

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

RTL Design and Analysis of Sparse Tensor Cores for Unstructured Sparsity in RISC-V GPUs

---

Συγγραφέας:
Γεώργιος Τριανταφύλλου

Επιβλέπων:
Σωτήριος Ξύδης
Επίκουρος Καθηγητής

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

15 Ιουνίου 2026



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

RTL Design and Analysis of Sparse Tensor Cores for Unstructured Sparsity in RISC-V GPUs

---

Συγγραφέας:
Γεώργιος Τριανταφύλλου

Επιβλέπων:
Σωτήριος Ξύδης
Επίκουρος Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 15η Ιουνίου 2026.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
Σωτήριος Ξύδης	Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή
Επίκουρος Καθηγητής	Καθηγητής	Ομότιμος Καθηγητής

15 Ιουνίου 2026

Πνευματικά Δικαιώματα

Copyright © Γεώργιος Τριανταφύλλου, 2026.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ΄ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μή κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσοβίου Πολυτεχνείου.

Υπογραφή:

_____

Ημερομηνία:

_____

Εθνικό Μετσόβιο Πολυτεχνείο

Περίληψη

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

RTL Design and Analysis of Sparse Tensor Cores for Unstructured Sparsity in RISC-V GPUs

του Γεώργιου Τριανταφύλλου

Η ραγδαία ανάπτυξη των DNNs και LLMs έχει αυξήσει την ανάγκη για αποδοτικό πολλαπλασιασμό πινάκων, ο οποίος κυριαρχεί στα σύγχρονα φορτία τεχνητής νοημοσύνης και είναι κεντρικός σε πολλές εφαρμογές HPC. Οι σύγχρονες GPUs επιταχύνουν το GEMM μέσω εξειδικευμένων μονάδων, όπως τα Tensor Cores, ωστόσο οι σχεδιασμοί αυτοί συχνά περιορίζονται από τη μετακίνηση δεδομένων, την πίεση στο αρχείο καταχωρητών και τα σημεία συμφόρησης της μνήμης.

Η αραιότητα (sparsity) μπορεί να μειώσει τόσο τους υπολογισμούς όσο και την κυκλοφορία δεδομένων στη μνήμη. Παρ' όλα αυτά, η υπάρχουσα υποστήριξη σε επίπεδο υλικού περιορίζεται συνήθως σε δομημένα πρότυπα, όπως η αραιότητα 2:4 (Ampere, NVIDIA, 2020), και αξιοποιεί την αραιότητα μόνο σε έναν από τους δύο τελεστέους, περιορίζοντας τα οφέλη απόδοσης και ενδεχομένως επηρεάζοντας την ακρίβεια του μοντέλου.

Η παρούσα διπλωματική εργασία παρουσιάζει τον RTL σχεδιασμό και την αξιολόγηση ενός Tensor Core με υποστήριξη αδόμητης αραιότητας και στους δύο τελεστέους, ενσωματωμένου σε μια ανοιχτού κώδικα RISC-V GPGPU. Η προτεινόμενη αρχιτεκτονική υποστηρίζει πυκνή και αραιή εκτέλεση GEMM, επιτρέπει άμεση πρόσβαση στη μοιραζόμενη μνήμη για τη μείωση της χρήσης του αρχείου καταχωρητών και του κόστους εντολών, και χρησιμοποιεί εξωτερικό γινόμενο με bitmap κωδικοποίηση ώστε να παραλείπει μη αποδοτικούς υπολογισμούς και στους δύο τελεστέους.

Ο σχεδιασμός υποστηρίζεται από επεκτάσεις στο ISA, βελτιστοποιημένο πυρήνα και μικροαρχιτεκτονικές βελτιστοποιήσεις, και αξιολογείται έναντι πυκνών και δομημένα αραιών Vortex Tensor Core baselines μέσω κύκλο-ακριβούς RTL προσομοίωσης και μετρικών σύνθεσης. Τα πειραματικά αποτελέσματα καταδεικνύουν μέση επιτάχυνση $12.73\times$ και μέγιστη επιτάχυνση $27\times$, ενώ ποσοτικοποιούν τους συμβιβασμούς σε επιφάνεια, ισχύ και απόδοση.

Λέξεις Κλειδιά: GPU, Vortex, Πολλαπλασιασμός πινάκων, Αδόμητη Αραιότητα, Tensor Core, 2:4 Αραιότητα, Μοιραζόμενη Μνήμη, DMA, HW / SW co-design

National Technical University of Athens

Abstract

Division of Computer Science
School of Electrical and Computer Engineering

Diploma Thesis

RTL Design and Analysis of Sparse Tensor Cores for Unstructured Sparsity in RISC-V GPUs

by Georgios Triantafyllou

The rapid growth of DNNs and LLMs has increased the demand for efficient matrix multiplication, which dominates modern AI workloads and is also central to many HPC applications. Modern GPUs accelerate GEMM through specialized units such as Tensor Cores [1, 2, 3], yet these designs often remain limited by data movement, register-file pressure, and memory-system bottlenecks [4, 5].

Sparsity offers an opportunity to reduce both computation and memory traffic. However, existing hardware support is typically restricted to structured patterns, such as 2:4 sparsity [6, 7], and usually exploits sparsity in only one operand, limiting performance gains and potentially affecting model accuracy.

This thesis presents the RTL design and evaluation of a dual-side unstructured sparse Tensor Core integrated into an open-source RISC-V GPGPU. The proposed architecture supports dense and sparse GEMM, enables direct shared-memory access to reduce register-file staging and instruction overhead, and uses an outer-product model with bitmap-based sparse encoding to skip ineffective computations in both operands [8, 9].

The design is supported by ISA extensions, an optimized kernel, and microarchitectural tuning, and is evaluated against dense and structured-sparse Vortex Tensor Core baselines using cycle-accurate RTL simulation and synthesis metrics. Experimental results demonstrate a mean speedup of $12.73\times$ and a peak speedup of $27\times$, while providing insight into the area, power, and performance trade-offs of dual-side unstructured sparsity support.

Keywords: GPU, Vortex, Matrix Multiplication, Unstructured Sparsity, Tensor Core, 2:4 Structured Sparsity, Shared Memory, DMA, HW / SW co-design

Acknowledgements

This thesis concludes my studies at the School of Electrical and Computer Engineering of the National Technical University of Athens.

I would like to express my profound and sincere thanks to Professor Sotirios Xydis for giving me the opportunity to carry out my diploma thesis under his supervision. His guidance and advice, alongside his invaluable research experience provided me with the motivation and inspiration to accomplish this work.

In addition, I would like to thank Panagiotis-Eleftherios Eleftherakis and Konstantinos Iliakis, who advised and stood by me throughout the duration of this thesis with accurate observations and suggestions. They were always willing to support me in every difficulty I faced and share their knowledge with me, so I feel grateful for having the opportunity to cooperate with them.

I give my special thanks to Demetra and all my friends for supporting me whenever I needed it the most. Finally, I would like to thank my family who always stood by and encouraged me to achieve my goals.

Contents

Declaration of Authorship v

Abstract ix

Acknowledgements xi

1 Εκτεταμένη Περίληψη 1
- 1.1 Εισαγωγή . . . 1
- 1.2 Θεωρητικό Υπόβαθρο . . . 4
- 1.3 Λεπτομέρειες της Υλοποίησης . . . 8
- 1.4 Αξιολόγηση . . . 15
- 1.5 Συμπεράσματα και Μελλοντικές Επεκτάσεις . . . 20

2 Introduction 23
- 2.1 The Rise of Artificial Intelligence in Modern Computing . . . 23
- 2.2 GPU Acceleration, Tensor Cores, and Sparsity . . . 25
- 2.3 Vortex GPU and RTL-Based Evaluation . . . 26
- 2.4 Proposal Overview . . . 27
- 2.5 Contributions . . . 28
- 2.6 Thesis Structure . . . 28

3 Background 31
- 3.1 Introduction . . . 31
- 3.2 GPUs as AI Acceleration Platforms . . . 31
- 3.3 Fundamentals of GPUs . . . 32
- 3.4 Tensor Cores in GPUs . . . 34
- 3.5 Outer-Product Matrix Multiplication . . . 36
- 3.6 Sparsity in GEMM Workloads . . . 38
- 3.7 Vortex: An RTL-Level RISC-V GPGPU . . . 40
- 3.8 Vortex Tensor Core Unit (TCU) . . . 42
  - 3.8.1 Operation of the Vortex TCU . . . 42
  - 3.8.2 Limitations of the Vortex TCU . . . 45
- 3.9 Overview of DSTC: Dual-Side Sparse Tensor Core . . . 46

4 Implementation Details **55**
4.1 Introduction . . . 55
4.2 FEOP Unit Design . . . 55
4.3 TCU Accumulation Buffer Design . . . 58
4.4 TCU Core Design . . . 62
TCU Core Control and Operand Dispatch . . . 62
Additional Uncompressed × Compressed sparse execution mode . . . 65
Memory-Bound Behavior and Direct Memory Access . . . 65
Barrier Handling . . . 67
4.5 Kernel Design . . . 68
4.6 Datatype Support and ISA Extension . . . 71

5 Microarchitecture Optimizations and Right Sizing **73**
5.1 Introduction . . . 73
5.2 Step Size Exploration . . . 73
5.3 Step Dimension Exploration . . . 74
5.4 Queue Depth Exploration . . . 78

6 Experimental Evaluation **79**
6.1 Introduction . . . 79
6.2 Benchmarks and Baselines . . . 79
6.3 Useful Metrics . . . 80
6.4 Performance Comparison . . . 81
6.5 Total Instruction Count and Front-End Pressure . . . 83
6.6 Memory Accesses . . . 85
6.7 Tensor Core Multiplier Utilization . . . 88
6.8 DMA Transactions . . . 89
6.9 K-Dimension Size Sensitivity . . . 91
6.10 Performance Gains Across Various Sparsities . . . 92
6.10.1 Compressed–Compressed Mode . . . 92
6.10.2 Uncompressed–Compressed Mode . . . 93

7 Related Work **95**
7.1 Introduction . . . 95
7.2 VEGETA . . . 95
7.3 Vortex Structured **2**:**4** Sparse Tensor Core . . . 96
7.4 Cooperative Warp Execution in Tensor Core for RISC-V GPGPU . . . 96
7.5 Virgo . . . 97

8 Conclusions and Future Work **99**
8.1 Conclusions 99
8.2 Future Work 100

A Source Code **103**
A.1 Source code repository 103

Bibliography **105**

List of Figures

1.1 Πολλαπλασιασμός πινάκων με εξωτερικό γινόμενο. Κάθε στήλη του A συνδυάζεται με την αντίστοιχη γραμμή του B, παράγοντας ένα μερικό πλακίδιο που συσσωρεύεται στο αποτέλεσμα. 5
1.2 Συμπίεση και bitmap αναπαράσταση αραιών τελεστέων. Τα μη μηδενικά στοιχεία αποθηκεύονται συνεχόμενα, ενώ τα bitmaps διατηρούν τις αρχικές τους θέσεις. 6
1.3 Υψηλού επιπέδου μικροαρχιτεκτονική του Vortex. Η ανοικτή RTL υλοποίηση επιτρέπει την ενσωμάτωση και αξιολόγηση νέων μηχανισμών Tensor Core. 7
1.4 Παράδειγμα εκτέλεσης ενός βήματος FEOP. Το πλήρες πλακίδιο εξόδου καλύπτεται σταδιακά από μικρότερα υποπλακίδια εξωτερικού γινομένου. 9
1.5 Οργάνωση του συσσωρευτή της TCU-OP. Οι ουρές απορροφούν προσωρινές συγκρούσεις, το crossbar δρομολογεί τα μερικά αποτελέσματα και οι τοπικοί αθροιστές ενημερώνουν τις θύρες του συσσωρευτή. 10
1.6 Υψηλού επιπέδου pipeline υπολογισμού της TCU-OP. Τα bitmaps αποκωδικοποιούνται, τα FEOP παράγουν μερικά γινόμενα, και ο συσσωρευτής ενημερώνει το πλακίδιο εξόδου. 11
1.7 Συνολική οργάνωση της προτεινόμενης TCU-OP και σύνδεση με το σύστημα μνήμης. 12
1.8 Pipeline του βελτιστοποιημένου πυρήνα. Η παραγωγή μεταδεδομένων, οι DMA μεταφορές και η ασύγχρονη εκτέλεση της TCU-OP επικαλύπτονται. 13
1.9 Επίδραση των διαστάσεων βήματος στον χρόνο εκτέλεσης και στα crossbar stalls. Η διαμόρφωση 2×16 επιλέγεται ως πιο αποδοτική. 14
1.10 Συνολικοί κύκλοι πυρήνα για SGEMM, SGEMM_SP και SGEMM_OP σε διαφορετικούς τύπους δεδομένων. 16
1.11 Συνολικός αριθμός εντολών. Η SGEMM_OP μειώνει δραστικά τις εντολές επειδή αποφεύγει το explicit register-file staging. 17

1.12 Συνολικές off-chip προσπελάσεις μνήμης. Η SGEMM_OP μειώνει την κίνηση μέσω μεγαλύτερης επαναχρησιμοποίησης και συνεχόμενων DMA μεταφορών. 17
1.13 Χρησιμοποίηση πολλαπλασιαστών Tensor Core. Η SGEMM_OP διατηρεί το datapath ενεργό πολύ συχνότερα από τα baselines. . . 18
1.14 DMA συναλλαγές για τους διαφορετικούς τρόπους αραιότητας. Η πλήρης συμπίεση μειώνει τις συναλλαγές όσο αυξάνεται η αραιότητα. 19
1.15 Σύγκριση συμπιεσμένου επί συμπιεσμένο και ασυμπίεστου επί συμπιεσμένο τρόπου σε διαφορετικά επίπεδα αραιότητας. 20

3.1 Outer-product matrix multiplication. Each step multiplies one column of A with one row of B, producing a rank-1 partial output tile that is accumulated into the output matrix. 36
3.2 High-level Vortex micro-architecture 40
3.3 Register-level fragmentation of Vortex Tensor Core warp tiles for a 4-thread configuration. Matrix tiles are distributed across thread registers, and registers with the same index form the blocks consumed by individual TCU micro-operations. 42
3.4 Bitmap-based outer-product sparse GEMM. Nonzero elements from a column of A and a row of B are multiplied to produce sparse partial output matrices, which are accumulated into the output tile. . . 46
3.5 Sparse outer-product execution in DSTC. (a) Outer-product multiplication when both input operands are sparse. (b) Compaction of the nonzero elements of the A column and B row enables zero-only computation steps to be skipped. (c) Bitmap metadata records the original positions of the nonzero elements in each input vector. . . 49
3.6 Simplified Tile Index Aligner (TIA) example. The TIA extracts the addresses of nonzero elements from bitmap metadata. At each stage, a shifter handles twice as many elements as the previous stage; therefore, for 32-bit inputs, one additional layer is required beyond the ones shown. 50
3.7 Partial sums produced from a single 4×4 step. (a) In dense execution, outputs are naturally distributed across different banks. (b) In sparse execution, irregular output addresses may cause multiple partial sums to target the same bank, creating bank conflicts. 52
3.8 Accumulation buffer organization. Queues absorb temporary imbalances in sparse output addresses, queue-local adders update the corresponding accumulation-buffer entries and the crossbar routes the results to the correct banks 52

4.1 High-level integration of the proposed TCU-OP inside the Vortex GPU. The highlighted components show the modified core paths and the supporting memory and synchronization mechanisms used by the proposed design. 56
4.2 Example of FEOP-based outer-product step execution. The accumulation buffer has dimensions 16×16, while each step computes a 4×8 region. Therefore, each set requires eight steps in total. 57
4.3 Organization of the TCU accumulation buffer. FEOP outputs and their corresponding addresses are first stored in queues. The crossbar routes each valid partial product to the appropriate accumulator bank, where a bank-local adder updates the stored value. 60
4.4 Example of step skipping for Set 1. The selected row of B is divided into four horizontal regions, corresponding to 25% sparsity granularity, while the selected column of A is divided into eight vertical regions, corresponding to 12.5% sparsity granularity. Only the highlighted steps contain useful nonzero work and must be executed. 62
4.5 High-level computation pipeline of TCU-OP 62
4.6 Compressed representation of an A block and its interpretation inside the TCU-OP. (a) Nonzero values are stored contiguously in memory, accompanied by bitmap metadata that records their original positions. (b) The TCU-OP uses the bitmap to recover the logical column positions of the compressed values before dispatching them to the FEOP units. 63
4.7 Operation of the Tile Index Aligner (TIA). In the simplified 8-bit example shown here, two sorter stages are sufficient; for the actual 32-bit inputs used in the design, two additional shifter stages are required. 64
4.8 Overview of the proposed TCU-OP architecture and its connections to the GPU memory system. The core control logic manages step and set execution, the request and response units orchestrate shared-memory accesses for A, B, bitmap, and C blocks, and the FEOP feeder dispatches operands to the FEOP units. Partial products are accumulated in the accumulation buffer, while final results are flushed back to memory during commit. 68
4.9 Kernel execution pipeline using shared-memory double buffering. Kernel-side metadata generation, DMA data transfers, and asynchronous TCU execution are overlapped. 69

4.10 Effect of accumulation-buffer size on input reuse. (a) With a small accumulator, the output matrix must be computed using smaller tiles, causing input tiles to be fetched twice. (b) With a larger accumulator, the tile shape can be expanded so that each fetched input tile is reused across a larger output region, being fetched only once and doubling reuse. 70

5.1 Effect of FEOP step dimensions on kernel cycles and crossbar stalls for a $64 \times 64 \times 512$ FP8 GEMM with queue depth $Q = 4$. The 2×16 configuration consistently achieves lower execution time than 4×8, mainly due to fewer crossbar stalls and lower flushing overhead. . . 77
5.2 Effect of crossbar queue depth on stall percentage for a $64 \times 64 \times 512$ FP8 GEMM. 78

6.1 Kernel execution cycles for SGEMM, SGEMM_SP, and SGEMM_OP across FP8, FP16, and FP32 input datatypes. The evaluated GEMM size is $M \times N \times K = 128 \times 128 \times 512$. 81
6.2 Kernel execution cycles of SGEMM_OP across input datatypes and unstructured A/B sparsity levels. The isolated view highlights the performance gains obtained from sparse step skipping in the proposed TCU-OP design. 82
6.3 Total kernel instruction count for SGEMM, SGEMM_SP, and SGEMM_OP across input datatypes and sparsity modes. SGEMM_OP substantially reduces instruction count by replacing explicit register-file staging with asynchronous DMA and TCU execution. 83
6.4 Instruction count of SGEMM_OP across input datatypes and sparsity levels. Sparse execution introduces a small fixed overhead due to bitmap address generation and additional DMA launches. 84
6.5 Total off-chip memory requests for SGEMM, SGEMM_SP, and SGEMM_OP across input datatypes and sparsity modes. Each bar is split into read and write requests. 85
6.6 Off-chip memory requests of SGEMM_OP across input datatypes and unstructured sparsity levels. Higher sparsity reduces the number of transferred nonzero operand values, despite the additional bitmap traffic. 85
6.7 L1 data-cache requests for SGEMM, SGEMM_SP, and SGEMM_OP across input datatypes and sparsity modes. Each bar is split into hits, read misses, and write misses. 87

6.8 L1 data-cache requests of SGEMM_OP across input datatypes and sparsity levels. The requests remain constant across sparsity levels, since operand movement is performed through DMA rather than regular data-cache accesses. 87
6.9 Tensor Core multiplier utilization across input datatypes and evaluated kernels. The baseline Vortex implementations exhibit very low utilization, while SGEMM_OP substantially improves multiplier activity by reducing operand-staging overhead and feeding the Tensor Core more efficiently. 88
6.10 DMA transactions from global memory to shared memory for different sparsity modes. The dense case loads uncompressed operands, the compressed–compressed mode loads compressed A and B values together with bitmap metadata, and the uncompressed–compressed mode loads uncompressed A values together with compressed B values and bitmap metadata. 90
6.11 TCU multiplier utilization as a function of the K dimension for the 2×16 and 4×8 FEOP step configurations. The number of output tiles is kept constant, so increasing K amortizes the fixed per-output-tile overheads. 91
6.12 End-to-end kernel cycles for the compressed–compressed sparsity mode across different A and B sparsity levels. Both operands are stored in compressed form and accompanied by bitmap metadata. . 93
6.13 End-to-end kernel cycles for the uncompressed–compressed sparsity mode compared with the compressed–compressed mode. Solid lines correspond to compressed A and compressed B, while dashed lines correspond to uncompressed A and compressed B. 94

Κεφάλαιο 1

Εκτεταμένη Περίληψη

1.1 Εισαγωγή

Η σύγχρονη υπολογιστική πραγματικότητα χαρακτηρίζεται από τη ραγδαία ανάπτυξη της Τεχνητής Νοημοσύνης, της Μηχανικής Μάθησης και των εφαρμογών Υπολογιστικής Υψηλών Επιδόσεων. Τα τελευταία χρόνια, μοντέλα βαθιάς μάθησης χρησιμοποιούνται σε ένα πολύ ευρύ φάσμα εφαρμογών, όπως η επεξεργασία φυσικής γλώσσας, η όραση υπολογιστών, η ανάλυση μεγάλων δεδομένων, η βιοϊατρική, τα αυτόνομα συστήματα και η επιστημονική προσομοίωση. Η εξέλιξη αυτή δεν οφείλεται μόνο σε καλύτερους αλγορίθμους ή σε μεγαλύτερα σύνολα δεδομένων. Εξίσου κρίσιμος παράγοντας είναι η δυνατότητα του υλικού να εκτελεί τους αντίστοιχους υπολογισμούς σε πρακτικό χρόνο και με αποδεκτή κατανάλωση ενέργειας.

Ένα από τα σημαντικότερα ορόσημα αυτής της εξέλιξης ήταν η εισαγωγή του μηχανισμού προσοχής και της αρχιτεκτονικής Transformer [10], καθώς και η βελτίωση υπάρχοντων μοντέλων. Μετά από αυτήν την εξέλιξη, τα μεγάλα γλωσσικά μοντέλα και γενικότερα τα foundation models αυξήθηκαν δραματικά σε μέγεθος, αριθμό παραμέτρων και υπολογιστικές απαιτήσεις. Η εκπαίδευση και η εκτέλεση τέτοιων μοντέλων απαιτεί τεράστιο αριθμό πράξεων κινητής υποδιαστολής, υψηλό εύρος ζώνης μνήμης και αποδοτική μεταφορά δεδομένων μεταξύ των επιπέδων της ιεραρχίας μνήμης. Καθώς τα μοντέλα μεγαλώνουν, η απλή αύξηση της κορυφαίας υπολογιστικής ισχύος δεν είναι αρκετή. Η πραγματική απόδοση εξαρτάται από το πόσο αποτελεσματικά χρησιμοποιούνται οι μονάδες υπολογισμού, πόσες εντολές απαιτούνται για τη μεταφορά των τελεστέων, και πόση κίνηση δημιουργείται στο αρχείο καταχωρητών και στη μνήμη [11].

Στον πυρήνα πολλών τέτοιων εφαρμογών βρίσκεται ο πολλαπλασιασμός πινάκων. Στα πολυεπίπεδα νευρωνικά δίκτυα, κάθε πλήρως συνδεδεμένο επίπεδο μπορεί να εκφραστεί ως πολλαπλασιασμός πίνακα ενεργοποιήσεων με πίνακα βαρών. Στα συνελικτικά νευρωνικά δίκτυα, οι συνελίξεις συχνά μετασχηματίζονται σε πράξεις GEMM μέσω τεχνικών όπως το im2col. Στα μοντέλα Transformer,

οι προβολές σε query, key και value, ο υπολογισμός των βαθμολογιών προσοχής και τα ενδιάμεσα MLP επίπεδα βασίζονται σε μεγάλους πολλαπλασιασμούς πινάκων. Παρόμοιες πράξεις εμφανίζονται και σε εφαρμογές HPC, όπως προσομοιώσεις ρευστών, πεπερασμένα στοιχεία, μοριακή δυναμική και αριθμητική γραμμική άλγεβρα.

