. 1446–1460.

[151] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023).

[152] Daniel Towner. *Intel AVX-512 - Permuting Data Within and Between AVX Registers*. Tech. rep. Intel - Technology Guide, 2022.

[153] Luca Valente. “A Vector Processing Unit implementation for RISC-V Vector Extension: Functional Verification and Assertions on submodules.” PhD thesis. Politecnico di Torino, 2020.

[154] Aravind Vasudevan, Andrew Anderson, and David Gregg. “Parallel multi channel convolution using general matrix multiplication”. In: *Application-specific systems, architectures and processors (ASAP)*. IEEE. 2017, pp. 19–24.

[155] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. “Attention is all you need”. In: *Intern. Conf. on Neural Information Processing Systems (NIPS)*. 2017, pp. 6000–6010.

[156] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. “GLUE: A multi-task benchmark and analysis platform for natural language understanding”. In: *arXiv preprint arXiv:1804.07461* (2018).

[157] Run Wang, Gamze Islamoglu, Andrea Belano, Viviane Potocnik, Francesco Conti, Angelo Garofalo, and Luca Benini. “VEXP: A Low-Cost RISC-V ISA Extension for Accelerated Softmax Computation in Transformers”. In: *arXiv preprint arXiv:2504.11227* (2025).

[158] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. “Linformer: Self-attention with linear complexity”. In: *arXiv preprint arXiv:2006.04768* (2020).

[159] Wenxun Wang, Shuchang Zhou, Wenyu Sun, Peiqin Sun, and Yongpan Liu. “SOLE: Hardware-Software Co-design of Softmax and LayerNorm for Efficient Transformer Inference”. In: *IEEE/ACM Intern. Conference on Computer Aided Design (ICCAD)*. 2023, pp. 1–9.

[160] Yipeng Wang, Mengtian Yang, Chieh-Pu Lo, and Jaydeep P Kulkarni. “Vecim: A 289.13 GOPS/W RISC-V Vector Co-Processor with Compute-in-Memory Vector Register File for Efficient High-Performance Computing”. In: *IEEE Intern. Solid-State Circuits Conf.(ISSCC)*. 2024, pp. 492–494.

[161] Zhican Wang, Gang Wang, and Guanghui He. “COSA Plus: Enhanced Co-Operative Systolic Arrays for Attention Mechanism in Transformers”. In: *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 44.2 (2025), pp. 723–736.

[162] Thomas Wolf et al. “Transformers: State-of-the-Art Natural Language Processing”. In: *Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 2020, pp. 38–45.

[163] J. Yiu. *Introduction to the ARM Cortex-M55 Processor*. ARM whitepaper. 2020.

[164] Jerry Zhao, Daniel Grubb, Miles Rusch, Tianrui Wei, Kevin Anderson, Borivoje Nikolic, and Krste Asanovic. "Instruction Scheduling in the Saturn Vector Unit". In: *arXiv preprint arXiv:2412.00997* (2024).

[165] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine". In: (2020).

[166] Tenghao Zhao and Zhaohui Ye. "ZeroVex: A Scalable and High-performance RISC-V Vector Processor Core for Embedded Systems". In: *IEEE Intern. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*. 2024, pp. 32–33.

[167] Aojun Zhou et al. "Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch". In: *Inter. Conf. on Learning Representations*. 2021.

[168] Zhe Zhou, Junlin Liu, Zhenyu Gu, and Guangyu Sun. "Energon: Toward efficient acceleration of transformers using dynamic sparse attention". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42.1 (2022), pp. 136–149.

[169] Kaijie Zhu et al. "PromptBench: A Unified Library for Evaluation of Large Language Models". In: *arXiv preprint arXiv:2312.07910* (2023).

[170] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs". In: *IEEE Int. Symp. on Microarchitecture*. 2019, pp. 359–371.