Για αυτόν τον λόγο, η επιτάχυνση του GEMM αποτελεί κεντρικό στόχο στις σύγχρονες αρχιτεκτονικές επεξεργαστών. Οι CPUs προσφέρουν ευελιξία και ισχυρή σειριακή απόδοση, αλλά δεν μπορούν να παρέχουν το επίπεδο μαζικής παραλληλίας που απαιτούν οι σημερινοί φόρτοι εργασίας τεχνητής νοημοσύνης. Οι ASICs, όπως η TPU της Google, μπορούν να επιτύχουν πολύ υψηλή ενεργειακή αποδοτικότητα μέσω εξειδικευμένων δομών, όπως οι συστοιχίες συστολικής εκτέλεσης [12]. Ωστόσο, η υπερβολική εξειδίκευση μειώνει την ευελιξία, κάτι που είναι σημαντικό επειδή τα μοντέλα και οι τελεστές της μηχανικής μάθησης αλλάζουν γρήγορα. Οι GPUs προσφέρουν έναν ενδιάμεσο και ιδιαίτερα επιτυχημένο συμβιβασμό: διατηρούν προγραμματιστική ευελιξία, ενώ παρέχουν μεγάλο αριθμό παράλληλων νημάτων, υψηλό εύρος ζώνης μνήμης και εξειδικευμένες μονάδες για πράξεις πινάκων.

Η ανάγκη για ακόμη υψηλότερη απόδοση από τη μία και ευελιξία από την άλλη οδήγησε στην εισαγωγή των Tensor Cores. Οι μονάδες αυτές εμφανίστηκαν εμπορικά στην αρχιτεκτονική NVIDIA Volta και σχεδιάστηκαν για να εκτελούν μικρές πράξεις πολλαπλασιασμού και συσσώρευσης πινάκων απευθείας στο υλικό [1, 2, 13]. Αντί ένας πολλαπλασιασμός πινάκων να αποσυντίθεται σε πολλές βαθμωτές ή διανυσματικές εντολές, μια εντολή MMA μπορεί να ενεργοποιήσει έναν εξειδικευμένο datapath που παράγει πολλά μερικά αποτελέσματα ανά κύκλο. Με την πάροδο των γενεών, οι Tensor Cores υποστήριξαν περισσότερους τύπους δεδομένων, όπως FP16, TF32, FP8 και ακέραιες μορφές, ενώ εισήχθη και υποστήριξη δομημένης αραιότητας σε νεότερες αρχιτεκτονικές [7, 14, 15].

Παρά τη μεγάλη επιτυχία των Tensor Cores, η απόδοσή τους στην πράξη περιορίζεται συχνά από την κίνηση δεδομένων. Οι τελεστέοι πρέπει να μεταφερθούν από την καθολική μνήμη στη μοιραζόμενη μνήμη, από εκεί στο αρχείο καταχωρητών, και τελικά στη μονάδα Tensor Core. Αυτή η διαδρομή δημιουργεί εντολές φόρτωσης, υπολογισμούς διευθύνσεων, πίεση στο αρχείο καταχωρητών και κυκλοφορία στην εσωτερική μικροαρχιτεκτονική του πυρήνα. Καθώς η υπολογιστική ικανότητα των Tensor Cores αυξάνεται, η μεταφορά και η επαναχρησιμοποίηση των τελεστέων γίνονται εξίσου σημαντικές με τον αριθμό των πολλαπλασιαστών [3, 4, 5].

Ένα δεύτερο σημαντικό χαρακτηριστικό των σύγχρονων φόρτων εργασίας είναι η αραιότητα. Σε πολλούς πίνακες βαρών ή ενεργοποιήσεων, ένα μεγάλο ποσοστό στοιχείων είναι μηδενικό. Η αραιότητα μπορεί να προκύψει από τεχνικές

κλαδέματος βαρών (pruning) [16], αλλά και δυναμικά από συναρτήσεις ενεργοποίησης όπως η ReLU [17]. Εφόσον κάθε πολλαπλασιασμός που περιλαμβάνει μηδενικό τελεστή δίνει μηδενικό αποτέλεσμα, ένα σύστημα που γνωρίζει τις θέσεις των μη μηδενικών στοιχείων μπορεί να παραλείψει άχρηστους υπολογισμούς και να μειώσει την κίνηση δεδομένων.

Η δυσκολία είναι ότι η αραιότητα μπορεί να είναι είτε δομημένη είτε αδόμητη. Η δομημένη αραιότητα, όπως το σχήμα 2:4 της NVIDIA Ampere, είναι φιλική προς το υλικό, επειδή περιορίζει το μοτίβο των μη μηδενικών στοιχείων και απλοποιεί την αποκωδικοποίηση [7, 6]. Όμως, αυτός ο περιορισμός μπορεί να μειώσει την ακρίβεια του μοντέλου ή να αφήσει ανεκμετάλλευτη επιπλέον αραιότητα. Αντίθετα, η αδόμητη αραιότητα επιτρέπει τα μηδενικά να βρίσκονται σε οποιεσδήποτε θέσεις και είναι γενικότερη, αλλά είναι δυσκολότερο να υποστηριχθεί αποδοτικά στο υλικό [18]. Επιπλέον, πολλές αρχιτεκτονικές εκμεταλλεύονται αραιότητα μόνο σε έναν τελεστή, συνήθως στον πίνακα βαρών. Σε πραγματικά νευρωνικά δίκτυα, όμως, τόσο τα βάρη όσο και οι ενεργοποιήσεις μπορούν να είναι αραιά.

Η παρούσα εργασία μελετά ακριβώς αυτό το κενό. Στόχος της είναι ο σχεδιασμός, η υλοποίηση και η αξιολόγηση μιας μονάδας Tensor Core που μπορεί να επιταχύνει πολλαπλασιασμό πινάκων με αδόμητη αραιότητα και στους δύο τελεστέους. Η προτεινόμενη αρχιτεκτονική, που ονομάζεται TCU-OP, υλοποιείται σε επίπεδο RTL μέσα στο Vortex, μια ανοικτού κώδικα RISC-V GPGPU αρχιτεκτονική [19]. Η επιλογή του Vortex είναι σημαντική, επειδή επιτρέπει αξιολόγηση ακρίβειας κύκλου και πραγματική ενσωμάτωση σε υλικό, σε αντίθεση με προσεγγίσεις που εξετάζονται μόνο σε υψηλού επιπέδου προσομοιωτές [20].

Η σχεδίαση εμπνέεται από την αρχιτεκτονική DSTC, η οποία πρότεινε έναν Dual-Side Sparse Tensor Core για αδόμητη αραιότητα και στους δύο τελεστέους [8, 9]. Ωστόσο, η παρούσα εργασία δεν περιορίζεται στην αναπαραγωγή της ιδέας. Αντίθετα, μεταφέρει τον μηχανισμό σε πραγματική RTL υλοποίηση μέσα στο Vortex, ανασχεδιάζει το datapath ώστε να ταιριάζει με τους περιορισμούς της συγκεκριμένης GPU, εισάγει άμεση πρόσβαση στη μοιραζόμενη μνήμη, μειώνει την εξάρτηση από το αρχείο καταχωρητών και συνδυάζει το υλικό με έναν βελτιστοποιημένο πυρήνα λογισμικού.

Η βασική ιδέα είναι η αντικατάσταση της οργάνωσης εσωτερικού γινομένου με μια οργάνωση εξωτερικού γινομένου. Σε κάθε βήμα, μια στήλη του A και μια γραμμή του B παράγουν ένα μερικό πλακίδιο εξόδου, το οποίο συσσωρεύεται σε έναν εσωτερικό buffer. Όταν οι τελεστέοι είναι αραιοί, τα μη μηδενικά στοιχεία συμπιέζονται και τα μηδενικά βήματα παραλείπονται. Οι θέσεις των μη μηδενικών στοιχείων αποθηκεύονται με bitmap μεταδεδομένα, ώστε τα μερικά αποτελέσματα να επιστρέφουν στις σωστές θέσεις του πυκνού πλακιδίου εξόδου.

Η συνεισφορά της εργασίας είναι διπλή. Από τη μία πλευρά, δείχνει ότι η

αδόμητη αραιότητα και στους δύο τελεστέους μπορεί να υποστηριχθεί σε μια συνθέσιμη GPU αρχιτεκτονική. Από την άλλη, δείχνει ότι η υποστήριξη αραιότητας δεν αρκεί από μόνη της. Για να υπάρξει ουσιαστική επιτάχυνση, πρέπει να μειωθούν ταυτόχρονα οι εντολές, η μεταφορά δεδομένων, η πίεση στο αρχείο καταχωρητών και το κόστος συσσώρευσης των αραιών μερικών αποτελεσμάτων. Για αυτόν τον λόγο, η TCU-OP δεν είναι μόνο ένας νέος πολλαπλασιαστικός datapath, αλλά ένα πλήρες σύστημα συν-σχεδίασης (co-design) υλικού και λογισμικού.

1.2 Θεωρητικό Υπόβαθρο

Οι GPUs οργανώνουν τον παραλληλισμό τους γύρω από μεγάλο αριθμό ελαφριών νημάτων. Τα νήματα ομαδοποιούνται σε warps, τα οποία εκτελούν την ίδια ροή εντολών σύμφωνα με το μοντέλο SIMT. Κάθε Streaming Multiprocessor ή αντίστοιχη μονάδα περιλαμβάνει εκτελεστικές μονάδες, αρχείο καταχωρητών, μοιραζόμενη μνήμη, μονάδες προγραμματισμού warps και, στις σύγχρονες αρχιτεκτονικές, εξειδικευμένες μονάδες πινάκων. Η μοιραζόμενη μνήμη λειτουργεί ως γρήγορη, ελεγχόμενη από το λογισμικό μνήμη για επαναχρησιμοποίηση δεδομένων, ενώ το αρχείο καταχωρητών παρέχει τους άμεσους τελεστέους στις εκτελεστικές μονάδες.

Σε έναν τυπικό πυρήνα GEMM, τα δεδομένα μεταφέρονται από την καθολική μνήμη στη μοιραζόμενη μνήμη, όπου επαναχρησιμοποιούνται από πολλά νήματα. Στη συνέχεια, κάθε νήμα φορτώνει ένα τμήμα των πλακιδίων σε καταχωρητές και καλεί εντολές MMA. Η προσέγγιση αυτή είναι αποδοτική όταν το κόστος μεταφοράς και προετοιμασίας των τελεστέων μπορεί να κρυφτεί πίσω από τον υπολογισμό. Όμως, όταν οι μονάδες Tensor Core είναι πολύ γρήγορες ή όταν οι πίνακες είναι μικροί, αραιοί ή χαμηλής ακρίβειας, ο χρόνος μεταφοράς και προετοιμασίας μπορεί να κυριαρχήσει.

Οι Tensor Cores έχουν σχεδιαστεί για να εκτελούν πράξεις της μορφής

$$D = A \times B + C.$$

Στην κλασική οργάνωση εσωτερικού γινομένου, κάθε στοιχείο του αποτελέσματος παράγεται ως εσωτερικό γινόμενο μιας γραμμής του A και μιας στήλης του B. Αυτή η μορφή είναι φυσική για πυκνούς πίνακες, επειδή κάθε συσσωρευτής μπορεί να παραμένει τοπικός μέχρι να ολοκληρωθεί το αντίστοιχο άθροισμα. Ωστόσο, η αξιοποίηση αδόμητης αραιότητας και στους δύο τελεστέους είναι δύσκολη, επειδή το υλικό θα έπρεπε να αντιστοιχίσει δυναμικά τα μη μηδενικά στοιχεία των δύο αραιών διανυσμάτων.

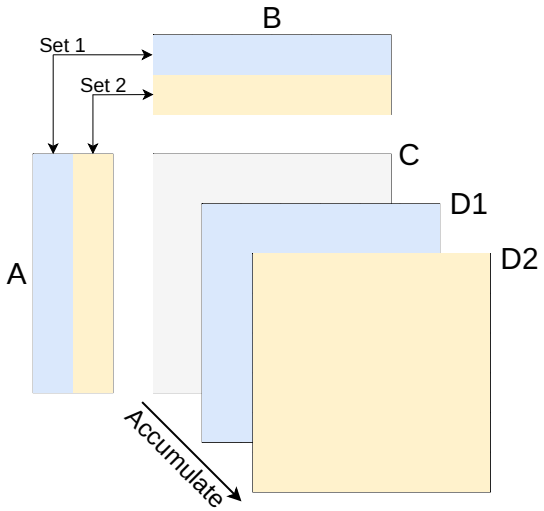
Μια εναλλακτική είναι η οργάνωση εξωτερικού γινομένου. Σε αυτήν την προσέγγιση, ο υπολογισμός προχωρά κατά μήκος της διάστασης K. Κάθε βήμα επιλέγει μια στήλη του A και την αντίστοιχη γραμμή του B, και υπολογίζει το

εξωτερικό τους γινόμενο:

$$\langle M, N\rangle = \langle M, 1\rangle \times \langle 1, N\rangle.$$

Το αποτέλεσμα είναι ένα μερικό πλακίδιο τάξης 1, το οποίο προστίθεται στο τρέχον πλακίδιο εξόδου. Μετά από όλα τα βήματα της διάστασης K, το τελικό αποτέλεσμα είναι

$$D = C + \sum_{k=0}^{K-1} A_{:,k} B_{k,:}.$$



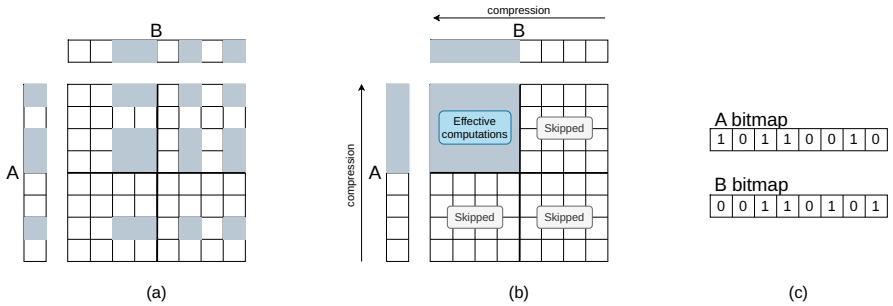
Σχήμα 1.1: Πολλαπλασιασμός πινάκων με εξωτερικό γινόμενο. Κάθε στήλη του A συνδυάζεται με την αντίστοιχη γραμμή του B, παράγοντας ένα μερικό πλακίδιο που συσσωρεύεται στο αποτέλεσμα.

Το Σχήμα 1.1 δείχνει αυτήν τη διαδικασία. Το κύριο πλεονέκτημα είναι η επαναχρησιμοποίηση των εισόδων. Κάθε στοιχείο από τη στήλη του A χρησιμοποιείται σε πολλούς πολλαπλασιασμούς με στοιχεία της γραμμής του B, και κάθε στοιχείο του B επαναχρησιμοποιείται αντίστοιχα σε πολλές γραμμές του αποτελέσματος. Η δυσκολία μεταφέρεται στη συσσώρευση: το πλακίδιο εξόδου ενημερώνεται σταδιακά και πρέπει να παραμένει διαθέσιμο σε έναν εσωτερικό συσσωρευτή ή να διαβάζεται και να γράφεται επανειλημμένα.

Η οργάνωση εξωτερικού γινομένου είναι ιδιαίτερα κατάλληλη για αδόμητη αραιότητα. Αν η επιλεγμένη στήλη του A και η επιλεγμένη γραμμή του B περιέχουν μηδενικά, τότε τα μη μηδενικά στοιχεία μπορούν να συμπιεστούν σε μικρότερα πυκνά διανύσματα. Ο πολλαπλασιασμός μεταξύ των συμπιεσμένων διανυσμάτων παραμένει κανονικός, επειδή κάθε μη μηδενικό στοιχείο του A πρέπει να πολλαπλασιαστεί με κάθε μη μηδενικό στοιχείο του B. Έτσι αποφεύγεται η ακριβή

διαδικασία εύρεσης ταιριαστών μη μηδενικών στοιχείων που θα απαιτούσε μια αραιή υλοποίηση εσωτερικού γινομένου.

Η DSTC αξιοποιεί ακριβώς αυτήν την ιδέα [8, 9]. Κάθε αραιό πλακίδιο αποθηκεύεται ως πίνακας τιμών και ως bitmap. Το bitmap έχει ένα bit ανά αρχικό στοιχείο και δηλώνει αν το στοιχείο είναι μη μηδενικό. Οι μη μηδενικές τιμές αποθηκεύονται συνεχόμενα, ώστε το υλικό να μπορεί να τις μεταφέρει και να τις καταναλώσει ως μικρότερα πυκνά διανύσματα. Για τον πίνακα A, οι τιμές οργανώνονται κατά στήλες, ενώ για τον πίνακα B, κατά γραμμές, επειδή αυτή η διάταξη ταιριάζει με το εξωτερικό γινόμενο.



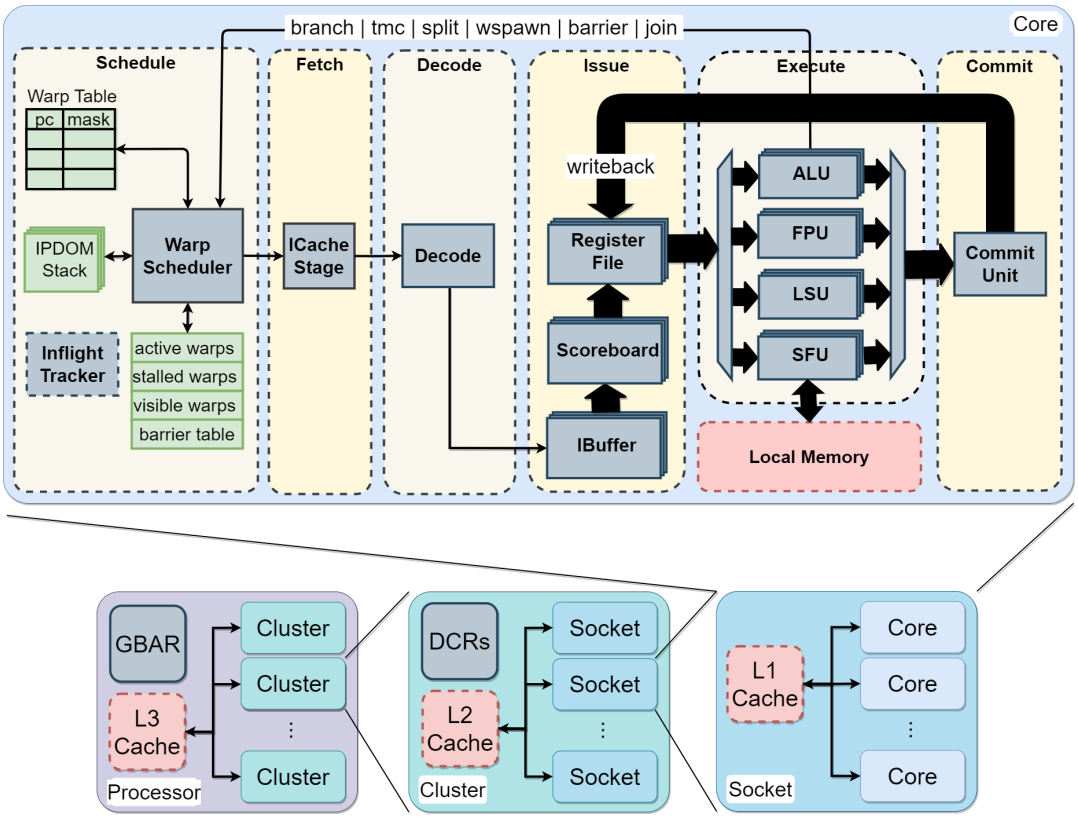
Σχήμα 1.2: Συμπίεση και bitmap αναπαράσταση αραιών τελεστέων. Τα μη μηδενικά στοιχεία αποθηκεύονται συνεχόμενα, ενώ τα bitmaps διατηρούν τις αρχικές τους θέσεις.

Στο Σχήμα 1.2 φαίνεται η βασική ευκαιρία επιτάχυνσης. Μετά τη συμπίεση, περιοχές που περιέχουν μόνο μηδενικά δεν χρειάζεται να εκτελεστούν. Το υλικό μπορεί να παραλείψει ολόκληρα βήματα εξωτερικού γινομένου, μειώνοντας τον αριθμό των πολλαπλασιασμών και την κίνηση τελεστέων. Ταυτόχρονα, τα bitmaps επιτρέπουν την ανάκτηση των αρχικών συντεταγμένων, ώστε κάθε μερικό γινόμενο να συσσωρεύεται στη σωστή θέση του πυκνού αποτελέσματος.

Για την ανάκτηση των θέσεων χρησιμοποιείται μια μονάδα που ονομάζεται Tile Index Aligner ή TIA. Η μονάδα αυτή λαμβάνει ένα bitmap, εντοπίζει τα set bits και παράγει τη διατεταγμένη λίστα των αρχικών δεικτών των μη μηδενικών στοιχείων. Οι δείκτες του A καθορίζουν τις γραμμές του αποτελέσματος, ενώ οι δείκτες του B καθορίζουν τις στήλες. Έτσι, κάθε μερικό γινόμενο συνοδεύεται από τη διεύθυνση στην οποία πρέπει να προστεθεί.

Η συσσώρευση των αραιών αποτελεσμάτων είναι ένα από τα δυσκολότερα προβλήματα. Στην πυκνή περίπτωση, τα μερικά αποτελέσματα ενός βήματος μπορούν να αντιστοιχιστούν σε διαφορετικές θύρες ενός συσσωρευτή χωρίς συγκρούσεις. Στην αραιή περίπτωση, όμως, η συμπίεση αλλάζει τις θέσεις των αποτελεσμάτων και πολλά μερικά γινόμενα μπορεί να στοχεύουν την ίδια τράπεζα στον ίδιο κύκλο. Αυτές οι συγκρούσεις τραπεζών δημιουργούν stalls και μειώνουν την πραγματική απόδοση. Για αυτόν τον λόγο, η αρχιτεκτονική χρειάζεται ουρές, crossbar και κατάλληλη χαρτογράφηση των συσσωρευτών.

Η πλατφόρμα της εργασίας είναι το Vortex. Το Vortex είναι μια ανοικτή, συνθέσιμη RISC-V GPGPU, γραμμένη σε Verilog, και αποτελεί κατάλληλη βάση για μικροαρχιτεκτονική έρευνα [19]. Η ύπαρξη RTL υλοποίησης επιτρέπει κύκλο-ακριβή αξιολόγηση με Verilator, καθώς και πιο ρεαλιστική παρατήρηση περιορισμών όπως stalls, συγκρούσεις πόρων, καθυστερήσεις buffers και αλληλεπίδραση με τη μνήμη. Το βασικό Vortex TCU υποστηρίζει πράξεις MMA, αλλά ακολουθεί register-file-based dataflow: οι τελεστέοι φορτώνονται πρώτα σε καταχωρητές, η εντολή Tensor Core καταναλώνει τα register fragments, και τα αποτελέσματα επιστρέφουν ξανά μέσω του αρχείου καταχωρητών.



Σχήμα 1.3: Υψηλού επιπέδου μικροαρχιτεκτονική του Vortex. Η ανοικτή RTL υλοποίηση επιτρέπει την ενσωμάτωση και αξιολόγηση νέων μηχανισμών Tensor Core.

Το βασικό αυτό μοντέλο έχει σημαντικούς περιορισμούς.

- Η χρήση του αρχείου καταχωρητών για τη μεταφορά πλακιδίων αυξάνει τον αριθμό εντολών και την πίεση στους καταχωρητές.
- Η διάσπαση των εντολών MMA σε πολλές μικρο-εντολές αυξάνει την κίνηση στο frontend και στο pipeline.
- Το μέγεθος του πλακιδίου εξόδου περιορίζεται από τους καταχωρητές που μπορεί να διαθέσει ένα warp.

- Η σύγχρονη εκτέλεση περιορίζει την επικάλυψη με ανεξάρτητη εργασία.
- Συνολικά, οι παραπάνω περιορισμοί οδηγούν σε πολύ χαμηλή πραγματική χρησιμοποίηση των πολλαπλασιαστών του baseline TCU, ακόμη και όταν υπάρχει εξειδικευμένο υλικό.

Η βιβλιογραφία γύρω από το θέμα δείχνει ότι υπάρχουν διαφορετικές κατευθύνσεις βελτίωσης. Η NVIDIA Ampere υποστηρίζει δομημένη 2:4 αραιότητα [7, 6]. Η VEGETA μελετά επεκτάσεις για δομημένη αραιότητα σε CPU matrix engines [21]. Η Cooperative Warp Execution βελτιώνει την αξιοποίηση του Vortex Tensor Core μέσω συνεργασίας πολλών warps [22]. Η Virgo αποσυνδέει τις μονάδες πινάκων από τους SIMT πυρήνες και χρησιμοποιεί μοιραζόμενη μνήμη και ειδική μνήμη συσσώρευσης [23].

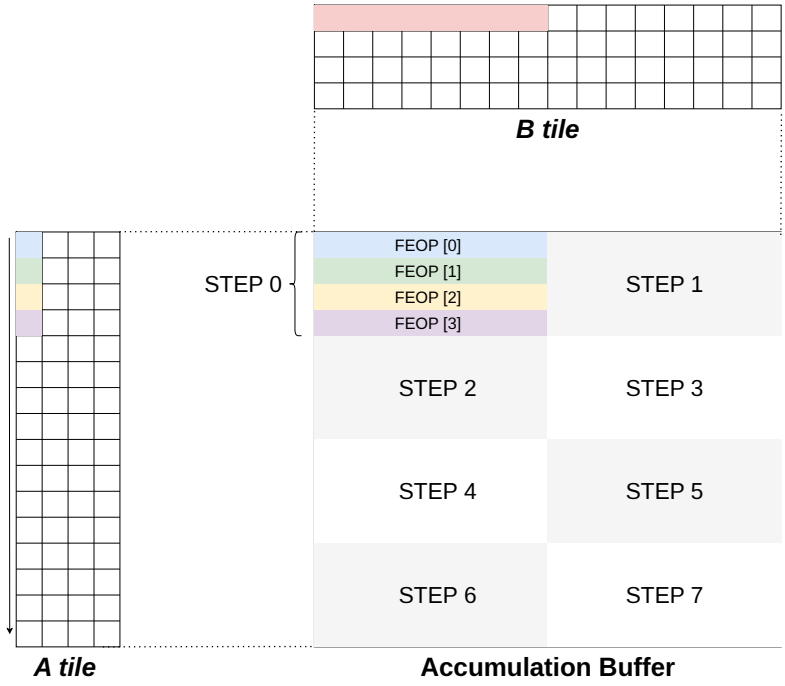
Σε σχέση με αυτές τις εργασίες, η TCU-OP συνδυάζει χαρακτηριστικά που συνήθως εμφανίζονται ξεχωριστά: υποστήριξη αδόμητης αραιότητας και στους δύο τελεστέους, RTL ενσωμάτωση σε GPU, άμεση πρόσβαση στη μοιραζόμενη μνήμη, εσωτερικό συσσωρευτή, παράκαμψη του αρχείου καταχωρητών και ασύγχρονη εκτέλεση. Αυτός ο συνδυασμός είναι το βασικό σημείο διαφοροποίησης της εργασίας.

1.3 Λεπτομέρειες της Υλοποίησης

Η υλοποίηση της TCU-OP ξεκινά από την αντικατάσταση του εσωτερικού γινομένου datapath με ένα εξωτερικού γινομένου datapath. Η βασική μονάδα ονομάζεται FEOP από το Fused-Element Outer-Product. Σε αντίθεση με μια μονάδα εσωτερικού γινομένου, η οποία παράγει ένα αποτέλεσμα μετά από άθροιση πολλών γινομένων, η FEOP παίρνει ένα στοιχείο από τον πίνακα A και πολλά στοιχεία από τον πίνακα B, και παράγει μια γραμμή μερικών γινομένων. Πολλές FEOP μονάδες λειτουργούν παράλληλα και παράγουν ένα υποπλακίδιο του εξωτερικού γινομένου σε κάθε βήμα.

Ο υπολογισμός οργανώνεται σε sets και steps. Ένα set αντιστοιχεί σε μία στήλη του A και μία γραμμή του B, καθένα 32 στοιχείων. Επειδή ένα πλήρες εξωτερικό γινόμενο 32×32 θα απαιτούσε 1024 βαθμωτούς πολλαπλασιασμούς, δεν είναι πρακτικό να εκτελείται σε έναν κύκλο. Για αυτόν τον λόγο, κάθε set χωρίζεται σε μικρότερα steps. Κάθε step έχει διαστάσεις $m \times n$ και παράγει $m \cdot n$ μερικά γινόμενα.

Το Σχήμα 1.4 παρουσιάζει αυτή τη λογική. Η επιλογή των m και n είναι κρίσιμη, επειδή καθορίζει ταυτόχρονα την κορυφαία υπολογιστική απόδοση, το κόστος του datapath και την λεπτομέρεια με την οποία μπορεί να αξιοποιηθεί η αραιότητα. Μεγαλύτερο m σημαίνει ότι κάθε βήμα καλύπτει μεγαλύτερο τμήμα της

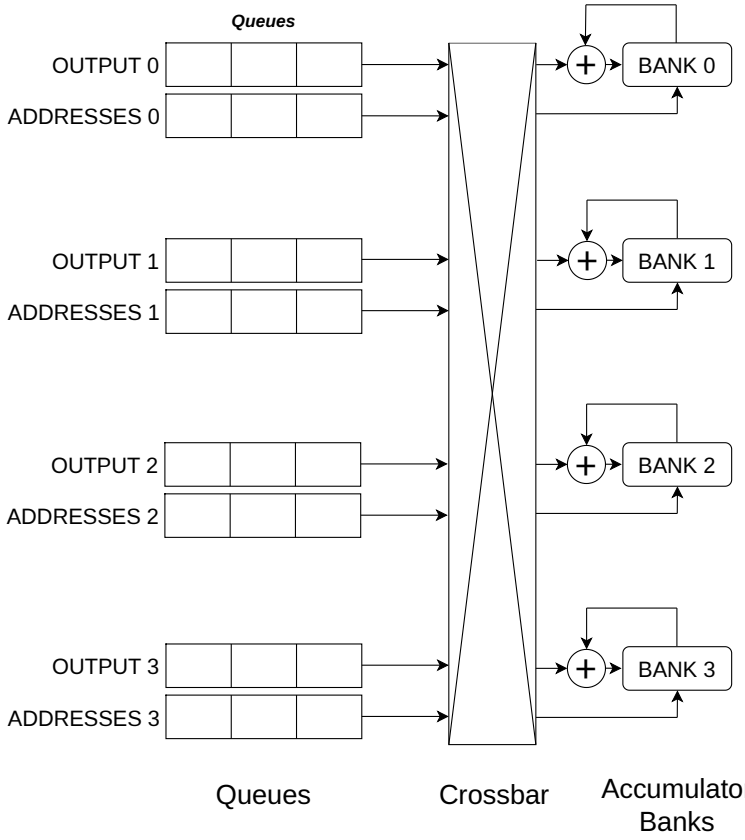


Σχήμα 1.4: Παράδειγμα εκτέλεσης ενός βήματος FEOP. Το πλήρες πλακίδιο εξόδου καλύπτεται σταδιακά από μικρότερα υποπλακίδια εξωτερικού γινομένου.

στήλης του A, άρα η αραιότητα στον A αξιοποιείται με πιο χονδρή λεπτομέρεια. Μεγαλύτερο n έχει αντίστοιχο αποτέλεσμα για τον B. Εφόσον ο συνολικός αριθμός πολλαπλασιαστών είναι σταθερός, η αύξηση της μίας διάστασης μειώνει την άλλη.

Η δεύτερη βασική δομή είναι ο συσσωρευτής της TCU-OP. Επειδή η εξωτερικού γινομένου εκτέλεση παράγει μερικά αποτελέσματα που ενημερώνουν σταδιακά το πλακίδιο εξόδου, το αποτέλεσμα πρέπει να παραμένει τοπικά μέσα στη μονάδα μέχρι να ολοκληρωθούν όλα τα sets της διάστασης K. Ο συσσωρευτής έχει λογικές διαστάσεις 32×32, με στοιχεία 32 bit. Κάθε είσοδος αντιστοιχεί σε ένα στοιχείο του πλακιδίου εξόδου και ενημερώνεται κάθε φορά που παράγεται νέο μερικό γινόμενο για την ίδια θέση.

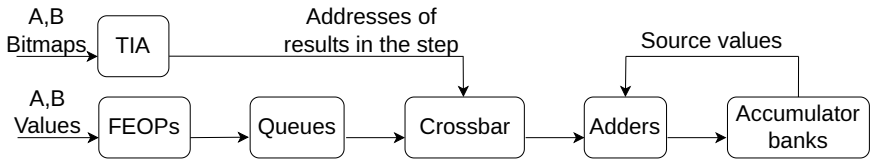
Ο συσσωρευτής είναι οργανωμένος σε θύρες, ώστε πολλά μερικά αποτελέσματα να ενημερώνονται παράλληλα. Στην πυκνή εκτέλεση, η χαρτογράφηση των διευθύνσεων μπορεί να επιλεγεί έτσι ώστε τα αποτελέσματα ενός βήματος να πηγαίνουν σε διαφορετικές θύρες. Στην αραιή εκτέλεση, όμως, τα μη μηδενικά στοιχεία έχουν συμπιεστεί και οι πραγματικές τους θέσεις ανακτώνται από τα bitmaps. Αυτό δημιουργεί ακανόνιστες διευθύνσεις και πιθανές συγκρούσεις τραπεζών. Για να αντιμετωπιστεί το πρόβλημα, κάθε έξοδος του FEOP τοποθετείται πρώτα σε ουρά και στη συνέχεια περνά από ένα crossbar που τη δρομολογεί στη σωστή τράπεζα.



Σχήμα 1.5: Οργάνωση του συσσωρευτή της TCU-OP. Οι ουρές απορροφούν προσωρινές συγκρούσεις, το crossbar δρομολογεί τα μερικά αποτελέσματα και οι τοπικοί αθροιστές ενημερώνουν τις θύρες του συσσωρευτή.

Η συγκεκριμένη υλοποίηση διαφοροποιείται από την αρχική DSTC ως προς τη θέση των αθροιστών. Στην TCU-OP, το crossbar βρίσκεται πριν από τους αθροιστές, και κάθε αθροιστής είναι τοπικός σε μια τράπεζα του συσσωρευτή. Έτσι, κάθε μερικό αποτέλεσμα δρομολογείται μία φορά στη σωστή τράπεζα, προστίθεται στην τρέχουσα τιμή και γράφεται πίσω τοπικά. Αυτή η οργάνωση μειώνει την κυκλοφορία στο crossbar και απλοποιεί το μονοπάτι εγγραφής.

Για την υποστήριξη αραιότητας, η TCU-OP χρησιμοποιεί δύο μονάδες TIA, μία για τον A και μία για τον B. Οι μονάδες αυτές λαμβάνουν τα bitmaps, παράγουν τους αρχικούς δείκτες των μη μηδενικών στοιχείων και υπολογίζουν πόσα έγκυρα στοιχεία υπάρχουν στην τρέχουσα στήλη ή γραμμή. Αυτή η πληροφορία επιτρέπει στο υλικό να αποφασίζει ποια βήματα έχουν χρήσιμο έργο και ποια μπορούν να παραλειφθούν. Σε αντίθεση με την αρχική DSTC, όπου η παράλειψη εκτίθεται σε μεγάλο βαθμό στο λογισμικό μέσω predicated instructions, εδώ ο προγραμματισμός των αραιών βημάτων μεταφέρεται μέσα στο υλικό.

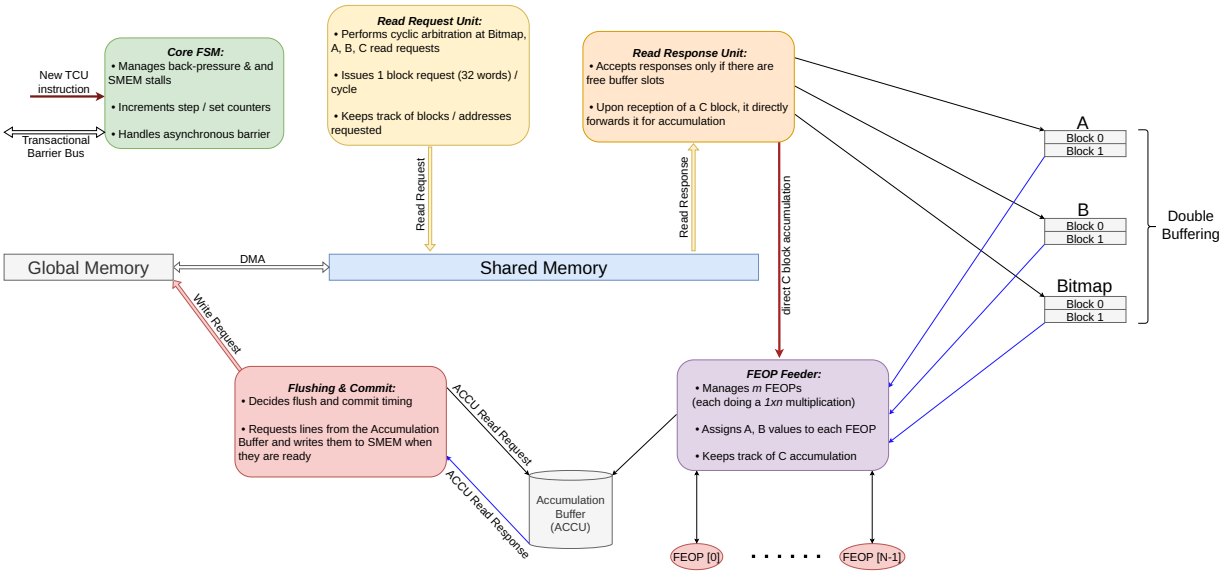


Σχήμα 1.6: Υψηλού επιπέδου pipeline υπολογισμού της TCU-OP. Τα bitmaps αποκωδικοποιούνται, τα FEOP παράγουν μερικά γινόμενα, και ο συσσωρευτής ενημερώνει το πλακίδιο εξόδου.

Η κεντρική μονάδα ελέγχου της TCU-OP διαχειρίζεται την αλληλουχία των φάσεων. Αρχικά, φορτώνει το πλακίδιο C στον συσσωρευτή, ώστε να χρησιμοποιηθεί ως αρχική τιμή ή bias. Στη συνέχεια, επεξεργάζεται τις στήλες του A και τις γραμμές του B κατά μήκος της διάστασης K. Για κάθε set, επιλέγει τα κατάλληλα συμπιεσμένα ή ασυμπίεστα δεδομένα, ενεργοποιεί τα FEOP βήματα που περιέχουν χρήσιμη εργασία και προωθεί τα αποτελέσματα στον συσσωρευτή. Όταν ολοκληρωθούν όλα τα sets, η μονάδα περνά στη φάση flush, όπου γράφει το τελικό πλακίδιο D στη μνήμη.

Ένα κεντρικό χαρακτηριστικό της υλοποίησης είναι η άμεση πρόσβαση στη μοιραζόμενη μνήμη. Το baseline Vortex TCU απαιτεί από κάθε warp να φορτώσει τα fragments των πινάκων σε καταχωρητές πριν εκδώσει την εντολή MMA. Αυτό οδηγεί σε πολλές εντολές φόρτωσης, υπολογισμούς διευθύνσεων και μεγάλη χρήση καταχωρητών. Η TCU-OP ακολουθεί διαφορετική προσέγγιση: τα πλακίδια A, B, C και τα bitmaps μεταφέρονται πρώτα στη μοιραζόμενη μνήμη μέσω του μηχανισμού DXA, δηλαδή της DMA-like υποδομής του Vortex. Στη συνέχεια, η TCU-OP διαβάζει τα δεδομένα απευθείας από τη μοιραζόμενη μνήμη, χωρίς να περάσουν από το αρχείο καταχωρητών.

Αυτή η επιλογή μειώνει σημαντικά την πίεση στους καταχωρητές και τον αριθμό εντολών. Τα warps δεν χρειάζεται να υπολογίσουν λεπτομερείς per-lane διευθύνσεις ούτε να γεμίσουν πολλούς καταχωρητές με fragments. Αντίθετα, η εντολή της TCU-OP περιέχει βασικές διευθύνσεις, δείκτες μεταδεδομένων, πληροφορία τύπου δεδομένων, κατάσταση αραιότητας και αναγνωριστικό φράγματος συγχρονισμού. Το υλικό αναλαμβάνει την ανάγνωση, την αποκωδικοποίηση, τον προγραμματισμό βημάτων και τη συσσώρευση.



Σχήμα 1.7: Συνολική οργάνωση της προτεινόμενης TCU-OP και σύνδεση με το σύστημα μνήμης.

Το Σχήμα 1.7 συνοψίζει τη συνολική αρχιτεκτονική. Η μονάδα περιλαμβάνει λογική αιτήσεων και αποκρίσεων μνήμης, buffers για τους τελεστέους και τα bitmaps, τον πυρήνα ελέγχου, το FEOP feeder, τις μονάδες TIA, τον συσσωρευτή, τις ουρές, το crossbar και τη λογική ολοκλήρωσης. Οι αιτήσεις προς τη μοιραζόμενη μνήμη γίνονται με κυκλικό διαιτητή και τα δεδομένα αποθηκεύονται σε μικρούς εσωτερικούς buffers. Ξεχωριστά streams επιτρέπουν διαφορετικούς ρυθμούς κατανάλωσης για A, B και bitmap δεδομένα, κάτι απαραίτητο όταν οι δύο τελεστέοι έχουν διαφορετικά επίπεδα αραιότητας.

Η μονάδα υποστηρίζει τρεις τρόπους εκτέλεσης:

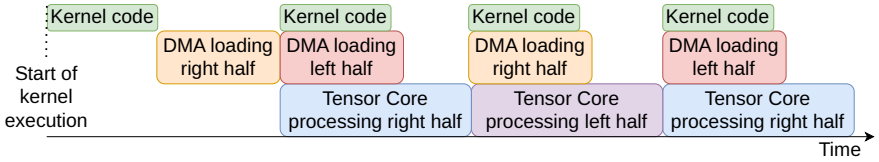
- **Ασυμπίεστος επί ασυμπίεστο:** αντιστοιχεί σε πυκνή εκτέλεση. Δεν χρησιμοποιούνται bitmaps και όλα τα βήματα εξωτερικού γινομένου εκτελούνται κανονικά.

- **Συμπιεσμένος επί συμπιεσμένο:** τόσο ο A όσο και ο B αποθηκεύονται σε συμπιεσμένη μορφή μαζί με τα αντίστοιχα bitmaps. Αυτός ο τρόπος μεγιστοποιεί τη μείωση υπολογισμών και μεταφοράς δεδομένων, αλλά απαιτεί προεπεξεργασία και των δύο τελεστέων.

- **Ασυμπίεστος επί συμπιεσμένο:** ο B είναι συμπιεσμένος, ενώ ο A μεταφέρεται ασυμπίεστος και συμπιέζεται δυναμικά μέσα στην TCU. Αυτός ο τρόπος στοχεύει στο συνηθισμένο σενάριο νευρωνικών δικτύων, όπου τα βάρη είναι στατικά και μπορούν να συμπιεστούν offline, ενώ οι ενεργοποιήσεις είναι δυναμικές.

Επειδή η TCU-OP εκτελείται ασύγχρονα, απαιτείται προσεκτικός συγχρονισμός. Το warp που εκδίδει την εντολή δεν χρειάζεται να περιμένει αμέσως την ολοκλήρωση της πράξης. Μπορεί να συνεχίσει με ανεξάρτητη εργασία και να συγχρονιστεί αργότερα. Για την ορθότητα, η μονάδα συνδέεται με τον μηχανισμό transactional barriers του Vortex. Όταν εκδίδεται η εντολή, καταχωρείται μια εκκρεμής συναλλαγή στο αντίστοιχο φράγμα. Όταν η TCU-OP ολοκληρώσει τον υπολογισμό και γράψει το αποτέλεσμα, απελευθερώνει τη συναλλαγή. Έτσι αποφεύγονται hazards ανάγνωσης μετά από εγγραφή ή εγγραφής πάνω σε εκκρεμές αποτέλεσμα.

Η απόδοση του υλικού εξαρτάται και από τον πυρήνα λογισμικού. Ο προτεινόμενος πυρήνας χρησιμοποιεί double buffering στη μοιραζόμενη μνήμη ώστε να επικαλύπτει τρεις φάσεις: παραγωγή μεταδεδομένων από το λογισμικό, μεταφορές DMA μέσω DXA, και ασύγχρονη εκτέλεση TCU. Όσο η TCU-OP επεξεργάζεται δεδομένα από το ένα μισό της μοιραζόμενης μνήμης, το DMA μεταφέρει τα επόμενα πλακίδια στο άλλο μισό. Με αυτόν τον τρόπο, το κόστος μεταφοράς κρύβεται πίσω από τον υπολογισμό, όταν υπάρχει αρκετή εργασία.



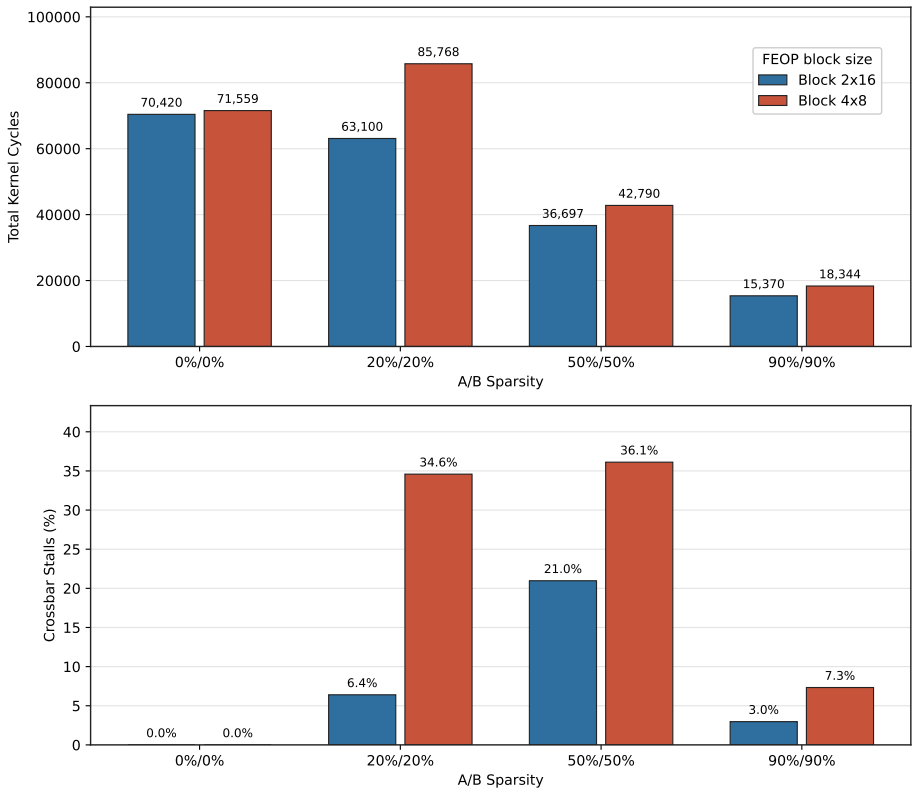
Σχήμα 1.8: Pipeline του βελτιστοποιημένου πυρήνα. Η παραγωγή μεταδεδομένων, οι DMA μεταφορές και η ασύγχρονη εκτέλεση της TCU-OP επικαλύπτονται.

Ο πυρήνας αλλάζει επίσης τον τρόπο συσσώρευσης κατά μήκος της διάστασης K. Αντί να αρχικοποιεί και να αδειάζει το πλακίδιο εξόδου για κάθε K-tile, η TCU-OP κρατά το πλακίδιο στον εσωτερικό συσσωρευτή μέχρι να ολοκληρωθεί ολόκληρη η διάσταση K. Έτσι, το κόστος αρχικοποίησης και τελικής εγγραφής πληρώνεται μία φορά ανά πλακίδιο εξόδου και αποσβένεται όσο μεγαλώνει το K. Επιπλέον, επειδή ο συσσωρευτής δεν περιορίζεται από το πλήθος καταχωρητών ενός warp, η TCU-OP μπορεί να χρησιμοποιήσει πλακίδιο εξόδου 32×32, αυξάνοντας την επαναχρησιμοποίηση των εισόδων σε σχέση με μικρότερα πλακίδια.

Η μικροαρχιτεκτονική παραμετροποίηση είναι κρίσιμη. Το baseline Vortex TCU σε διαμόρφωση 32 νημάτων περιλαμβάνει 128 πολλαπλασιαστές, όμως η

αξιολόγηση δείχνει ότι αυτοί οι πολλαπλασιαστές παραμένουν σε μεγάλο βαθμό ανενεργοί, επειδή η εκτέλεση περιορίζεται από τη μεταφορά δεδομένων. Για αυτόν τον λόγο, η TCU-OP επιλέγει συνολικό μέγεθος βήματος $m \times n = 32$, δηλαδή 32 μερικά γινόμενα ανά κύκλο. Αυτό μειώνει σημαντικά το κόστος πολλαπλασιαστών, αθροιστών, ουρών, τραπεζών και crossbar, χωρίς να μειώνει ουσιαστικά την απόδοση στα μετρούμενα memory-bound workloads.

Μετά τη σταθεροποίηση του συνολικού μεγέθους, εξετάζονται διαφορετικά σχήματα βήματος. Οι βασικοί υποψήφιοι είναι 2×16 και 4×8. Το 4×8 παρέχει πιο ισορροπημένη λεπτομέρεια αραιότητας μεταξύ A και B, αλλά έχει μεγαλύτερο κόστος flush και περισσότερες συγκρούσεις στο crossbar. Η ανάλυση των υπολοίπων modulo δείχνει ότι η συμπιεσμένη αραιή πρόσβαση οδηγεί σε λιγότερες μοναδικές θύρες για το 4×8, άρα σε περισσότερα conflicts κατά μέσο όρο. Τα πειραματικά αποτελέσματα επιβεβαιώνουν ότι το 2×16 έχει μικρότερο χρόνο εκτέλεσης στις εξεταζόμενες περιπτώσεις.



Σχήμα 1.9: Επίδραση των διαστάσεων βήματος στον χρόνο εκτέλεσης και στα crossbar stalls. Η διαμόρφωση 2×16 επιλέγεται ως πιο αποδοτική.

Τέλος, εξετάζεται το βάθος των ουρών πριν από το crossbar. Μεγαλύτερο βάθος επιτρέπει την απορρόφηση περισσότερων προσωρινών συγκρούσεων, αλλά αυξάνει το κόστος αποθήκευσης και κατανάλωσης ενέργειας. Η αξιολόγηση

δείχνει ότι το βάθος $Q = 4$ παρέχει το μεγαλύτερο μέρος του οφέλους των βαθύτερων ουρών, χωρίς περιττό κόστος. Έτσι, η τελική διαμόρφωση της αξιολόγησης χρησιμοποιεί βήμα 2×16, συνολικά 32 μερικά γινόμενα ανά κύκλο, συσσωρευτή 32×32 και ουρές βάθους 4.

1.4 Αξιολόγηση

Η αξιολόγηση της TCU-OP γίνεται σε επίπεδο RTL μέσα στο Vortex, χρησιμοποιώντας κύκλο-ακριβή προσομοίωση. Το βασικό workload είναι ο πολλαπλασιασμός πινάκων

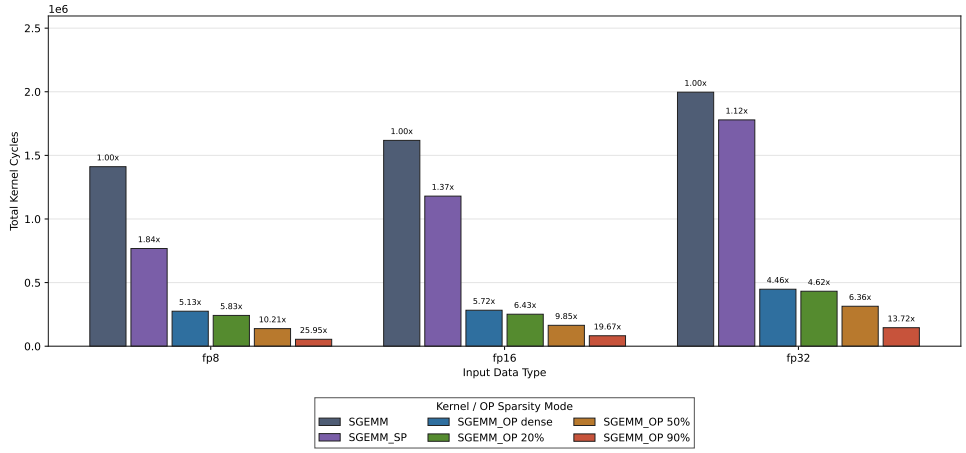
$$D = A \times B + C,$$

όπου A έχει διαστάσεις $M \times K$, B έχει διαστάσεις $K \times N$, και τα C και D έχουν διαστάσεις $M \times N$. Η αξιολόγηση εστιάζει σε end-to-end μετρήσεις πυρήνα, δηλαδή περιλαμβάνει αρχικοποίηση, μεταφορές, συγχρονισμό, υπολογισμό και τελική εγγραφή. Αυτό είναι σημαντικό, επειδή πολλές επιταχύνσεις που φαίνονται μεγάλες σε απομονωμένο datapath μπορούν να μειωθούν όταν ληφθεί υπόψη το πλήρες κόστος εκτέλεσης.

Η προτεινόμενη υλοποίηση συγκρίνεται με δύο baselines του Vortex. Το πρώτο είναι το SGEMM, δηλαδή η πυκνή Tensor Core υλοποίηση με register-file-based dataflow. Το δεύτερο είναι το SGEMM_SP, το οποίο υλοποιεί δομημένη 2:4 αραιότητα στον τελεστή B, παρόμοια με τον μηχανισμό των NVIDIA Sparse Tensor Cores. Η προτεινόμενη υλοποίηση αξιολογείται ως SGEMM_OP σε πυκνή και αραιή εκτέλεση, με διαφορετικούς τύπους δεδομένων και επίπεδα αραιότητας.

Οι βασικές μετρικές είναι οι συνολικοί κύκλοι πυρήνα, ο συνολικός αριθμός εντολών, οι προσπελάσεις στη μνήμη, οι DMA συναλλαγές και η χρησιμοποίηση των πολλαπλασιαστών της Tensor Core. Οι κύκλοι δείχνουν την πραγματική απόδοση. Οι εντολές δείχνουν την πίεση στο frontend, στον scheduler και στο pipeline. Οι προσπελάσεις μνήμης δείχνουν την πίεση στη DRAM και στην L1 data cache. Η χρησιμοποίηση πολλαπλασιαστών δείχνει αν το datapath κάνει χρήσιμη αριθμητική εργασία ή περιμένει δεδομένα και συγχρονισμό.

Στην κύρια σύγκριση απόδοσης, το μέγεθος του προβλήματος είναι $128 \times 128 \times 512$. Τα αποτελέσματα του Σχήματος 1.10 δείχνουν ότι η SGEMM_OP υπερτερεί των δύο baselines σε όλους τους τύπους δεδομένων. Ακόμη και στην πυκνή περίπτωση, όπου δεν παραλείπεται κανένα αραιό βήμα, η SGEMM_OP επιτυγχάνει επιταχύνσεις $4.46\times$, $5.72\times$ και $5.13\times$ για FP32, FP16 και FP8 αντίστοιχα, σε σχέση με το πυκνό SGEMM. Αυτό δείχνει ότι μεγάλο μέρος του οφέλους δεν προέρχεται μόνο από την αραιότητα, αλλά από την καλύτερη τροφοδοσία τελεστέων, την παράκαμψη του αρχείου καταχωρητών και την επαναχρησιμοποίηση μέσα στον συσσωρευτή.



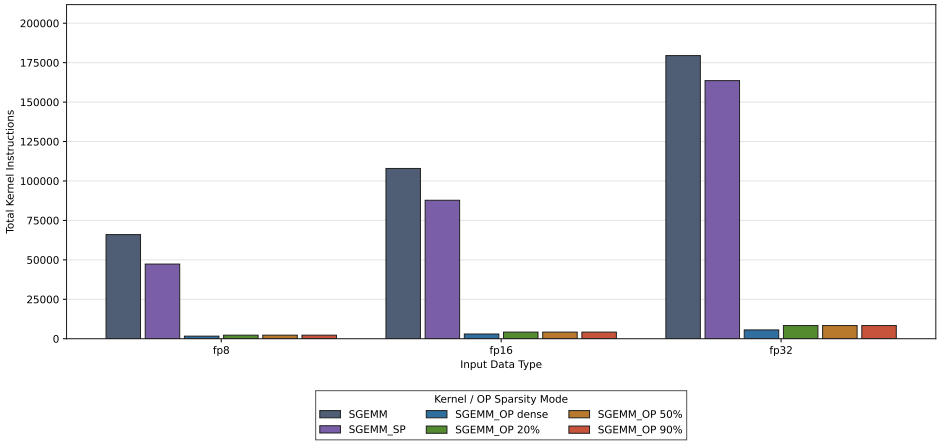
Σχήμα 1.10: Συνολικοί κύκλοι πυρήνα για SGEMM, SGEMM_SP και SGEMM_OP σε διαφορετικούς τύπους δεδομένων.

Η δομημένη αραιότητα του SGEMM_SP προσφέρει περιορισμένο όφελος. Οι επιταχύνσεις του είναι $1.12\times$, $1.37\times$ και $1.84\times$ για FP32, FP16 και FP8. Το αποτέλεσμα αυτό είναι αναμενόμενο, επειδή το σχήμα $2{:}4$ μπορεί θεωρητικά να φτάσει το $2\times$, αλλά μόνο όταν το υπόλοιπο κόστος είναι πλήρως κρυμμένο. Στην πράξη, η μεταφορά δεδομένων, οι εντολές και η register-file-based οργάνωση περιορίζουν το πραγματικό όφελος.

Καθώς αυξάνεται η αδόμητη αραιότητα και στους δύο τελεστέους, η SGEMM_OP επιταχύνεται περισσότερο. Στο $50\%/50\%$ αραιότητας, οι επιταχύνσεις σε σχέση με το πυκνό SGEMM φτάνουν $6.36\times$, $9.85\times$ και $10.21\times$ για FP32, FP16 και FP8. Στο $90\%/90\%$, οι αντίστοιχες επιταχύνσεις γίνονται $13.72\times$, $19.67\times$ και $25.95\times$. Η FP8 περίπτωση ωφελείται περισσότερο επειδή τα στενότερα δεδομένα μειώνουν την κίνηση μνήμης και επιτρέπουν αποτελεσματικότερη χρήση των blocks μεταφοράς.

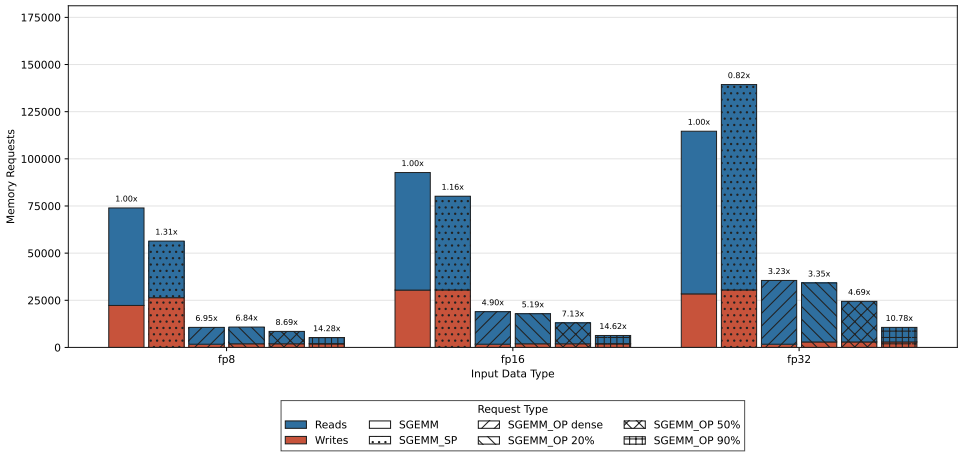
Ο αριθμός εντολών μειώνεται ακόμη πιο έντονα. Στα baselines, κάθε warp εκτελεί πολλές εντολές για υπολογισμό διευθύνσεων, φόρτωση fragments σε καταχωρητές, έκδοση MMA και αποθήκευση αποτελεσμάτων. Επιπλέον, οι υψηλού επιπέδου MMA εντολές μεταφράζονται εσωτερικά σε πολλές μικρο-εντολές. Η SGEMM_OP αντικαθιστά μεγάλο μέρος αυτής της ακολουθίας με DMA μεταφορές και μία συμπαγή ασύγχρονη εντολή TCU. Ως αποτέλεσμα, η εργασία αναφέρει συνολική μείωση εντολών έως $29.3\times$, κάτι που μειώνει την πίεση στο frontend και απελευθερώνει χώρο για μελλοντική επικάλυψη με ανεξάρτητη εργασία.

Η μνήμη αποτελεί δεύτερη καθοριστική μετρική. Η SGEMM_OP μειώνει τις προσπελάσεις DRAM επειδή χρησιμοποιεί μεγαλύτερο πλακίδιο εξόδου και συνεπώς αυξάνει την επαναχρησιμοποίηση των εισόδων. Επιπλέον, οι μεταφορές γίνονται συνεχόμενα μέσω DMA, άρα αξιοποιούν καλύτερα τις γραμμές μνήμης



Σχήμα 1.11: Συνολικός αριθμός εντολών. Η SGEMM_OP μειώνει δραστικά τις εντολές επειδή αποφεύγει το explicit register-file staging.

από τις per-lane φορτώσεις των baselines. Στα αραιά σενάρια, η συμπιεσμένη αναπαράσταση μεταφέρει μόνο τα μη μηδενικά στοιχεία, μαζί με τα bitmaps. Παρότι τα bitmaps προσθέτουν metadata traffic, σε υψηλή αραιότητα το συνολικό κόστος μειώνεται σημαντικά. Η εργασία αναφέρει μείωση κίνησης μνήμης έως $30.2\times$ σε σχέση με το baseline.

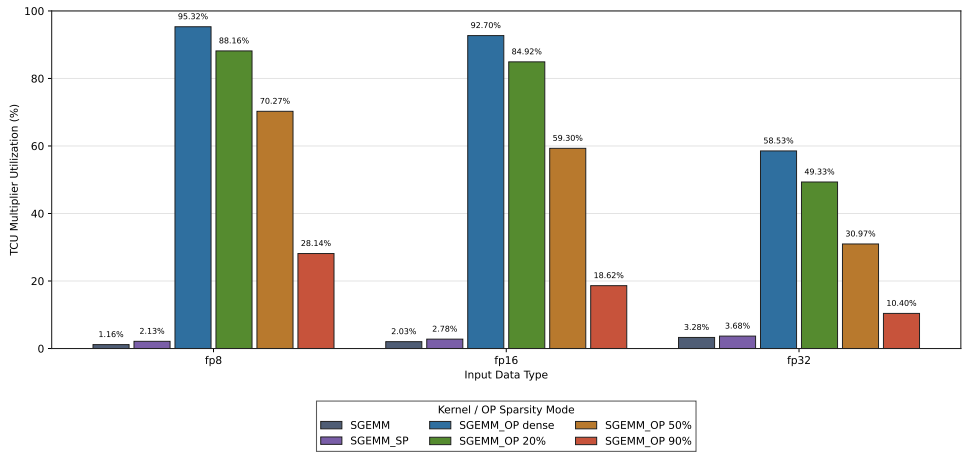


Σχήμα 1.12: Συνολικές off-chip προσπελάσεις μνήμης. Η SGEMM_OP μειώνει την κίνηση μέσω μεγαλύτερης επαναχρησιμοποίησης και συνεχόμενων DMA μεταφορών.

Η δραστηριότητα της L1 data cache ακολουθεί διαφορετική τάση. Επειδή η SGEMM_OP μεταφέρει τους κύριους τελεστέους μέσω DMA προς τη μοιραζόμενη μνήμη, δεν επιβαρύνει την D-cache με τις ίδιες φορτώσεις που εκτελούν τα baselines. Αυτό μειώνει το cache pollution και αφήνει την cache κυρίως για bookkeeping, μεταδεδομένα και βοηθητικές προσπελάσεις του πυρήνα. Η δραστηριότητα της D-cache παραμένει σχεδόν σταθερή καθώς αλλάζει η αραιότητα, επειδή η

sparsity-dependent μεταφορά τελεστέων δεν γίνεται με κανονικές cacheable load instructions.

Η χρησιμοποίηση των πολλαπλασιαστών δείχνει καθαρά τη διαφορά μεταξύ των αρχιτεκτονικών. Στα baselines του Vortex, η μέγιστη χρησιμοποίηση είναι κάτω από 4%: 3.28% για το πυκνό SGEMM και 3.68% για το SGEMM_SP. Αυτό σημαίνει ότι οι πολλαπλασιαστές υπάρχουν αλλά σπάνια παράγουν χρήσιμα αποτελέσματα, επειδή το pipeline περιμένει δεδομένα ή εκτελεί βοηθητικές εντολές. Αντίθετα, η SGEMM_OP φτάνει έως 95.32% χρησιμοποίηση στην πυκνή FP8 περίπτωση. Το αποτέλεσμα αυτό οφείλεται στο double buffering, στην άμεση πρόσβαση στη μοιραζόμενη μνήμη και στη διατήρηση του αποτελέσματος στον συσσωρευτή.

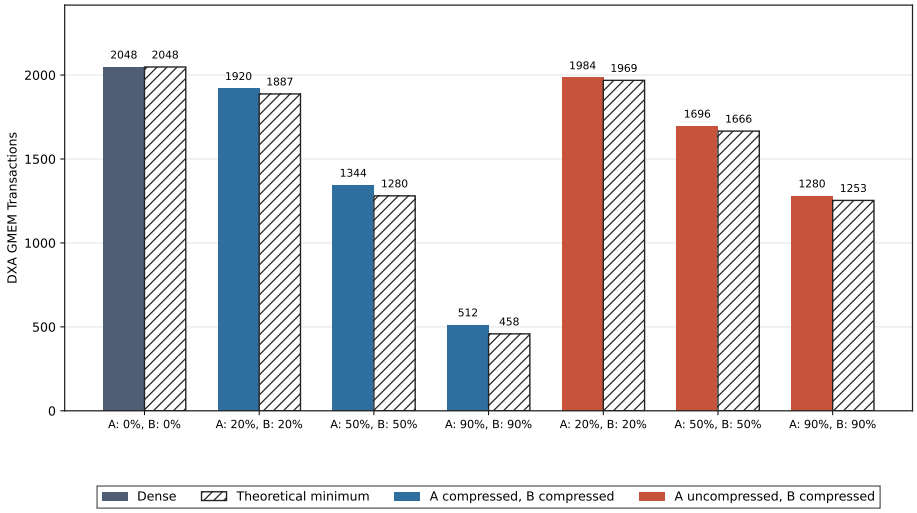


Σχήμα 1.13: Χρησιμοποίηση πολλαπλασιαστών Tensor Core. Η SGEMM_OP διατηρεί το datapath ενεργό πολύ συχνότερα από τα baselines.

Καθώς η αραιότητα αυξάνεται, η χρησιμοποίηση των πολλαπλασιαστών της SGEMM_OP μειώνεται. Αυτό δεν είναι αρνητικό αποτέλεσμα. Σημαίνει ότι το χρήσιμο αριθμητικό έργο μειώνεται και η μονάδα ολοκληρώνει πιο γρήγορα τα βήματα. Σε πολύ υψηλή αραιότητα, το σταθερό κόστος συγχρονισμού, μεταδεδομένων και μεταφοράς γίνεται πιο ορατό. Επιπλέον, ο φόρτος υπολογισμού μειώνεται πιο γρήγορα σε σχέση με τη μεταφορά δεδομένων. Έτσι, σε μεγάλη αραιότητα ο πυρήνας γίνεται ξανά περισσότερο memory-bound.

Οι DMA συναλλαγές επιβεβαιώνουν ότι η συμπιεσμένη αναπαράσταση μειώνει πραγματικά την κίνηση δεδομένων. Στην πυκνή περίπτωση, και οι δύο τελεστέοι μεταφέρονται ασυμπίεστοι. Στον συμπιεσμένο επί συμπιεσμένο τρόπο, όσο αυξάνεται η αραιότητα, μεταφέρονται λιγότερες τιμές για A και B, μαζί με τα απαραίτητα bitmaps. Στο $90\%/90\%$, ο αριθμός συναλλαγών πέφτει από 2048 στην πυκνή περίπτωση σε 512, δηλαδή μειώνεται κατά 75%. Στον ασυμπίεστο επί

συμπιεσμένο τρόπο, η κίνηση είναι ενδιάμεση: ο B ωφελείται από τη συμπίεση, αλλά ο A εξακολουθεί να μεταφέρεται πλήρης.

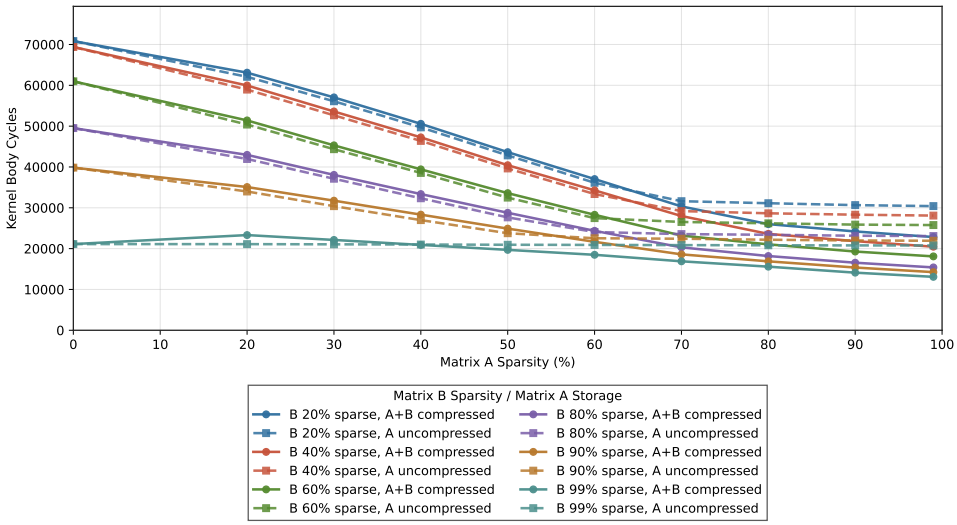


Σχήμα 1.14: DMA συναλλαγές για τους διαφορετικούς τρόπους αραιότητας. Η πλήρης συμπίεση μειώνει τις συναλλαγές όσο αυξάνεται η αραιότητα.

Η ευαισθησία στη διάσταση K είναι επίσης σημαντική. Κάθε πλακίδιο εξόδου πληρώνει σταθερά κόστη αρχικοποίησης, συγχρονισμού και flush. Όσο μεγαλώνει το K, αυτά τα κόστη αποσβένονται σε περισσότερη αριθμητική εργασία. Για τη διαμόρφωση 2×16, η χρησιμοποίηση είναι ήδη 81.2% για $K = 128$ και φτάνει περίπου 93% για $K \geq 512$. Αυτό επιβεβαιώνει ότι η αρχιτεκτονική είναι πιο αποδοτική όταν υπάρχει αρκετή εργασία ανά πλακίδιο εξόδου ώστε να κρύβεται το σταθερό overhead.

Η τελική αξιολόγηση εξετάζει ένα δισδιάστατο sweep αραιότητας για τους δύο τελεστέους. Στον συμπιεσμένο επί συμπιεσμένο τρόπο, η απόδοση βελτιώνεται σημαντικά όσο αυξάνεται η αραιότητα και των δύο πινάκων. Μέχρι περίπου $70\% - 80\%$, η εκτέλεση παραμένει κυρίως compute-bound, οπότε η παράλειψη βημάτων μεταφράζεται σχεδόν άμεσα σε επιτάχυνση. Πέρα από αυτό, η βελτίωση συνεχίζεται αλλά με μικρότερο ρυθμό, επειδή το σταθερό και memory-related κόστος γίνεται κυρίαρχο. Στο $90\%/90\%$, η επιτάχυνση είναι $4.58\times$ σε σχέση με την πυκνή εκτέλεση της ίδιας SGEMM_OP, ενώ στο $99\%/99\%$ φτάνει $5.38\times$.

Ο ασυμπίεστος επί συμπιεσμένο τρόπος ακολουθεί πολύ κοντά τον πλήρως συμπιεσμένο τρόπο για αραιότητα του A έως περίπου $60\% - 70\%$. Σε αυτήν την περιοχή, ο υπολογισμός εξακολουθεί να κυριαρχεί, και το γεγονός ότι ο A μεταφέρεται ασυμπίεστος δεν έχει μεγάλο κόστος. Όταν η αραιότητα του A αυξηθεί περισσότερο, οι δύο τρόποι αποκλίνουν. Ο πλήρως συμπιεσμένος τρόπος συνεχίζει να μειώνει τη μεταφορά δεδομένων, ενώ ο ασυμπίεστος επί συμπιεσμένο τρόπος συνεχίζει να μεταφέρει και τα μηδενικά του A. Παρόλα αυτά, παραμένει



Σχήμα 1.15: Σύγκριση συμπιεσμένου επί συμπιεσμένο και ασυμπίεστου επί συμπιεσμένο τρόπου σε διαφορετικά επίπεδα αραιότητας.

ελκυστικός: στο $90\%/90\%$ φτάνει $3.20\times$ και στο $99\%/99\%$ φτάνει $3.39\times$ σε σχέση με την πυκνή εκτέλεση της SGEMM_OP, αποφεύγοντας το κόστος software compression του δυναμικού A.

Συνολικά, τα αποτελέσματα δείχνουν ότι η TCU-OP επιτυγχάνει μέση επιτάχυνση $12.73\times$ και μέγιστη επιτάχυνση $27\times$ στα μετρούμενα σενάρια σε σχέση με τα αντίστοιχα baselines. Το σημαντικότερο συμπέρασμα δεν είναι μόνο ότι η αραιότητα δίνει επιτάχυνση, αλλά ότι η επιτάχυνση αυτή εμφανίζεται επειδή το σύστημα σχεδιάστηκε ολιστικά: η άμεση πρόσβαση της TCU-OP στη μοιραζόμενη μνήμη μειώνει το register-file staging και τον αριθμό εντολών, το μεγαλύτερο πλακίδιο εξόδου αυξάνει την επαναχρησιμοποίηση των τελεστέων A και B, ο εσωτερικός συσσωρευτής διατηρεί τα μερικά αθροίσματα τοπικά κατά μήκος της διάστασης K, και ο έλεγχος των αραιών βημάτων στο υλικό επιτρέπει την παράλειψη βημάτων εξωτερικού γινομένου που δεν περιέχουν χρήσιμη εργασία..

1.5 Συμπεράσματα και Μελλοντικές Επεκτάσεις

Η εργασία παρουσίασε την TCU-OP, μια αρχιτεκτονική Tensor Core εξωτερικού γινομένου για πυκνή και αραιή εκτέλεση GEMM, υλοποιημένη σε επίπεδο RTL μέσα στο Vortex RISC-V GPGPU. Το βασικό κίνητρο ήταν ότι η υψηλή θεωρητική υπολογιστική ισχύς των Tensor Cores δεν αρκεί όταν η εκτέλεση περιορίζεται από τη μεταφορά δεδομένων, την πίεση στο αρχείο καταχωρητών, το πλήθος εντολών και τη μνήμη. Η υποστήριξη αραιότητας πρέπει να συνοδεύεται από αλλαγή του συνολικού dataflow.

Η προτεινόμενη σχεδίαση αποσυνδέει την εκτέλεση από το αρχείο καταχωρητών, επιτρέποντας άμεση πρόσβαση στη μοιραζόμενη μνήμη και ασύγχρονη εκτέλεση με transactional barriers, αντικαθιστά το baseline datapath εσωτερικού γινομένου με εξωτερικό γινόμενο, εισάγει παραμετροποιήσιμες μονάδες FEOP, εσωτερικό συσσωρευτή 32×32, bitmap-based αποκωδικοποίηση, μονάδες TIA, hardware-controlled παράλειψη αραιών βημάτων, ουρές και crossbar για ακανόνιστη συσσώρευση. Ο βελτιστοποιημένος πυρήνας χρησιμοποιεί DMA και double buffering ώστε να επικαλύπτει μεταφορά, παραγωγή μεταδεδομένων και υπολογισμό.

Η μικροαρχιτεκτονική εξερεύνηση έδειξε ότι η σωστή διαστασιολόγηση είναι απαραίτητη. Επειδή το baseline είναι έντονα υποχρησιμοποιημένο, η TCU-OP μειώνει το μέγεθος του datapath σε 32 μερικά γινόμενα ανά κύκλο, επιλέγει σχήμα 2×16 και ουρές βάθους 4. Αυτή η επιλογή διατηρεί το μεγαλύτερο μέρος της απόδοσης, ενώ περιορίζει το κόστος πολλαπλασιαστών, αθροιστών, τραπεζών και crossbar.

Τα πειραματικά αποτελέσματα επιβεβαιώνουν ότι η σχεδίαση βελτιώνει τόσο την πυκνή όσο και την αραιή εκτέλεση. Η πυκνή λειτουργία ωφελείται στην καλύτερη μεταφορά τελεστέων και τη μεγαλύτερη επαναχρησιμοποίηση. Η αραιή λειτουργία προσθέτει επιπλέον όφελος μέσω συμπίεσης και παράλειψης βημάτων. Η εργασία αναφέρει μέγιστη επιτάχυνση $27\times$, μέση γεωμετρική επιτάχυνση $12.73\times$, μείωση εντολών $29.3\times$, μείωση κίνησης μνήμης $30.2\times$, και αύξηση της χρησιμοποίησης πολλαπλασιαστών από κάτω από 4% σε έως 95.3%.

Η μελλοντική εργασία μπορεί να κινηθεί σε πολλές κατευθύνσεις. Πρώτον, πρέπει να μετρηθεί πλήρως το κόστος προεπεξεργασίας των συμπιεσμένων τελεστέων, ειδικά για δυναμικές ενεργοποιήσεις. Δεύτερον, μπορούν να εξεταστούν εναλλακτικές αραιές αναπαραστάσεις, όπως COO, CSR ή υβριδικά σχήματα με bitmaps. Τρίτον, το δίκτυο συσσώρευσης μπορεί να βελτιωθεί με χαμηλότερου κόστους δρομολόγηση, ιεραρχικά crossbars ή conflict-aware scheduling. Τέταρτον, η TCU-OP πρέπει να αξιολογηθεί σε μεγαλύτερα και πιο ρεαλιστικά workloads, όπως attention, fused MLP layers και convolutions χαμηλωμένα σε GEMM. Τέλος, απαιτείται περαιτέρω φυσικός σχεδιασμός για ακριβέστερη εκτίμηση επιφάνειας, ισχύος, χρονισμού και δυνατοτήτων κλιμάκωσης.

Το γενικό συμπέρασμα είναι ότι η αποδοτική επιτάχυνση αραιών Tensor Cores δεν επιτυγχάνεται απλώς με την παράλειψη μηδενικών πολλαπλασιασμών. Απαιτείται συνολικός επανασχεδιασμός της ροής δεδομένων, της συσσώρευσης, της διεπαφής μνήμης και της συνεργασίας με το λογισμικό. Η TCU-OP δείχνει ότι αυτή η προσέγγιση μπορεί να υλοποιηθεί σε πραγματική RTL GPU αρχιτεκτονική και να προσφέρει σημαντικά οφέλη για πυκνά και αδόμητα αραιά workloads.

Chapter 2

Introduction

2.1 The Rise of Artificial Intelligence in Modern Computing

The era in which we live is largely characterized by rapid technological progress in computer science, algorithm design, and hardware systems. Among these advances, Artificial Intelligence (AI) and Machine Learning (ML) have become dominant forces, transforming a wide range of applications, from natural language processing and computer vision to scientific discovery, healthcare, autonomous systems, and large-scale data analytics. Problems that previously required significant manual effort or prohibitive execution time can now be solved, approximated, or automated at unprecedented scale.

The recent growth of AI can be mainly attributed to three mutually reinforcing factors. First, novel algorithms and model architectures have significantly improved the accuracy and generality of ML systems. Second, the availability of massive datasets has enabled the training of increasingly large and expressive models. Third, continuous improvements in hardware have provided the computational throughput required to train and deploy these models within practical time and energy budgets. Without the corresponding advances in computing platforms, many modern AI models would remain impractical despite their algorithmic potential.

A major milestone in this progress was the introduction of the attention mechanism and the Transformer architecture in 2017 [10]. Since then, AI research has accelerated dramatically, leading to the development of Large Language Models (LLMs) and other foundation models with billions or even trillions of parameters. This trend has created a constant demand for platforms capable of executing AI workloads as fast as possible, while also minimizing power consumption and improving energy efficiency [11]. As model sizes grow, performance alone is no longer sufficient; efficient hardware utilization and reduced data movement become equally important.

Although the term Artificial Intelligence covers a broad range of algorithms and

architectures, many of the most widely used models rely heavily on linear algebra operations. In particular, General Matrix–Matrix Multiplication (GEMM) forms the computational backbone of several important neural network families, including Multilayer Perceptrons (MLPs), Convolutional Neural Networks (CNNs), and LLMs. Matrix multiplication dominates both training and inference time, while also contributing significantly to memory traffic and energy consumption.

- **Multilayer Perceptrons (MLPs):** In an MLP layer, the input activations are multiplied by a weight matrix, followed by the application of a nonlinear activation function. Therefore, the core computation of each fully connected layer is a matrix multiplication between the activation matrix and the weight matrix.

- **Convolutional Neural Networks (CNNs):** Although CNNs are expressed in terms of convolution operations, high-performance implementations often transform convolutions into matrix multiplications using techniques such as `im2col`. This transformation rearranges image patches into columns, allowing the convolution between input feature maps and filters to be executed as a GEMM operation.

- **Large Language Models (LLMs):** Transformer-based LLMs are also dominated by matrix multiplications. In the attention mechanism, the input matrix is first projected into the query, key, and value matrices through three matrix multiplications. The attention scores are then computed as

  $$S = QK^T,$$

  followed by another matrix multiplication between the normalized attention scores and the value matrix,

  $$O = \mathrm{softmax}(S)V.$$

  In addition, the feed-forward or MLP blocks inside each Transformer layer consist of large matrix multiplications, further increasing the importance of GEMM acceleration.

The importance of matrix multiplication is not limited to AI workloads. Hardware improvements have also enabled the development of more complex and accurate High-Performance Computing (HPC) applications, including weather prediction, computational fluid dynamics, molecular dynamics, finite-element analysis, and other large-scale simulations. As computing platforms become more powerful, scientists and engineers can use finer discretizations, larger datasets, and more

detailed models, making simulations more realistic and reliable. This, in turn, further increases the demand for higher hardware throughput and improved energy efficiency.

Across both AI and traditional scientific computing, linear algebra kernels are fundamental building blocks. Among them, matrix multiplication is one of the most performance-critical operations, as it appears either directly or indirectly in a wide variety of algorithms. Consequently, accelerating GEMM has become a central objective in modern computer architecture. Improvements in matrix multiplication throughput can translate directly into faster training, lower inference latency, reduced energy consumption, and more practical execution of large-scale AI and HPC workloads.

2.2 GPU Acceleration, Tensor Cores, and Sparsity

The rapid growth of Artificial Intelligence (AI) and High-Performance Computing (HPC) workloads has created an increasing need for platforms that provide both high throughput and energy efficiency. In many AI models and scientific computing applications, a large fraction of execution time is spent in matrix multiply–accumulate operations, which are highly parallel since different output elements can be computed independently. This makes them well suited to architectures that exploit massive data-level parallelism.

Several platforms have been explored for AI acceleration, including CPUs, GPUs, FPGAs, and ASICs. CPUs offer high programmability but limited parallel throughput, while ASICs, such as Google's Tensor Processing Unit (TPU), can achieve high efficiency using specialized structures such as systolic arrays [12]. However, ASIC specialization reduces flexibility, which is important because AI models evolve rapidly and include operations beyond matrix multiplication, such as activation functions, normalization, softmax, and control flow. GPUs provide a strong trade-off between programmability and performance, combining a flexible software model with massive parallelism and high memory bandwidth. As a result, they have become the dominant platform for AI training and inference in modern datacenters.

To further improve matrix multiplication throughput, modern GPUs integrate specialized matrix engines known as Tensor Cores. Tensor Cores accelerate tiled matrix multiply–accumulate operations using dedicated hardware datapaths, avoiding the overhead of executing the same computation through many scalar or vector instructions. Since their introduction in NVIDIA Volta [1, 2, 13], Tensor Cores have evolved with support for additional datatypes, improved data movement, and structured sparsity [7, 14, 6]. This evolution reflects the need to increase matrix

throughput while reducing operand movement through the register file and memory hierarchy.

Despite the efficiency of dense Tensor Cores, many AI workloads exhibit sparsity, meaning that a significant fraction of matrix elements are zero. Weight sparsity is commonly introduced through pruning [16], while activation sparsity can arise dynamically through functions such as ReLU [17]. Since any multiplication involving a zero produces zero, sparse matrix multiplication can skip ineffective computations and reduce memory traffic.

Existing sparse Tensor Core mechanisms typically exploit structured sparsity, such as NVIDIA's 2:4 pattern, where exactly two elements out of every four are nonzero [7, 6, 18]. This format is hardware-friendly but restrictive: it supports only a fixed sparsity structure and usually exploits sparsity in one operand. Therefore, irregular or higher sparsity cannot be fully utilized unless the model is constrained to the supported format. This motivates more general sparse Tensor Core architectures capable of exploiting unstructured and dual-side sparsity [8, 9].

2.3 Vortex GPU and RTL-Based Evaluation

This work uses the Vortex RISC-V GPGPU as the development and evaluation platform. Vortex is an open-source, synthesizable GPU written in Verilog and developed by Georgia Institute of Technology and UCLA [19]. Its open RTL implementation makes it a suitable platform for architectural research, allowing new GPU mechanisms to be implemented, modified, and evaluated in a realistic hardware design rather than only through high-level models.

Vortex also includes a Tensor Core architecture inspired by NVIDIA-style matrix engines, making it particularly relevant for studying GEMM acceleration. Recent extensions have added support for a 2:4 structured-sparsity mode, providing both a dense Tensor Core baseline and a structured sparse Tensor Core baseline. This enables a direct comparison between the proposed design and existing dense or structured-sparse execution approaches within the same GPU framework.

A key reason for using Vortex is that it can be evaluated at the RTL level. While high-level simulators, such as Accel-Sim, are useful for fast design-space exploration [20], they rely on abstract models, predefined latencies, and simplified assumptions. In contrast, RTL simulation with Verilator executes the actual RTL design and provides cycle-accurate behavior, exposing timing effects, pipeline interactions, and implementation constraints that may be hidden in higher-level models.

Therefore, implementing the proposed Tensor Core modifications directly in RTL provides a more reliable basis for evaluating performance and architectural feasibility. It also allows the design to be further synthesized or deployed on FPGA

platforms, enabling more accurate analysis of implementation cost, area, and power compared to purely C++-based simulation.

2.4 Proposal Overview

To the best of our knowledge, there is currently no RTL implementation of a Tensor Core Unit (TCU) capable of accelerating matrix multiply–accumulate operations where both input matrices exhibit unstructured sparsity. Existing sparse Tensor Core approaches either focus on structured sparsity patterns, such as 2:4 sparsity, or study dual-side unstructured sparsity at a higher level of abstraction, without providing a complete synthesizable hardware implementation.

Motivated by this gap, this thesis builds upon the ideas introduced in DSTC and its extension, "DSTC: Dual-Side Sparse Tensor Core for DNNs Acceleration on Modern GPU Architectures", which target dual-side unstructured sparse GEMM [8, 9]. Inspired by this direction, we develop an extension to the existing Vortex Tensor Core with the goal of improving both dense and sparse matrix multiplication execution. In particular, for sparse MMA, the proposed design aims to skip ineffectual computations when one or both input operands contain unstructured sparsity, while preserving the correct output positions and accumulation semantics.

Unlike DSTC, which was implemented and evaluated in the Accel-Sim simulator using a higher-level C++ model, this work implements the proposed architecture directly in RTL within the Vortex GPU. This enables a more detailed and realistic study of the design, exposing microarchitectural constraints that are often hidden in higher-level simulation frameworks. Such constraints include pipeline timing, varying timing behavior within the Functional Unit, operand routing and control logic complexity. As a result, the RTL implementation provides a stronger basis for evaluating both the feasibility and the performance impact of the proposed sparse Tensor Core architecture. After establishing the baseline RTL design, we further extend and optimize the architecture through design-space exploration.

Furthermore, this work improves the operand delivery mechanism by enabling the Tensor Core to access shared memory directly. This reduces the dependence on register-file staging, alleviates register-file pressure, and lowers the instruction overhead associated with operand movement. Combined with an optimized kernel structure and suitable ISA extensions, the proposed architecture significantly reduces execution time across dense and sparse inputs.

Overall, this thesis presents a complete RTL implementation and evaluation of a sparse Tensor Core architecture for the Vortex RISC-V GPGPU. The proposed design targets dual-side unstructured sparsity, improves operand delivery through direct shared-memory access, and introduces sparse outer-product execution inside the

Tensor Core. Compared to the Vortex baseline, the final optimized design achieves a maximum speedup of $27\times$ and a geometric mean speedup of $12.73\times$ across the evaluated dense and sparse workloads.

2.5 Contributions

To summarize, this thesis makes the following contributions:

- We implement a dual-side sparse Tensor Core mechanism in Verilog and integrate it into Vortex, a state-of-the-art open-source synthesizable RISC-V GPU.
- We perform a detailed microarchitectural exploration of the proposed design, evaluating architectural choices and optimizations with respect to their performance benefits.
- We develop a fully parameterizable Tensor Core design and conduct design-space exploration to identify suitable configurations for different dense and sparse execution scenarios.
- We optimize the sparse mechanism at the RTL level, improving the DSTC design for efficient integration into the Vortex Tensor Core pipeline.
- We further optimize operand delivery by enabling direct Tensor Core access to shared memory, reducing register-file pressure and resembling the data-movement direction adopted by recent NVIDIA Hopper and Blackwell architectures [14, 15].
- We extend the supported execution modes to cover dense and sparse input combinations, including uncompressed $\times$ uncompressed, uncompressed $\times$ compressed, and compressed $\times$ compressed matrix multiplication.
- We extend the ISA to support the proposed Tensor Core functionality and redesign the kernel, resulting in a hardware–software co-designed sparse GEMM execution flow.

2.6 Thesis Structure

The remaining chapters of this thesis are organized as follows:

- Chapter 3 provides the necessary background on GPUs, Tensor Cores, sparsity in GEMM workloads, and the DSTC architecture that motivates this work.

- Chapter 4 presents the implementation details of the proposed sparse Tensor Core design and its integration into the Vortex GPU.
- Chapter 5 discusses the microarchitectural optimizations of the proposed design, including the right-sizing and parameter tuning of the Tensor Core unit.
- Chapter 6 presents the experimental evaluation and analyzes the insights derived from the performance, memory and utilization results.
- Chapter 7 reviews related work and compares the proposed design against existing dense, structured-sparse, and sparse Tensor Core approaches.
- Chapter 8 concludes the thesis and discusses possible directions for future work.

Chapter 3

Background

3.1 Introduction

This chapter provides the background necessary to understand the motivation and design choices of the proposed sparse Tensor Core architecture. It first introduces the main principles of modern GPU architectures and explains why they are well suited for AI and high-performance computing workloads. It then discusses Tensor Cores as specialized matrix multiplication accelerators, followed by an overview of sparsity in deep neural networks and the opportunities it creates for reducing computation and memory traffic. The chapter also presents the Vortex RISC-V GPGPU, which is used as the development and evaluation platform in this thesis, together with its baseline Tensor Core architecture. Finally, it introduces the Dual-Side Sparse Tensor Core (DSTC), which serves as the main inspiration and reference point for the proposed design.

3.2 GPUs as AI Acceleration Platforms

The computational requirements of modern AI workloads have led to the exploration of several acceleration platforms, including CPUs, GPUs, FPGAs, and ASICs. CPUs remain the most flexible general-purpose processors, but their relatively small number of cores and limited arithmetic throughput make them inefficient for the highly parallel kernels that dominate deep learning. FPGAs and ASICs, on the other hand, can provide high performance and energy efficiency by tailoring the hardware to specific computation patterns. A representative example is Google's Tensor Processing Unit (TPU), which uses a systolic-array organization to accelerate tiled matrix multiplication [12]. However, such specialization often reduces programmability and makes it harder to adapt to new algorithms, operators, and model architectures.

The reason massively parallel accelerators are effective for AI is closely related to the structure of matrix multiplication. In a matrix multiply–accumulate operation, each output element is computed independently from the others, making

the operation naturally data-parallel. This property allows thousands of arithmetic operations to be executed concurrently, provided that the hardware can supply operands at sufficient bandwidth. GPUs are designed exactly around this principle: they execute a large number of lightweight threads in parallel and rely on high memory bandwidth and latency hiding to maintain throughput.

Compared to ASICs, GPUs offer a more balanced compromise between performance and flexibility. Although matrix multiplication is usually the dominant operation, complete AI models also include non-GEMM operations such as activation functions, normalization, softmax, indexing, reductions, memory transformations, and sometimes control-flow-dependent execution. A fully specialized accelerator may perform GEMM very efficiently, but supporting the full workload with sufficient flexibility can be more challenging. GPUs, through programming models such as CUDA, HIP, and OpenCL, allow developers to express auxiliary model operations within the same platform [24, 25].

This combination of high parallel throughput, programmability, and mature software support has made GPUs the standard choice for large-scale AI training and inference. Modern AI datacenters are therefore commonly built around large GPU clusters, where thousands of devices cooperate to train and serve increasingly large models. As a result, GPU vendors such as NVIDIA and AMD have placed AI acceleration at the center of their architectural roadmaps, introducing specialized matrix engines, high-bandwidth memory systems, and optimized software libraries. Consequently, understanding and improving GPU-based matrix acceleration is essential for increasing the performance and energy efficiency of future AI systems.

3.3 Fundamentals of GPUs

As GPU architectures evolved from fixed-function graphics pipelines into massively parallel general-purpose processors, several programming abstractions and architectural components emerged to expose and manage this parallelism efficiently. These concepts form the basis of modern GPU programming and are used throughout the rest of this thesis. A brief overview of the most relevant terms is given below.

GPGPU (General-Purpose GPU): A GPGPU refers to the use of a graphics processing unit for general-purpose computation beyond graphics rendering. It exploits massive parallelism through thousands of lightweight threads executing concurrently. This execution model is particularly effective for data-parallel workloads such as linear algebra, scientific simulations, and machine learning.

Kernel: A kernel is a function that runs on the GPU and is executed by many threads in parallel. Each thread executes the same kernel code but operates on different portions of the input data. Kernels are launched by the host CPU and define the main computation performed by a GPU program.

Thread: A thread is the smallest unit of execution in a GPU program. Each thread has its own registers and executes the kernel independently, enabling fine-grained parallelism. Threads are organized into warps and thread blocks to support efficient scheduling, synchronization, and data sharing.

Warp: A warp is a group of threads that execute instructions in lockstep under the SIMT execution model. In NVIDIA architectures, a warp typically consists of 32 threads. Threads within a warp share a common instruction stream, although control-flow divergence can be handled through masking. Warps are the fundamental scheduling and execution units inside a Streaming Multiprocessor.

SM (Streaming Multiprocessor): A Streaming Multiprocessor is the main processing unit of a GPU and is responsible for executing warps. It contains execution units such as ALUs and Tensor Cores, a register file, shared memory, and control logic such as warp schedulers. Multiple SMs operate in parallel to provide the total computational throughput of the GPU.

Shared Memory (SMEM): Shared memory is a programmer-managed, low-latency on-chip memory shared among threads within the same thread block. It is commonly used to stage data and enable efficient reuse, reducing accesses to slower global memory. Proper use of shared memory is essential for optimizing memory-bound GPU kernels.

Thread Block: A thread block is a group of threads that execute the same kernel and can cooperate during execution. Threads within a block can synchronize using barriers and share data through shared memory, enabling efficient coordination and data reuse. A thread block is scheduled onto a single SM and remains there for its entire lifetime, ensuring fast communication among its threads. Multiple thread blocks can execute independently across different SMs, providing scalability across the GPU.

Register File (RF): The register file is a fast on-chip storage structure that holds operands and intermediate results for GPU threads. It provides high bandwidth and low latency, making it critical for sustaining compute throughput. Registers are private to each thread and are heavily used during kernel execution.

Core Pipeline: The core pipeline is the sequence of hardware stages that process instructions, including scheduling, fetching, decoding, issuing, execution, and commit. In a GPU, this pipeline is warp-scheduled and follows the SIMT execution model, interleaving instructions from different warps across cycles in order to hide latency and keep the execution units active.

Async Transaction Barriers: Asynchronous transaction barriers are synchronization mechanisms used to coordinate access to shared resources without fully blocking execution. They are often used together with asynchronous operations, such as memory transfers, to overlap computation and communication. This helps improve hardware utilization by avoiding unnecessary pipeline stalls.

DMA / TMA: DMA (Direct Memory Access) refers to hardware engines that transfer data between memory regions without direct involvement of the main execution units. TMA (Tensor Memory Accelerator) is a specialized form of such a mechanism in modern GPUs, designed to efficiently move multi-dimensional tensor data between global memory and shared memory. These engines enable high-throughput asynchronous data movement and allow memory transfers to overlap with computation.

Micro-operations (u-ops): Micro-operations, or u-ops, are internal low-level operations into which higher-level instructions may be decomposed during decoding. This allows complex instructions to be executed as a sequence of simpler hardware operations that can be scheduled, issued, and executed efficiently by the core pipeline.

Transactional / asynchronous Barriers: Transactional barriers are synchronization primitives used to coordinate groups of threads with asynchronous hardware mechanisms. They allow software to wait until a set of operations has been completed consistently before execution proceeds.

3.4 Tensor Cores in GPUs

The continuous demand for higher GPU throughput has exposed important inefficiencies in the execution of matrix multiply–accumulate (MMA) operations using only conventional scalar or vector datapaths. Matrix multiplication has significant data reuse, since each element of an input tile contributes to multiple output values. However, when MMA is implemented using standard ALU or FPU instructions, this reuse is not always exploited efficiently. Matrix fragments must typically be loaded

into the register file, repeatedly read by the execution units, multiplied and accumulated through several instructions, and finally written back. This creates substantial instruction overhead and register-file traffic, especially for tiled GEMM kernels where the same operands are reused many times.

To address this bottleneck, NVIDIA introduced Tensor Cores in the Volta architecture [1, 2, 13]. A Tensor Core is a specialized execution unit inside the GPU designed to perform small tiled matrix multiply–accumulate operations directly in hardware. Instead of decomposing matrix multiplication into many scalar multiply and add instructions, Tensor Cores execute an entire MMA operation as a single matrix instruction at the warp level. This significantly increases arithmetic throughput and improves energy efficiency, especially for deep learning workloads that can tolerate reduced-precision input formats with higher-precision accumulation [3].

Most Tensor Core designs are optimized around an inner-product formulation of matrix multiplication. In this model, each output element is computed as the dot product between a row of matrix (A) and a column of matrix (B). This organization maps well to hardware, since multiple products can be computed in parallel and accumulated through reduction trees or deeply pipelined fused multiply–accumulate datapaths [3, 4]. To sustain high utilization, operand tiles are staged close to the compute units, typically through a combination of shared memory and register-file fragments. Shared memory is commonly used to stage reusable matrix tiles from global memory, while the register file provides the immediate operands consumed by MMA instructions [25, 26].

Since their introduction, Tensor Cores have evolved significantly across GPU generations. Volta introduced FP16 Tensor Cores with FP32 accumulation, enabling large speedups for mixed-precision training [1, 2]. Turing extended Tensor Core support to additional low-precision formats, such as INT8 and INT4, making the units more suitable for inference workloads. Ampere introduced TF32, allowing many FP32 workloads to benefit from Tensor Core acceleration with minimal software changes, and also added support for structured sparsity, where sparse Tensor Cores can provide up to $2\times$ higher throughput when the required sparsity pattern is satisfied [7].

Hopper further expanded the role of Tensor Cores by adding FP8 support and the Transformer Engine, targeting the training and inference of large Transformer-based models. It also introduced new mechanisms for improving data movement, such as the Tensor Memory Accelerator (TMA), and expanded MMA execution beyond the traditional warp-level model through warp-group MMA instructions [14, 27, 28]. These features reflect a broader trend toward decoupling data movement from computation and increasing the granularity of tensor operations to improve utilization.

More recent NVIDIA architectures, such as Blackwell, continue this trend by increasing Tensor Core throughput and supporting even lower-precision formats, including FP4 variants for AI inference [15]. This progression shows that Tensor Cores have evolved from fixed-function mixed-precision matrix units into sophisticated tensor-processing pipelines tightly coupled with the GPU memory hierarchy.

Overall, Tensor Cores are now central to GPU performance for AI and HPC workloads. Their efficiency, however, depends not only on raw arithmetic throughput, but also on how effectively operands are delivered, reused, and accumulated. As Tensor Core compute capability increases, data movement through the register file and memory hierarchy becomes an increasingly important bottleneck. This motivates architectural techniques that reduce operand traffic, improve reuse, and exploit additional workload properties such as sparsity.

3.5 Outer-Product Matrix Multiplication

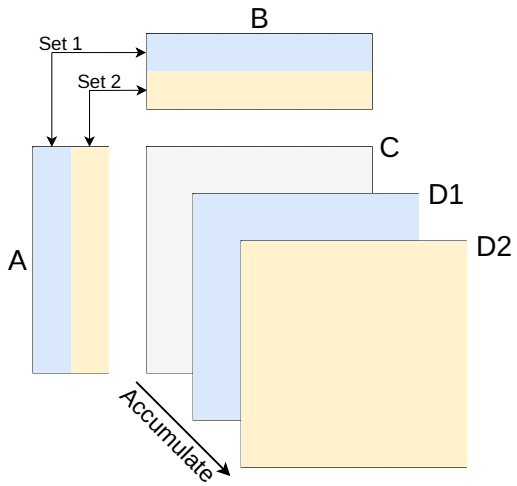


FIGURE 3.1: Outer-product matrix multiplication. Each step multiplies one column of A with one row of B, producing a rank-1 partial output tile that is accumulated into the output matrix.

As discussed previously, conventional Tensor Core architectures are typically designed around the inner-product formulation of matrix multiplication. In this model, each output element is computed independently as the dot product between one row of matrix A and one column of matrix B. This organization is well suited for dense matrix multiplication, since each output accumulator can remain local to the datapath until the corresponding dot product is completed.

An alternative formulation is the outer-product approach. In outer-product matrix multiplication, computation proceeds along the reduction dimension K. A basic

step in this method computes the cross-product between one column of matrix A and the corresponding row of matrix B. If the output tile has dimensions $M \times N$, this operation can be expressed as

$$\langle M, N \rangle = \langle M, 1 \rangle \times \langle 1, N \rangle,$$

where a column vector of A with M elements is multiplied by a row vector of B with N elements, producing a rank-1 partial output matrix. This partial matrix is then accumulated into the output tile:

$$D = C + \sum_{k=0}^{K-1} A_{:,k} B_{k,:}.$$

Thus, instead of computing one output element at a time, each step contributes to many elements of the output tile.

Figure 3.1 illustrates this process. The initial output tile C, often corresponding to a bias or previously accumulated value, is updated through a sequence of partial products. Each partial product is generated from a column vector of A and a row vector of B, and the resulting rank-1 matrix is accumulated into the output tile D. After all K-dimension steps have been processed, the final output tile is produced.

The main advantage of the outer-product formulation is its high input reuse. Each loaded element from the selected column of A is reused across an entire row of the partial output tile, while each loaded element from the selected row of B is reused across an entire column. This makes the approach attractive when input delivery is a bottleneck. However, this comes at the cost of more challenging output accumulation. Since the output tile is updated incrementally over many steps, partial sums must either be kept in an internal accumulation buffer or repeatedly read and written, increasing pressure on the accumulation storage.

This behavior contrasts with the inner-product formulation commonly used in Tensor Cores. Inner-product execution favors output reuse: each accumulator is updated locally until its dot product is complete, reducing the need to repeatedly access the output tile. However, the input operands A and B are typically reused over a smaller scope. In summary, inner-product matrix multiplication favors local accumulation and output reuse, while outer-product matrix multiplication favors input reuse and as will be explained exposes more opportunities for skipping computation when sparse input operands are present.

3.6 Sparsity in GEMM Workloads

A common property of many GEMM workloads is *sparsity*, i.e., the presence of zero-valued elements in one or both input matrices. Sparsity is particularly important because any scalar multiplication involving a zero operand produces a zero result. Therefore, if the hardware can identify such cases early, the corresponding multiplication can be skipped, reducing both computation and energy consumption.

In AI workloads, sparsity mainly arises from two sources. The first is *weight sparsity*, which is commonly introduced through pruning [16]. During pruning, weights considered less important for the model output are removed or set to zero, reducing the number of parameters and potentially improving generalization by limiting overfitting. The second is *activation sparsity*, which appears dynamically during execution. A typical example is the Rectified Linear Unit (ReLU), one of the most widely used activation functions in deep neural networks [17, 29, 30, 31]. ReLU preserves positive inputs and maps negative inputs to zero, thereby introducing zeros in the activation matrices that are forwarded to subsequent layers. Sparsity is also common in HPC workloads, where it often originates from the structure of the underlying problem. For example, graph algorithms frequently operate on adjacency matrices, which are usually highly sparse because most vertices are connected to only a small subset of the graph.

Sparsity can be broadly classified as either *structured* or *unstructured*. In unstructured sparsity, zeros may appear at arbitrary positions, without a regular pattern. This form is the most general and can preserve accuracy, since pruning is not constrained by a predefined structure. However, it is also harder to exploit efficiently in hardware because the positions of the nonzero elements are irregular. Structured sparsity, in contrast, imposes a pattern on the locations of zeros and nonzeros, making the sparse representation easier to decode and execute.

Several structured sparsity patterns are commonly used:

- **Row-wise sparsity:** Entire rows of a matrix are zero, while the remaining rows are dense. If the zero rows follow a regular pattern, the sparsity is structured; otherwise, it is row-wise but irregular.

- **Block-wise sparsity:** The matrix is divided into fixed-size blocks, and each block is either entirely zero or contains nonzero values. This representation reduces metadata overhead and is attractive when zeros appear in coarse-grained regions.

- ***M* : *N* sparsity:** In $M : N$ sparsity, each group of N consecutive elements contains at most M nonzero values. NVIDIA's 2:4 sparsity is a well-known example, where two out of every four elements are nonzero [7, 6]. Such formats are hardware-friendly because they provide a bounded and predictable

number of nonzeros per group, but they require metadata to indicate the positions of the surviving elements.

Structured sparsity simplifies hardware design, but it also limits the amount and type of sparsity that can be exploited. For example, NVIDIA's 2:4 sparsity can exploit exactly 50% sparsity in the compressed operand. If the matrix contains more zeros, the additional sparsity cannot directly translate into further speedup. Conversely, if the matrix does not naturally satisfy the required pattern, additional pruning or retraining may be needed to enforce it. This can reduce model accuracy, since some weights may be forced to zero even if they are not negligible. As a result, structured sparsity offers an efficient but restricted trade-off between hardware simplicity, performance, and accuracy.

Most existing sparse Tensor Core approaches exploit either dense $\times$ structured-sparse multiplication or, less commonly, structured-sparse $\times$ structured-sparse multiplication [18, 6]. This limits both the achievable speedup and the range of sparsity patterns that can be used. Ideally, a sparse Tensor Core should be able to exploit unstructured sparsity in both operands, skipping all ineffectual multiplications while keeping area and energy overheads low [8, 9].

To execute sparse matrices, the compressed values must be accompanied by metadata that describes their original positions. Several sparse encodings are commonly used:

- **COO (Coordinate format):** Each nonzero value is stored together with its row and column indices. This format is simple and flexible, but it requires significant metadata storage and multiple metadata accesses per nonzero element.
- **CSR (Compressed Sparse Row):** CSR stores nonzero values row by row, together with column indices and a row-pointer array that indicates where each row begins. It is widely used for very sparse matrices and provides compact storage when the number of nonzeros is extremely small compared to the dense matrix size. However, accessing the position of a nonzero element requires dependent metadata accesses, which can limit parallelism.
- **Bitmap encoding:** Bitmap encoding uses one bit per matrix element to indicate whether the corresponding value is zero or nonzero. A bit value of zero denotes a zero element, while a bit value of one denotes that the element is present in the compressed value array. This format is simple to decode and requires only one metadata stream.

In this work, bitmap encoding is selected for the sparse matrix representation. Among the available sparse formats, bitmap and CSR are two of the most commonly used alternatives. CSR can provide lower metadata overhead for extremely

sparse matrices, since it stores information only for nonzero elements. However, recovering the original position of a value in CSR requires accessing two separate metadata structures: the row-pointer array and the column-index array. This introduces additional memory requests and load-to-load dependencies, which can limit parallelism and complicate hardware decoding.

Bitmap encoding, in contrast, requires only one metadata stream. Each bit directly indicates whether the corresponding dense matrix position contains a nonzero value, making the format simple to decode and well suited for parallel hardware processing. Although the bitmap size is fixed with respect to the dense tile size, this overhead is acceptable for the sparsity levels commonly targeted in sparse neural networks and several HPC workloads. CSR becomes more attractive only at extremely high sparsity levels, where the number of nonzero values is very small. For the sparsity range considered in this thesis, typically up to approximately 95%–99%, bitmap encoding provides a practical balance between metadata storage, decoding simplicity, reduced memory dependencies, and hardware parallelism.

3.7 Vortex: An RTL-Level RISC-V GPGPU

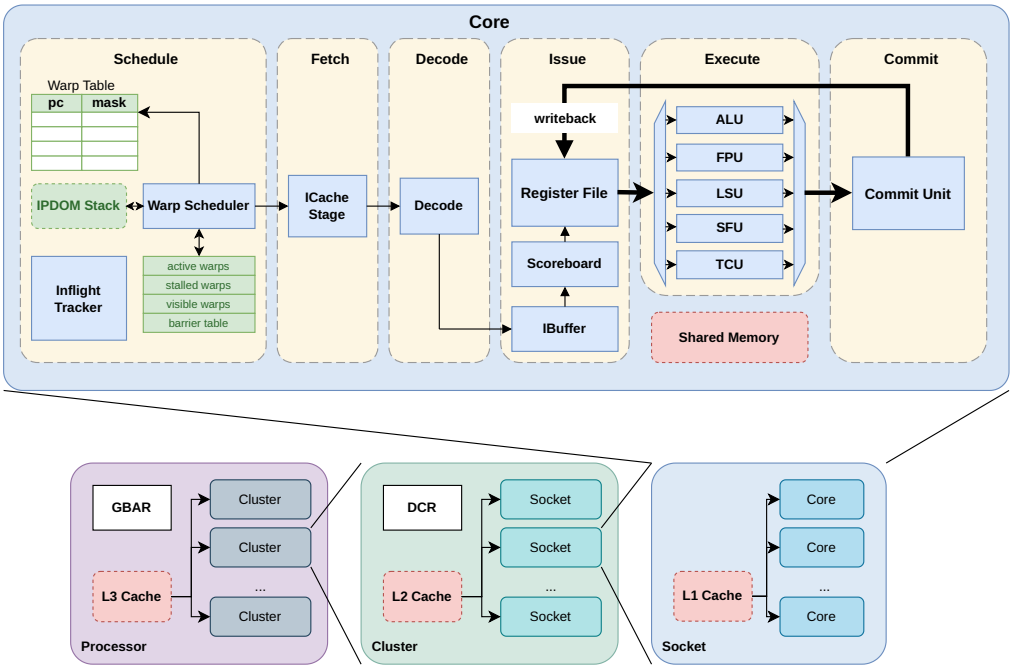


FIGURE 3.2: High-level Vortex micro-architecture

Vortex is an open-source, research-oriented GPGPU architecture based on the RISC-V ISA [19]. It is fully implemented in synthesizable Verilog, enabling both

cycle-level RTL simulation and FPGA-based prototyping. This makes Vortex particularly suitable for architectural research, since new mechanisms can be implemented directly in hardware description language and evaluated under realistic pipeline, memory, and resource constraints.

Figure 3.2 shows the high-level organization of the Vortex architecture. At the processor level, Vortex is organized hierarchically into clusters, sockets, and cores, connected through a cache hierarchy that includes private and shared cache levels. Each core follows a SIMT execution model, where warps are scheduled and executed through a pipelined datapath consisting of scheduling, fetch, decode, issue, execute, and commit stages. The execute stage includes multiple functional units, such as the ALU, FPU, Load–Store Unit (LSU), and Special Function Unit (SFU), while the issue stage manages register-file accesses, scoreboarding, and instruction buffering.

Vortex also provides several architectural features that make it suitable for studying modern GPU workloads. Each core includes a register file for per-thread operands and intermediate values, as well as shared memory (mentioned as local memory in the terminology adopted by Vortex) that can be used for low-latency data staging and inter-thread cooperation. In addition, Vortex includes a general-purpose DMA mechanism for moving data without relying solely on scalar load/store instructions. Such asynchronous transfers are coordinated through transactional barriers, which allow software to detect when a DMA transaction has completed and safely continue execution. These mechanisms are particularly important for tiled GEMM kernels, where overlapping data movement and computation is essential for high performance.

The modular design of Vortex also allows custom accelerators to be integrated as additional execution units inside the core pipeline. In recent versions, Vortex has been extended with a Tensor Core unit for accelerating tiled matrix multiply–accumulate operations. This baseline Tensor Core resembles early NVIDIA-style Tensor Core designs, where operands are staged through the register file before execution. Moreover, structured sparse execution support, inspired by NVIDIA's 2:4 sparsity mechanism, has also been incorporated, providing both dense and structured sparse Tensor Core modes.

For these reasons, Vortex provides an appropriate platform for this thesis. It combines an open RTL implementation, a GPU-like SIMT execution model, shared-memory and DMA support, transactional synchronization mechanisms, and an extensible Tensor Core architecture. This enables the proposed sparse Tensor Core modifications to be developed and evaluated within a realistic GPU microarchitecture rather than in isolation.

3.8 Vortex Tensor Core Unit (TCU)

3.8.1 Operation of the Vortex TCU

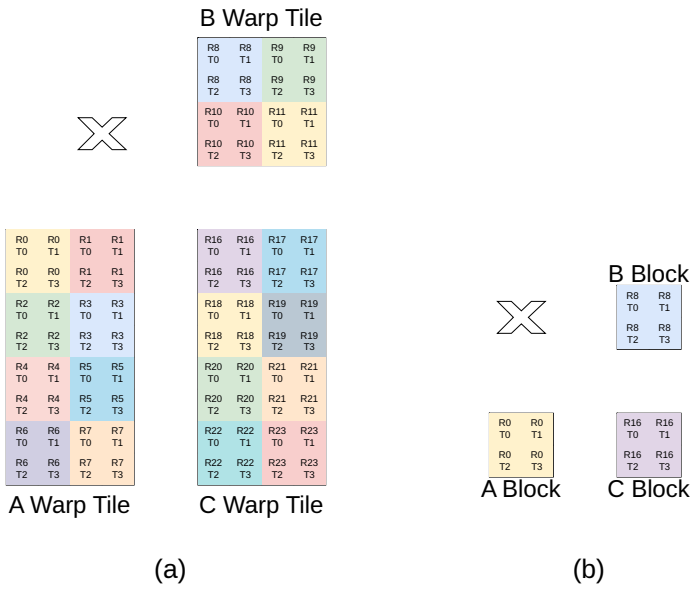


FIGURE 3.3: Register-level fragmentation of Vortex Tensor Core warp tiles for a 4-thread configuration. Matrix tiles are distributed across thread registers, and registers with the same index form the blocks consumed by individual TCU micro-operations.

As part of the effort to keep Vortex a competitive and extensible open-source GPU research platform, a Tensor Core Unit (TCU) has been integrated into the architecture and is continuously optimized. The Vortex TCU is designed to accelerate tiled matrix multiply–accumulate operations and follows a programming and execution model similar to modern NVIDIA Tensor Cores. In particular, the current implementation resembles the warp-level execution model of recent NVIDIA architectures, where all threads of a warp cooperate to execute a single matrix operation.

The baseline Vortex TCU follows a register-file-based dataflow. Matrix fragments are first loaded from memory into the registers of the participating threads. The TCU then consumes these register-resident fragments, performs the matrix multiply–accumulate operation, and writes the resulting fragments back to the register file. Therefore, the general data path is

$$\text{MEM} \rightarrow \text{RF} \rightarrow \text{TCU} \rightarrow \text{RF} \rightarrow \text{MEM}$$

where (MEM) may refer either to global memory or shared memory. Recent Vortex optimizations also include a DMA-like mechanism, called DXA, which enables asynchronous data movement from global memory to shared memory. This reduces

data-fetch latency and allows tiled kernels to stage operands more efficiently before they are loaded into registers.

The Vortex TCU supports several numerical formats used in machine learning workloads, including FP32, FP16, BF16, TF32, FP8, BF8, NVFP4, and INT8. This datatype flexibility allows the same unit to target different execution scenarios, ranging from high-accuracy training to low-precision inference. To expose Tensor Core functionality to software, Vortex extends the baseline RISC-V ISA with custom matrix instructions. The three main instructions are `load_matrix_sync`, `mma_sync`, and `store_matrix_sync`.

`load_matrix_sync` The `load_matrix_sync` instruction synchronously loads matrix tiles from memory into the register file. In the RISC-V architecture, each thread has 32 general-purpose registers. In the Vortex Tensor Core programming model, a subset of these registers is reserved for matrix fragments: eight registers for matrix A, eight registers for matrix B, and eight registers for matrix C. Consequently, a warp collectively stores the fragments of the input and output tiles across the registers of its participating threads.

This instruction is responsible for mapping the elements of a warp-level matrix tile to the correct registers of each thread. Figure 3.3 illustrates this fragmentation for a 4-thread warp configuration. Each cell corresponds to a register entry and contains both the register index and the thread identifier that owns it. Cells with the same color correspond to registers with the same index across different threads and form a block. The layout is not arbitrary: it is chosen so that later `mma_sync` instructions can select matching blocks of A, B, and C using register indices. It is also worth noting that, for some warp-size configurations, such as 4-thread and 16-thread warps, the B tile does not fully utilize all eight allocated registers, which affects the block scheduling of the Tensor Core operation.

`mma_sync` The `mma_sync` instruction invokes the Tensor Core using matrix fragments that have already been loaded into the register file. Conceptually, the operation performed is

$$C = A \times B + C$$

However, a single RISC-V instruction can reference only a limited number of registers: source registers for the operands and a destination register for the result. Therefore, each issued `mma_sync` instruction can operate only on one selected register block from each matrix fragment. Since each thread holds multiple registers per matrix, the full warp-level matrix multiplication cannot be completed with a single hardware invocation. Instead, it must be decomposed into multiple smaller block operations.

For example, an instruction of the form

```
mma_sync R16, R0, R8, R16
```

uses registers (R0), (R8), and (R16) as the selected fragments of A, B, and C, respectively, and writes the updated result back to (R16). With the register layout shown in Figure 3.3, this corresponds to a block-level operation on submatrices of the full warp tile. For a 4-thread warp configuration, the complete output tile requires 16 such block multiplications. Therefore, the high-level `mma_sync` instruction is decomposed into multiple internal micro-operations.

To support this behavior, Vortex includes a sequencer at the head of the instruction buffer. When an `mma_sync` instruction reaches the sequencer, it is expanded into a sequence of micro-operations, each targeting different register indices. Each micro-operation invokes the TCU on a specific block of the original warp-level tile and performs the block computation:

$$C_{\text{block}} = A_{\text{block}} \times B_{\text{block}} + C_{\text{block}}$$

The number of generated micro-operations depends on the warp configuration and tile mapping. For example, a 4-thread warp requires 16 micro-operations, while a 32-thread warp requires 32 micro-operations. Only after all micro-operations of the current `mma_sync` have been issued can the scoreboard allow the next instruction from the same warp to proceed.

For each micro-operation, the TCU performs the block computation using its internal multipliers, which are organized into Fused-Element Dot-Product (FEDP) units. Each FEDP unit computes one element of the output block by forming the dot product between one row of the selected A block and one column of the selected B block. For instance, in the 4-thread warp configuration, the TCU contains four FEDP units, matching the four elements of the output block. Each FEDP consumes the corresponding operands from two registers of A and two registers of B, performs the required multiplications and reduction, and produces one accumulated output element.

`store_matrix_sync` The `store_matrix_sync` instruction performs the inverse operation of `load_matrix_sync`. It collects the output fragments stored in the thread registers and writes the reconstructed C tile back to memory. This completes the register-file-based Tensor Core execution flow.

Overall, the baseline Vortex TCU provides a realistic open-source implementation of a modern Tensor Core-style matrix engine. However, its reliance on register-file staging means that all operands must pass through the register file before reaching the TCU, and all results must return to the register file before being stored. This data movement becomes a significant bottleneck for high-throughput GEMM execution and motivates the direct shared-memory access and sparse execution mechanisms developed in this thesis.

3.8.2 Limitations of the Vortex TCU

Although highly optimized for matrix-multiply-accumulate operations, the Vortex Tensor Core Unit introduces several important limitations. First, the inner-product MMA formulation inherently provides limited data reuse for the input operands A and B. Since the computation is organized around relatively small fragments, each operand element is reused fewer times compared to architectures that expose larger output tiles or alternative dataflows. This limitation is further amplified by the fact that the output tile, namely the C tile, is constrained by the number of registers that a warp can dedicate to storing partial results. Even in 32-thread configurations, where the output tile reaches 16×16, the achievable reuse remains limited, reducing the overall efficiency of the unit.

A second limitation stems from the way data are supplied to the TCU. As described above, matrix fragments are first loaded into the register file before they are consumed by the Tensor Core. This design simplifies the interface between the warp execution pipeline and the TCU, but it also increases instruction overhead, register-file pressure, power consumption, and latency. In practice, the warps that invoke Tensor Core operations must allocate a large portion of their registers to matrix fragments. As a result, any additional temporary values required by the surrounding code increase the risk of register spilling, which can further degrade performance.

Moreover, each `mma_op` instruction is internally decomposed into multiple micro-operations. Depending on the configuration, a single instruction may generate 16 or 32 micro-ops, increasing traffic through the execution pipeline. This not only raises power consumption, but also introduces additional cycle overhead, limiting the benefit of the specialized hardware when the surrounding pipeline cannot supply or retire operations efficiently.

Another important constraint is that the three newly introduced Tensor Core instructions are synchronous. Once a warp invokes one of these instructions, it cannot proceed with independent work while the operation is being carried out. This prevents useful overlap between Tensor Core execution and other instructions

within the same warp, reducing the opportunities for latency hiding and making performance more dependent on warp-level scheduling.

The load and store instructions also introduce non-negligible overhead. In order to map matrix tiles to per-thread register fragments, complex indexing operations must be performed. In many cases, this indexing overhead can become comparable to, or even larger than, the cost of the actual data movement. Consequently, even when matrix tiles are preloaded into shared memory, the observed speedup may remain limited, because each thread still has to compute the precise location of the data that should be loaded into each register. This behavior contributes to the low utilization of the TCU observed in further analysis.

Finally, the recent introduction of 2:4 structured sparsity support in the Vortex TCU, closely resembling NVIDIA's sparsity mechanism, introduces additional constraints. As discussed earlier, this form of sparsity can only exploit a fixed sparsity pattern and therefore cannot take advantage of arbitrary sparse matrices. Furthermore, enforcing the required 2:4 structure may lead to accuracy degradation, depending on the model and pruning strategy. Thus, while sparse Tensor Core execution can theoretically improve throughput, its practical benefits are limited by both structural restrictions and possible numerical-quality trade-offs.

3.9 Overview of DSTC: Dual-Side Sparse Tensor Core

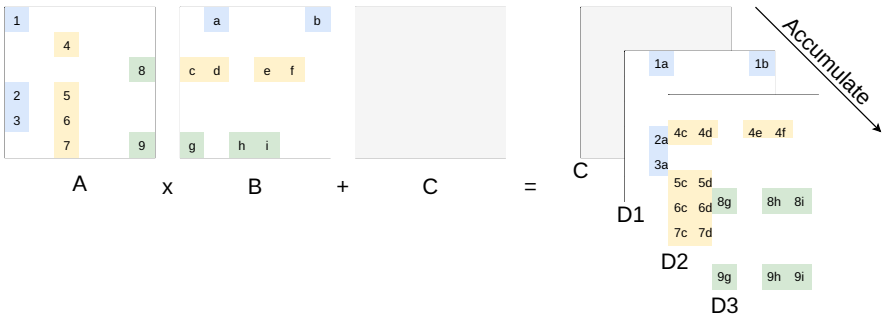


FIGURE 3.4: Bitmap-based outer-product sparse GEMM. Nonzero elements from a column of A and a row of B are multiplied to produce sparse partial output matrices, which are accumulated into the output tile.

The sparse Tensor Core mechanism developed in this thesis is inspired by the Dual-Side Sparse Tensor Core (DSTC) architecture and its extended work, *DSTC: Dual-Side Sparse Tensor Core for DNNs Acceleration on Modern GPU Architectures* [9]. DSTC targets the acceleration of sparse GEMM and convolution by exploiting sparsity in both input operands, instead of only considering sparsity in the weight matrix. This is an important distinction from conventional sparse Tensor Core designs,

which typically assume one sparse operand and one dense operand. In DNN workloads, however, both weights and activations may contain a significant number of zeros, making dual-side sparsity an important opportunity for further acceleration.

The main idea of DSTC is to combine an outer-product computation primitive with bitmap-based sparse encoding. Conventional Tensor Cores, including NVIDIA Tensor Cores and the baseline Vortex TCU, are generally designed around the inner-product formulation of matrix multiplication. In that case, each output element is computed as the dot product between one row of matrix A and one column of matrix B. While this approach is efficient for dense GEMM, it becomes more challenging for dual-side sparse execution, because the hardware must dynamically match nonzero elements from two sparse vectors before performing the multiplications. This matching operation, often referred to as sparse inner-join, introduces irregularity and additional hardware overhead.

DSTC avoids this problem by adopting an outer-product formulation. At each step, the Tensor Core multiplies one column vector of matrix A with the corresponding row vector of matrix B. The result is a partial $(M \times N)$ matrix that must be accumulated with the current output tile. Repeating this process across the K dimension produces the final output. As shown in Figure 3.4, the initial C tile is updated by a sequence of partial matrices $(D_1, D_2, D_3, \ldots)$, each produced from one column of A and one row of B.

The FEDP units used in dot-product Tensor Core architectures are replaced by Fused-Element Outer-Product (FEOP) units, following the outer-product computation model. Each FEOP unit receives one element from A and n elements from B, producing $(1 \times n)$ partial products in a single step. Consequently, an $(m \times n)$ output region can be generated by operating m FEOP units in parallel, each corresponding to one selected element of A.

This formulation is particularly suitable for unstructured sparsity. If the selected column of A and row of B contain zeros, the nonzero elements can first be compacted into shorter dense vectors. The Tensor Core can then perform a dense outer-product over these compacted vectors, instead of executing multiplications involving zeros. In other words, the challenge is moved from dynamically matching nonzero pairs, as in sparse inner-product execution, to computing the cross-product of the remaining nonzero elements and recovering their original output positions. Since every nonzero element in the selected column of A must be multiplied with every nonzero element in the corresponding row of B, the multiplication itself becomes regular after compaction.

DSTC represents each sparse input tile using a bitmap and a value array. The bitmap stores the original sparsity pattern, where a bit value of one denotes a

nonzero element and a bit value of zero denotes a zero element. The value array stores only the nonzero elements. To match the outer-product dataflow, values of matrix A are arranged in column-major order, while values of matrix B are arranged in row-major order. This layout allows the hardware to fetch the compacted column and row vectors required for each outer-product step.

The DSTC algorithm can be described through three main operations. First, the *multiply-value* operation computes the outer-product between the compacted nonzero values of A and B. Second, the input bitmaps are processed to derive the original coordinates of the generated partial sums. This step is necessary because, after compaction, the values are dense in the compressed representation but still correspond to irregular positions in the original output tile. Third, the *merge* operation uses the generated addresses to accumulate the partial sums into the correct entries of the output tile.

The main trade-off of the outer-product approach is related to output accumulation. Outer-product execution increases the reuse of input operands, since each nonzero element of A is reused across multiple elements of B, and vice versa. However, the output tile is updated incrementally across many outer-product steps. Therefore, partial sums must be stored and repeatedly updated until all K-dimension contributions have been processed. To support this efficiently, DSTC introduces an internal accumulation buffer inside the Tensor Core. This buffer holds the partial output tile locally, avoiding frequent accesses to the register file or shared memory during accumulation.

In the design considered in this thesis, the accumulation buffer has the same logical dimensions as the output tile, namely (32×32). Each entry of the buffer corresponds to one element of the output tile being computed. Since a complete outer-product step over a (32×32) output tile would require $32 \times 32 (= 1024)$ scalar multiplications, the computation cannot be performed in a single cycle with a realistic number of multipliers. Instead, processing is divided into smaller hardware steps. At each step, a subset of the compacted column of A and a subset of the compacted row of B are fed into the outer-product datapath, producing a smaller block of partial sums. The Tensor Core sweeps across the output tile in both dimensions until the full outer-product contribution has been generated and accumulated.

The performance opportunity of DSTC comes from compacting the nonzero elements of the selected column of A toward the top and the nonzero elements of the selected row of B toward the left, as illustrated in Figure 3.5(a). After this compaction, the useful values are packed into shorter dense vectors, while the zero elements are effectively removed from the computation stream. As a result, several outer-product steps may correspond entirely to zero operands and can therefore be skipped, as shown in Figure 3.5(b). Skipping these zero-only steps reduces not

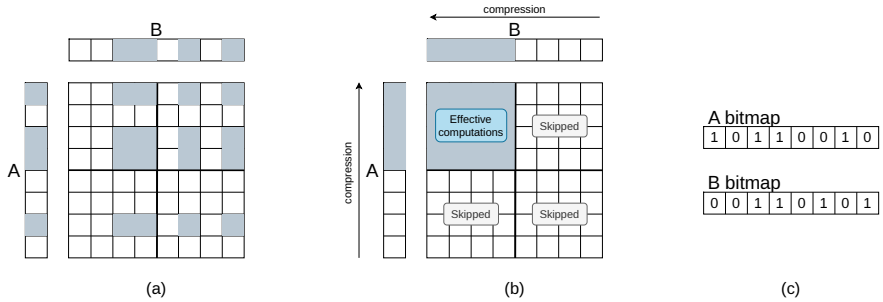


FIGURE 3.5: Sparse outer-product execution in DSTC. (a) Outer-product multiplication when both input operands are sparse. (b) Compaction of the nonzero elements of the A column and B row enables zero-only computation steps to be skipped. (c) Bitmap metadata records the original positions of the nonzero elements in each input vector.

only the number of arithmetic operations, but also the amount of data that must be transported through the Tensor Core datapath.

It is worth noting that, in the sparse representation, matrix tiles are compressed in memory before being consumed by the TCU, rather than being compressed online inside the unit. This preprocessing introduces an additional computational overhead, since the nonzero values must be compacted and the corresponding bitmap metadata must be generated. However, once this representation is available, the executing warps transfer fewer values, reducing both memory traffic and register-file pressure.

To support this mechanism, DSTC stores each sparse operand in compressed form. Only the nonzero values are kept in the value array, while a bitmap is used as metadata to record their original positions in the uncompressed tile. This extraction of zeros reduces the memory footprint of the workload and correspondingly lowers the number of memory transactions required to fetch the operands. As shown in Figure 3.5(c), the bitmap uses one bit per matrix element: a bit value of one denotes that the corresponding element is nonzero and therefore present in the compressed value array, while a bit value of zero denotes a zero element that has been omitted. This bitmap-based representation is particularly attractive for tile-level sparsity because it has relatively low metadata overhead and allows the hardware to reconstruct the original coordinates of the compressed operands efficiently.

Based on this representation, the original DSTC mechanism supports two execution modes. In *dense mode*, both operands are treated as uncompressed. No bitmap loading or sparse preprocessing is required, and all outer-product steps are executed in the conventional manner. In *sparse mode*, both operands are stored in compressed form, and the Tensor Core uses the corresponding bitmaps to determine the true addresses. In this way, the execution can skip zero-only steps and operate

only on the compacted nonzero vectors, while the bitmap metadata preserves the information needed for correct placement and accumulation of the partial results.

In the DSTC design, the decision to skip sparse outer-product steps is exposed to the software instruction stream rather than being handled entirely by an internal hardware controller. A warp-level sparse matrix operation is lowered into a sequence of outer-product Tensor Core instructions, referred to as OHMMA instructions, which act as outer-product counterparts to conventional NVIDIA HMMA instructions. Each OHMMA instruction triggers the execution of a step on the TCU. Sparse acceleration is achieved by conditionally suppressing OHMMA instructions whose corresponding compacted operand regions contain no useful work. This is implemented through predication: the bitmaps of A_v and B_v are analyzed using population-count operations, and the resulting predicate values determine which OHMMA instructions are issued to the Tensor Core and which are bypassed. Consequently, the execution schedule is partly managed by software, requiring repeated instruction-level coordination to drive the sparse computation.

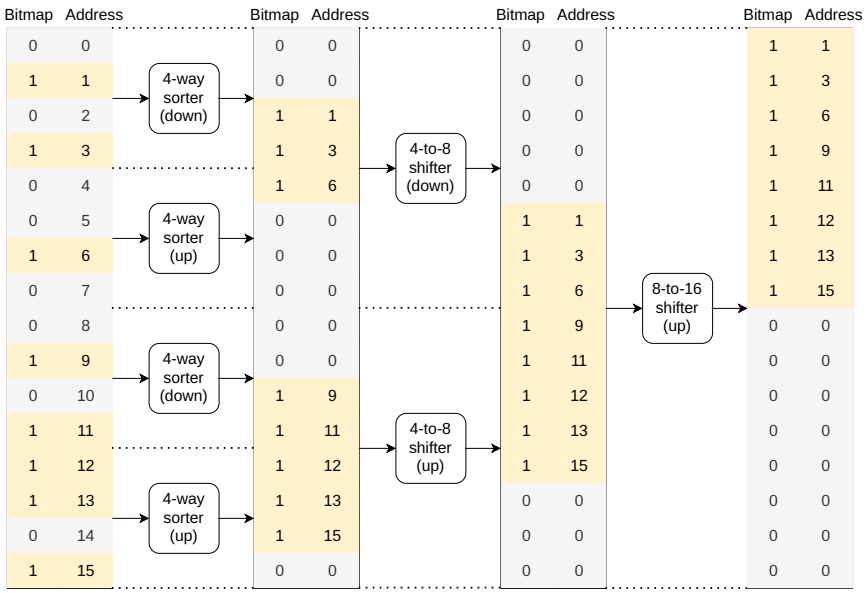


FIGURE 3.6: Simplified Tile Index Aligner (TIA) example. The TIA extracts the addresses of nonzero elements from bitmap metadata. At each stage, a shifter handles twice as many elements as the previous stage; therefore, for 32-bit inputs, one additional layer is required beyond the ones shown.

After multiplication, the compressed partial sums must be accumulated at their original locations in the accumulation buffer. This requires translating the compressed output indices back to their coordinates in the dense output tile. To perform this translation, the design uses the bitmap metadata together with the Tile Index Aligner (TIA), a simplified example of which is shown in Figure 3.6.

The TIA is a hardware unit inside the TCU that generates aligned indices for sparse accumulation by exploiting the shift-invariant structure of the outer-product sparsity pattern. It receives a 32-bit bitmap and processes it through a hierarchical sorting network. The bitmap is first divided into eight 4-bit groups, each handled by a 4-way sorter. Each sorter extracts the set bits and preserves their original addresses. The following stages progressively merge larger groups, with each stage processing twice as many bits as the previous one, until the final ordered list of nonzero addresses is produced. Figure 3.6 illustrates the same process for a 16-bit bitmap for simplicity.

Once the original addresses have been extracted, they are forwarded to the accumulation buffer together with the corresponding partial products, allowing the output tile to be updated at the correct locations. The accumulation process follows a gather-accumulate-scatter pattern. First, the current values corresponding to the generated output addresses are gathered from the accumulation buffer. Then, the newly generated partial sums are added to these values. Finally, the updated results are scattered back to their original positions. Although the accumulation buffer contains 32×32 entries, at most N new partial sums can be produced per cycle. Therefore, the buffer is organized into N banks, allowing multiple updates to proceed in parallel.

To support the irregular access pattern caused by unstructured sparsity, a crossbar is placed before the accumulation-buffer banks. The crossbar receives the multiplication outputs, i.e., the newly generated partial sums, and routes each one to the appropriate bank according to the address generated by the TIA. This allows sparse partial results to be accumulated in their correct dense output positions while still enabling parallel updates across the banked accumulation buffer.

However, sparse execution introduces irregular output addresses, which can lead to bank conflicts when multiple partial sums target the same accumulation-buffer bank in the same cycle, as illustrated in Figure 3.7. These conflicts stall the TCU pipeline and reduce the effective throughput of the sparse datapath. To mitigate this issue, DSTC introduces queues before the crossbar. These queues absorb short-term imbalances in the generated addresses and smooth the access pattern before requests are routed to the accumulation-buffer banks.

In this original design, the adders are placed before the crossbar, with one adder associated with each queue. Each adder combines the generated partial sum with the previously stored accumulator value, and the updated result is then forwarded through the crossbar to be written back to the appropriate accumulation-buffer bank. Once all elements have been processed and the queues have been drained, the warp flushes the contents of the accumulation buffer into the register file and commits, completing a full synchronous TCU operation.

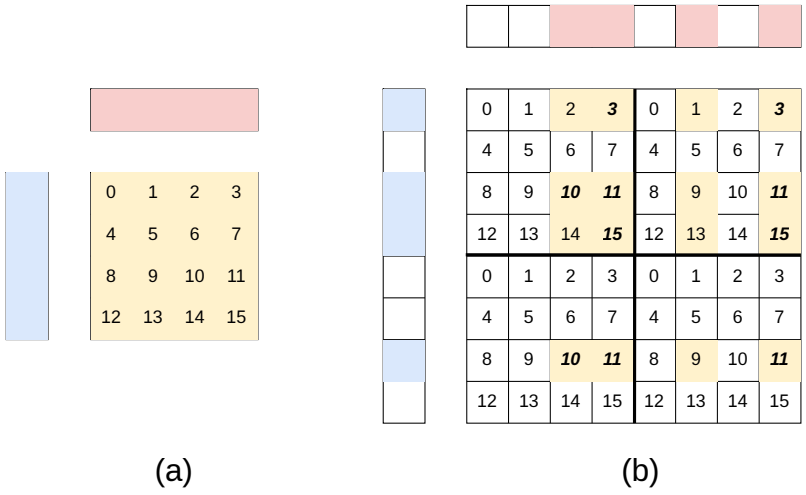


FIGURE 3.7: Partial sums produced from a single 4×4 step. (a) In dense execution, outputs are naturally distributed across different banks. (b) In sparse execution, irregular output addresses may cause multiple partial sums to target the same bank, creating bank conflicts.

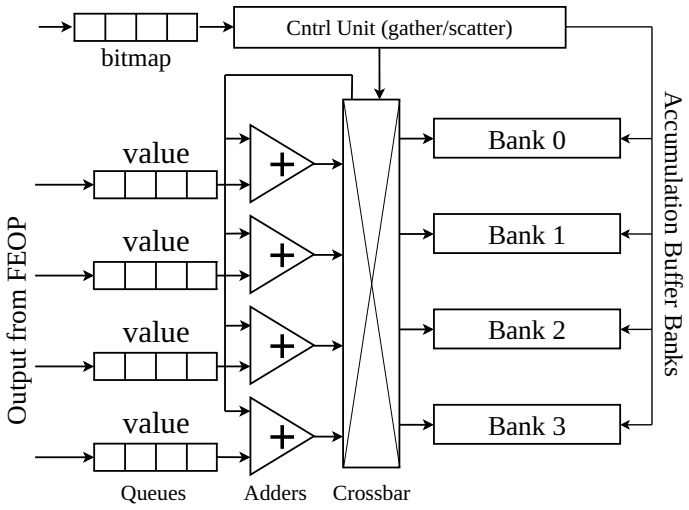


FIGURE 3.8: Accumulation buffer organization. Queues absorb temporary imbalances in sparse output addresses, queue-local adders update the corresponding accumulation-buffer entries and the crossbar routes the results to the correct banks

The results reported in the DSTC work indicate that this organization provides significant speedup opportunities for sparse execution while incurring relatively low hardware overhead.

Chapter 4

Implementation Details

4.1 Introduction

This chapter presents the implementation and architectural details of the proposed Outer-Product Tensor Core Unit (TCU-OP). The design introduces two main hardware modifications for supporting sparse execution. First, it replaces the conventional inner-product datapath with an outer-product organization, implemented through the FEOP unit described in Section 4.2. Second, it introduces an accumulation buffer, described in Section 4.3, to store and update the partial output tile generated by the outer-product computation.

The implementation is not limited to the internal arithmetic datapath of the Tensor Core. To make the new unit useful inside the Vortex GPU, the design also modifies how Tensor Core operations are issued, tracked, synchronized, and supplied with data.

Figure 4.1 summarizes this integration at the Vortex system level. The TCU-OP is attached to the core execution path, but it also communicates directly with shared memory and the asynchronous barrier infrastructure.

Section 4.4 presents the TCU control unit, which orchestrates the execution flow and coordinates the interaction between the FEOP units and the accumulation buffer. This unit extends both the DSTC proposal and the baseline Vortex TCU by enabling direct memory accesses from the Tensor Core. Section 4.5 then discusses the software kernel implementation used to expose the hardware capabilities and maximize performance. Finally, Section 4.6 describes the ISA extensions required to support the new unit, together with the supported input and output data types.

4.2 FEOP Unit Design

The proposed TCU-OP replaces the Fused-Element Dot-Product (FEDP) units of the baseline inner-product architecture with Fused-Element Outer-Product (FEOP) units. This change reflects the transition from dot-product execution, where each

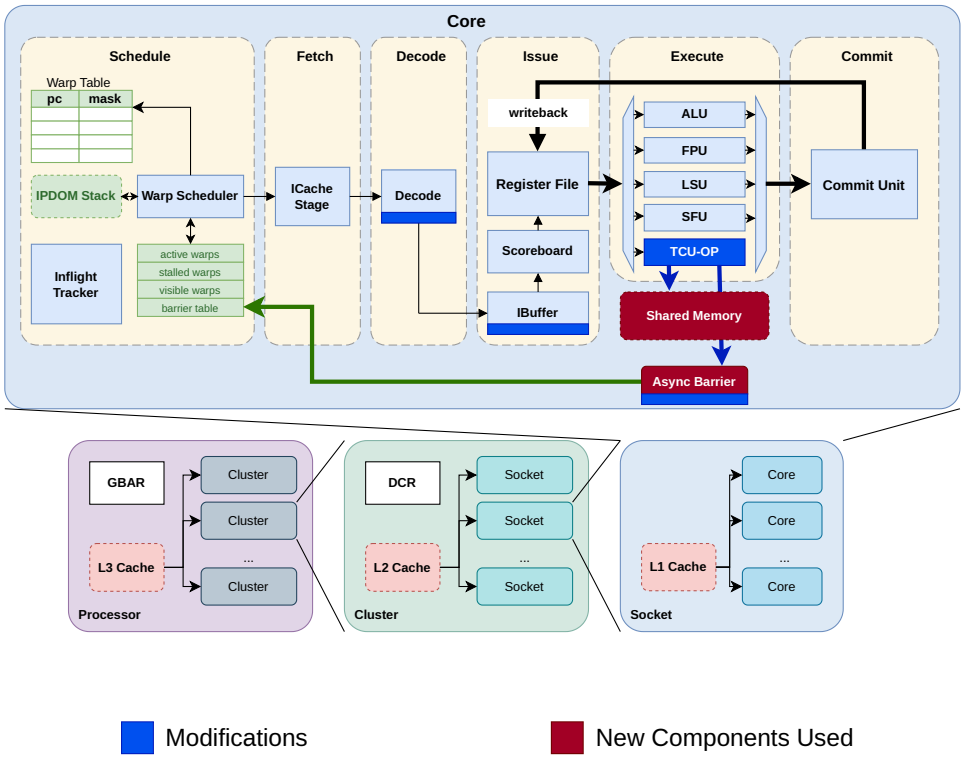


FIGURE 4.1: High-level integration of the proposed TCU-OP inside the Vortex GPU. The highlighted components show the modified core paths and the supporting memory and synchronization mechanisms used by the proposed design.

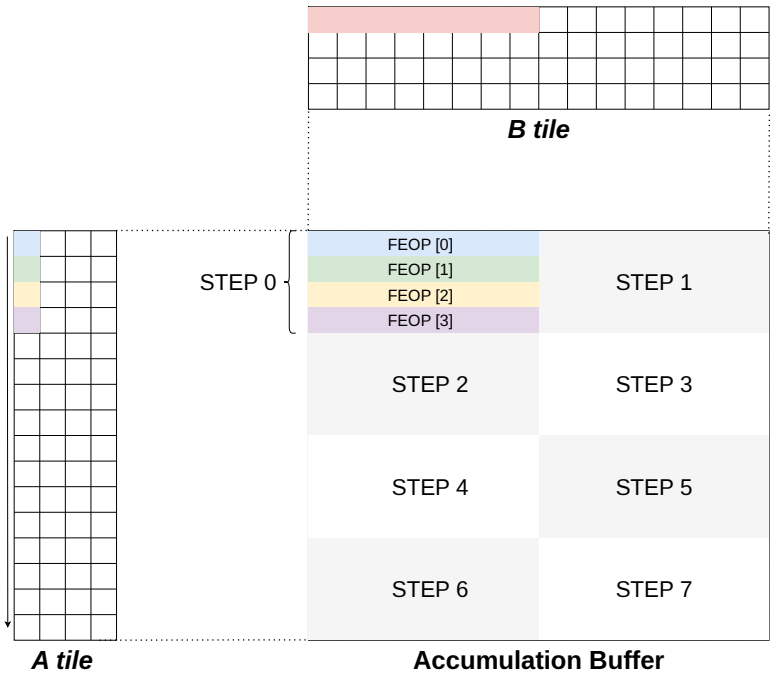


FIGURE 4.2: Example of FEOP-based outer-product step execution. The accumulation buffer has dimensions 16×16, while each step computes a 4×8 region. Therefore, each set requires eight steps in total.

unit produces one reduced output element, to outer-product execution, where each unit generates a vector of partial products.

The computation is organized into sets and steps. A set corresponds to one selected column of A and one selected row of B, whose outer product contributes to the output tile. Since the full outer product of these vectors is larger than the number of multipliers available in the hardware, each set is decomposed into multiple steps. Each step computes a fixed-size sub-region of the output tile and the complete set is produced by iterating over all required steps.

For a step of size $(m \times n)$, the FEOP datapath generates an $(m \times n)$ grid of products:

$$D[i][j] = A[i] \times B[j].$$

These values are not final output elements. Instead, they are partial products that must be accumulated with the previous contents of the corresponding positions in the output tile.

The FEOP datapath is fully parameterizable through the $\mathtt{m} \times \mathtt{n}$ dimensions, which define the size of each outer-product step. In the example of Figure 4.2, $\mathtt{m}$ is set to 4 and $\mathtt{n}$ to 8, meaning that four FEOP units operate in parallel, each producing 8 results. Each FEOP receives one element from the selected A block and multiplies it with $\mathtt{n}$ elements from the selected B block, producing one row of the

step output. Therefore, the FEOP units together generate a $\mathtt{m} \times \mathtt{n}$ region of the accumulation buffer in one step. The remaining regions of the output tile are covered by subsequent steps. In this example, the accumulation buffer has dimensions 16×16. Therefore, each set is computed over two horizontal steps and four vertical, resulting in a total of eight steps per set.

The choice of m and n directly affects both throughput and sparsity granularity. Larger step dimensions increase the amount of work performed per cycle, but reduce the opportunity to skip smaller sparse regions. For example, increasing m means that each vertical step covers more elements of the selected A column, making the exploitable sparsity granularity for A coarser. Similarly, increasing n makes the sparsity granularity for B coarser. If the total number of multipliers is fixed, increasing one dimension typically requires decreasing the other, creating a trade-off between the sparsity granularity of the two operands. The parameterized FEOP design makes this trade-off explicit and enables exploration of different step shapes.

Compared with the DSTC organization [9], our FEOP design removes the adders shown inside the outer-product units. Since the FEOP does not compute a reduction, these adders are unnecessary in the multiplication datapath. The FEOP always produces pure multiplication results, while all accumulation is handled by the accumulation buffer. This separation also clarifies the handling of the C operand: the accumulation buffer is initialized with the C tile before partial products are accumulated, so C does not need to be injected into the FEOP datapath.

Finally, the FEOP supports multiple numerical formats, including integer and floating-point operands at 8, 16, and 32 bits. Additional formats can be supported by modifying the multiplier path while preserving the same outer-product structure. Overall, the FEOP provides a simpler and more modular replacement for the FEDP datapath, better matching the requirements of sparse outer-product execution.

4.3 TCU Accumulation Buffer Design

The accumulation buffer stores and updates the output tile produced by the outer-product datapath. Unlike an inner-product Tensor Core, where each unit produces a reduced output element, the FEOP units generate partial products that must be accumulated across multiple sets. Therefore, the output tile remains locally resident inside the TCU until all outer-product contributions have been processed.

The buffer is organized as an $(M \times N)$ storage structure, matching the dimensions of the output tile. In each cycle, the TCU core may produce up to $(m \times n)$ partial products, where m and n define the dimensions of one FEOP step. Each incoming value is accompanied by an output-tile address and a valid bit. The address

selects the accumulator entry to be updated, while the valid bit indicates whether the corresponding value participates in the accumulation.

In dense execution, the access pattern is regular. Each FEOP step updates a fixed sub-region of the accumulation buffer, and the complete output tile is covered after

$$\text{STEPS} = \frac{M \times N}{m \times n}$$

steps. The dense address mapping is chosen so that all $(m \times n)$ values produced in the same cycle target different banks, allowing conflict-free parallel accumulation.

To support this parallelism, the accumulation buffer is implemented as a banked storage structure with $(m \times n)$ banks. Each bank stores a subset of the output-tile entries. In the absence of conflicts, all FEOP outputs generated in one step can therefore be accumulated in parallel.

Each update follows a gather-accumulate-scatter flow. First, the current value is read from the target bank. Then, it is added to the generated partial product. Finally, the updated value is written back to the same accumulator entry. Since each FEOP output updates an independent position of the output tile, no reduction tree is required; instead, the design uses one accumulator adder per bank-local update path.

Sparse execution, however, makes the access pattern irregular. Because the nonzero elements of the selected A column and B row are compacted before multiplication, FEOP outputs no longer map to consecutive positions in the dense output tile. Their true locations are recovered from bitmap metadata through the Tile Index Aligner. As a result, multiple valid partial products generated in the same cycle may target the same accumulation-buffer bank. In this case, only one request can be served immediately, while the remaining requests must wait.

To route partial products to their correct banks, a crossbar is placed between the FEOP output queues and the accumulation-buffer banks, as shown in Figure 4.3. Each queue stores a generated partial product together with its output address. The crossbar uses the target bank implied by the address to forward each request to the appropriate bank-local accumulation path.

The queues tolerate short-term bank conflicts. Without them, any conflict at the crossbar would immediately stall the FEOP datapath, since the TCU could not accept a new batch of partial products. With queues, temporary conflicts are absorbed locally, allowing computation to continue as long as queue space remains available. The TCU stalls only when back-pressure propagates because one or more queues become full. Thus, queue depth creates a direct trade-off between performance and hardware cost: deeper queues hide more irregularity, but increase area and storage overhead.

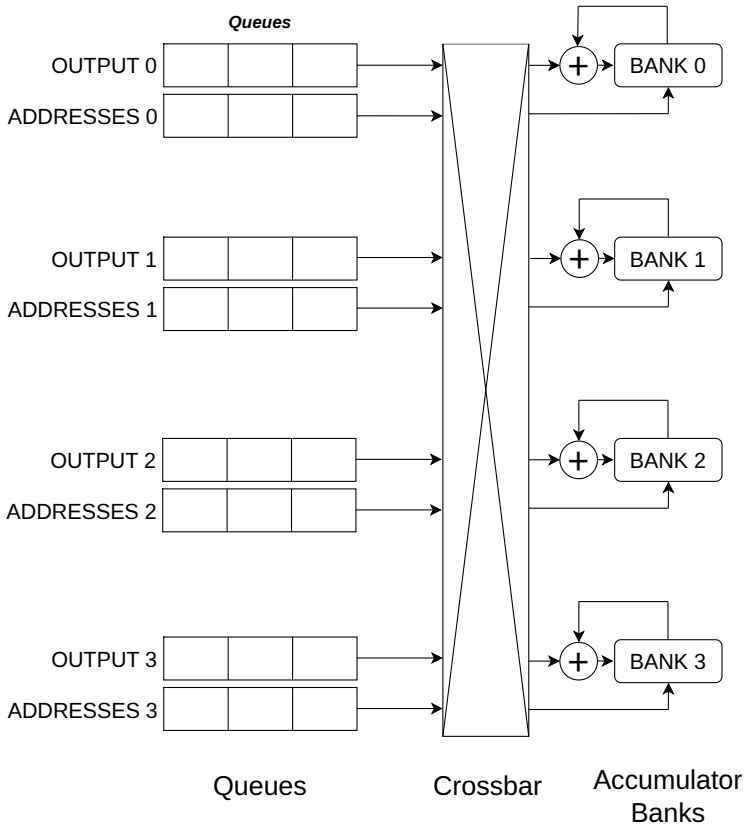


FIGURE 4.3: Organization of the TCU accumulation buffer. FEOP outputs and their corresponding addresses are first stored in queues. The crossbar routes each valid partial product to the appropriate accumulator bank, where a bank-local adder updates the stored value.

A key difference from the DSTC-style accumulator organization is the placement of the adders relative to the crossbar. In our design, the crossbar is placed before the adders, so each adder is associated with a specific accumulation-buffer bank rather than with a specific FEOP output (figure 4.3). Thus, each incoming partial product is routed once to the correct bank-local accumulation path. After the addition, the updated value is written back directly to the same bank without passing through the crossbar again. This reduces crossbar traffic and simplifies the write-back path.

The crossbar remains one of the main hardware costs of the accumulation buffer. While high-level models often treat routing as an abstract operation, the RTL implementation exposes its hardware complexity. Its area, delay, and wiring complexity increase rapidly with the number of inputs and outputs. Consequently, the choice of $(m \times n)$, which determines the number of crossbar inputs, is an important microarchitectural parameter. Larger steps increase instantaneous compute throughput, but also require more crossbar connections as well as multipliers, adders, queues and banks. For this reason, the accumulation-buffer organization must be tuned together with the FEOP step dimensions.

In this work, the accumulation buffer stores a (32×32) output tile with 32-bit accumulation entries, resulting in a total capacity of $(4\ \mathrm{KB})$. This tile size matches the original DSTC design since it improves input operand reuse while keeping the buffer size practical. Given that the baseline Vortex TCU is substantially memory-bound, the selected FEOP configuration incorporates 32 multipliers, producing 32 partial products per cycle, 4 tiles less than the baseline Vortex TCU. This is a necessary trade-off where we have saved area and power consumption while the throughput displays a negligible change. Therefore, a complete dense set requires

$$\frac{32 \times 32}{32} = 32$$

steps. This choice reduces the amount of compute hardware compared to the baseline 32-thread Vortex TCU, while preserving high performance for the memory-bound GEMM workloads.

Figure 4.4 illustrates step skipping for Set 1. In this example, the selected row of B is partitioned into four horizontal regions, so sparsity in B can be exploited at a granularity of 25%. Similarly, the selected column of A is partitioned into eight vertical regions, allowing sparsity in A to be exploited at a granularity of 12.5%. After compaction, only the highlighted region contains useful nonzero work for this set; therefore, the corresponding steps are executed, while the remaining zero-only steps are skipped.

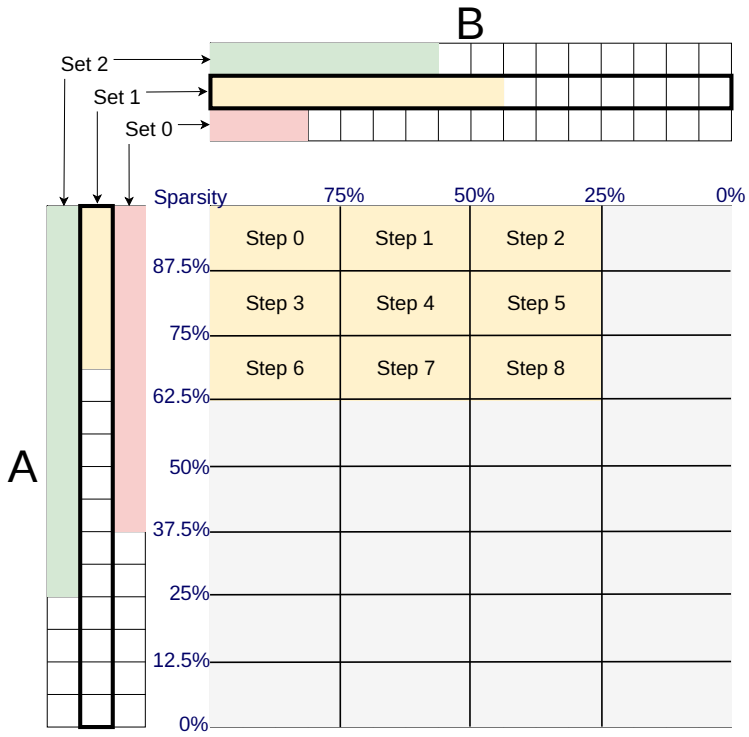


FIGURE 4.4: Example of step skipping for Set 1. The selected row of B is divided into four horizontal regions, corresponding to 25% sparsity granularity, while the selected column of A is divided into eight vertical regions, corresponding to 12.5% sparsity granularity. Only the highlighted steps contain useful nonzero work and must be executed.

4.4 TCU Core Design

TCU Core Control and Operand Dispatch

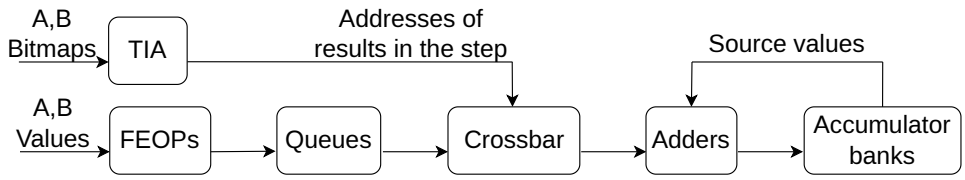


FIGURE 4.5: High-level computation pipeline of TCU-OP

The TCU core is the central control unit of the proposed outer-product Tensor Core. It coordinates the execution of the FEOP datapath, manages the movement of operands, and forwards the generated partial products to the accumulation buffer. Its main role is to receive tiles of A, B, C and $bitmaps$, select the operands required for the current outer-product step, dispatch them to the FEOP units, and provide the resulting partial products with their corresponding output addresses. Figure 4.5

illustrates the high-level computation pipeline of TCU-OP, from bitmap decoding and FEOP multiplication to crossbar routing and bank-local accumulation.

Before the outer-product computation begins, the TCU initializes the accumulation buffer by loading the output tile C that corresponds to the current computation. This initialization both overwrites any residual values from a previous TCU invocation and establishes the bias values onto which the FEOP partial products will be accumulated. Once all entries of C have been loaded into the accumulator, the TCU proceeds to process the A and B tiles. After all partial products along the K dimension have been accumulated, the TCU enters a flushing phase, in which the final output tile is drained from the accumulator and written back to memory.

To control the computation, the TCU core maintains internal counters for the current set and step. A set corresponds to one column of A and one row of B, while a step corresponds to a sub-region of the outer product produced by that set. These counters determine which operands are selected from the input buffers and which region of the output tile is updated at each cycle. The core also handles datapath backpressure. If the accumulation buffer cannot accept new partial products because its queues are full, or if the required input operands have not yet arrived, the TCU core stalls execution until the pipeline can safely proceed.

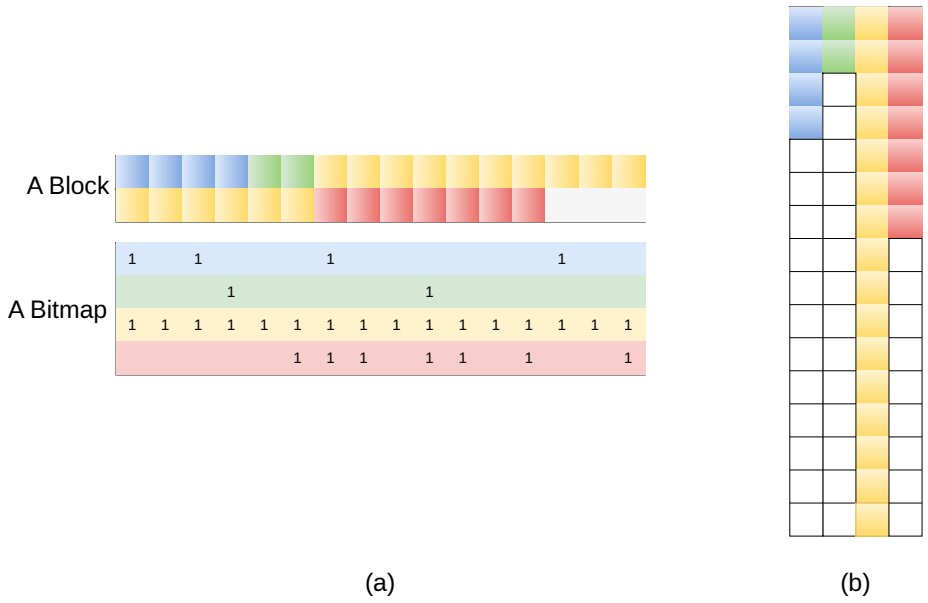


FIGURE 4.6: Compressed representation of an A block and its interpretation inside the TCU-OP. (a) Nonzero values are stored contiguously in memory, accompanied by bitmap metadata that records their original positions. (b) The TCU-OP uses the bitmap to recover the logical column positions of the compressed values before dispatching them to the FEOP units.

For sparse execution, the TCU core supports the compressed sparse representation introduced by DSTC [9], where both input operands are stored as compacted

nonzero values together with bitmap metadata. The bitmap records the original positions of the nonzero elements, allowing the hardware to recover their dense coordinates after compression. Figure 4.6(a) shows an example of how the compressed values of an A block and the corresponding bitmap are stored in memory. The nonzero values are packed contiguously, while the bitmap preserves the original locations of these values inside the dense tile. Figure 4.6(b) shows how the TCU-OP interprets the same data when feeding the FEOP units: the compacted values are expanded logically into their original column positions.

The core includes two Tile Index Aligner (TIA) modules, one for operand A and one for operand B, following the principle introduced in DSTC. Each TIA processes bitmap metadata and produces the dense positions of the compacted nonzero elements. The TIA associated with A generates row indices for the produced partial products, while the TIA associated with B generates the corresponding column indices. These indices are forwarded to the accumulation buffer together with the FEOP partial products, allowing each value to update the correct location in the output tile.

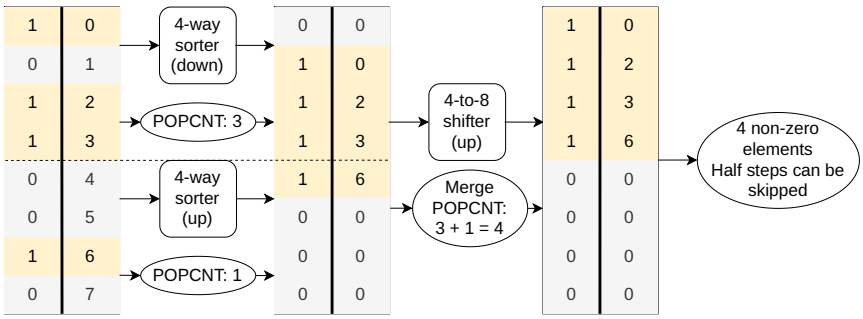


FIGURE 4.7: Operation of the Tile Index Aligner (TIA). In the simplified 8-bit example shown here, two sorter stages are sufficient; for the actual 32-bit inputs used in the design, two additional shifter stages are required.

In our design, the TIA is also used to derive the number of effective computation steps that must be executed. This allows step selection to be controlled inside the hardware, instead of relying on continuous software-level instruction predication as in the original DSTC design. As illustrated in Figure 4.7, each 4-way sorter identifies the set bits within a 4-element bitmap segment and preserves their original positions. The following shifter stages merge the compacted outputs of previous stages, while the corresponding population counts are accumulated. The final output therefore provides both the ordered nonzero positions and the total number of nonzero elements.

Using the number of nonzero elements detected in the selected A column and B row, the TCU core determines which outer-product steps contain useful work

and which can be skipped. This is the main source of sparse compute acceleration: after compaction, steps that contain only zero operands are not issued to the FEOP units. Since the sparsity pattern is unstructured, the number of valid operands may vary across sets and steps. The core therefore updates the A- and B-side offsets dynamically, selecting the next compacted operands that must be processed. Once the FEOP units generate their partial products, the TCU core forwards them to the accumulation buffer together with the dense output addresses produced by the TIA modules.

Additional Uncompressed $\times$ Compressed sparse execution mode

In addition to the uncompressed $\times$ uncompressed (dense $\times$ dense) and compressed $\times$ compressed (sparse $\times$ sparse) modes, the proposed design introduces a third execution mode: uncompressed $\times$ compressed. This mode targets the common DNN scenario in which B represents static weights that can be compressed offline, while A represents activations that may exhibit sparsity but are costly to preprocess in software due to their dynamic nature. In this mode, B is fetched from memory in compressed form alongside its bitmap metadata, while A is supplied uncompressed and compressed on the fly inside the TCU. This eliminates software preprocessing overhead for activations while still enabling the hardware to exploit runtime sparsity in A.

Overall, the TCU supports three execution modes. In uncompressed $\times$ uncompressed mode, both operands are treated as dense and no sparse preprocessing is required. In uncompressed $\times$ compressed mode, only B is stored in compressed form, while A is compressed online within the TCU. In compressed $\times$ compressed mode, both operands are provided pre-compressed, maximizing reductions in memory traffic and computational skipping at the cost of offline software preprocessing. Together, these modes expose the trade-off between preprocessing overhead, memory-bandwidth reduction, and computation skipping across a range of sparsity scenarios. In the evaluation, we directly quantify the speedup attributable to reduced memory traffic when moving from an uncompressed to a compressed representation of A.

Memory-Bound Behavior and Direct Memory Access

The baseline Vortex TCU follows a synchronous execution model in which all operands are staged through the register file. As discussed in Chapter 3, each thread computes the addresses of its assigned matrix fragments, loads the corresponding A, B, and C elements into registers, issues the MMA instruction, and receives the output fragments back into the register file. This register-file-based dataflow introduces

significant overhead. Operand loading is serialized through per-thread register accesses, and each participating thread must reserve a large number of registers to hold its matrix fragments — 24 registers per thread in this case — both of which degrade performance. Furthermore, the result tile must be written back to memory even when it will be reused in a subsequent iteration along the K dimension, introducing unnecessary memory traffic.

This design also limits the output tile size. For a 32-thread warp, the baseline TCU computes a 16×16 FP32 output tile, since the tile dimensions are constrained by the number of registers available to the warp. This reduces input reuse compared to larger tiles. In addition, each high-level MMA instruction is expanded into multiple micro-operations; in the 32-thread configuration, one MMA instruction generates 32 micro-operations. These micro-operations increase pipeline traffic, occupy issue bandwidth, and contribute to power consumption [4, 5, 32, 33, 34]. As a result, the baseline TCU execution is dominated by operand movement and indexing calculations rather than arithmetic throughput, with measured Tensor Core utilization below 5%. Under these conditions, adding sparse computation support alone would provide limited benefit, since the primary bottleneck is data delivery rather than the Tensor Core datapath.

To address this limitation, the proposed TCU-OP is directly connected to the memory system and primarily consumes operands from shared memory. The design follows the direction of recent GPU architectures, where Tensor Core execution is increasingly decoupled from explicit register-file operand staging [35, 14, 27, 23]. In our implementation, the A, B, C and bitmap data are first transferred to shared memory by DXA, the DMA-like mechanism of Vortex. Consequently, threads are no longer required to compute fine-grained element addresses and fill many registers before invoking the Tensor Core. Instead, the TCU reads contiguous memory blocks directly from shared memory, manages operand movement internally, and executes asynchronously, decoupling the threads and the RF from the operand delivery.

Internally, the TCU can request and receive one memory block per cycle. Separate input buffers are used for A, B, and bitmap data, with two slots per buffer to support internal double buffering between shared-memory responses and FEOP consumption. This organization hides shared-memory response latency as long as the next required block is available before the current one is consumed. Memory requests are issued by a cyclic arbiter. The arbiter first prioritizes the C blocks required for accumulation-buffer initialization and then requests A, B, and bitmap blocks in the sparse case, whenever the corresponding buffer has free space.

This buffering policy is particularly important for unstructured sparsity. Since the sparsity levels of A and B may differ significantly, the consumption rates of their compressed values and metadata are not necessarily balanced. For example,

one operand may be highly sparse while the other remains nearly dense. Therefore, each stream must be requested independently according to tensor core consumption rate, rather than assuming a fixed access pattern. Internal counters track the number of requested, received, and consumed blocks for each stream, ensuring correct progress through the output tile.

When a C block arrives, it is written directly into the accumulation buffer, eliminating the need for a separate C-operand storage buffer. During computation, A, B, and optionally bitmap blocks are retained only while they are needed by the TCU core. After all sets of the current invocation have been processed, the accumulation buffer is flushed according to the destination address provided by the instruction. The result tile may be written either to shared memory or directly to global memory. Writing to shared memory enables faster on-chip reuse, while direct global-memory write-back avoids an additional shared-memory-to-global-memory transfer when the result is final.

Barrier Handling

Since TCU-OP executes asynchronously, the warp that issues the operation is not required to wait until the Tensor Core completes. This enables the warp, as well as other warps, to continue executing independent instructions while the TCU processes the output tile. However, asynchronous execution also introduces ordering hazards. A dependent warp may read the output tile before the TCU has written the final values, causing a read-after-write hazard, or another operation may overwrite a memory region that is still being updated, causing a write-after-write hazard.

To ensure correctness, the TCU-OP is integrated with the transactional barrier mechanism of Vortex. The kernel allocates a barrier object and passes its identifier as part of the TCU instruction. When the instruction is issued, the TCU registers a pending transaction with the corresponding barrier, indicating that an asynchronous operation is in progress. After the computation has completed and the result tile has been written back, the TCU releases the transaction through a dedicated connection to the barrier interface.

This mechanism follows an arrive-and-wait synchronization model. Warps may continue executing independent work after launching the TCU operation and synchronize only when they explicitly wait on the associated barrier. Therefore, the barrier preserves the required ordering between asynchronous TCU execution and dependent memory operations, without forcing the entire kernel to stall while the Tensor Core is active.

The overall organization of the proposed Tensor Core, together with its connections to the rest of the GPU, is shown in Figure 4.8.

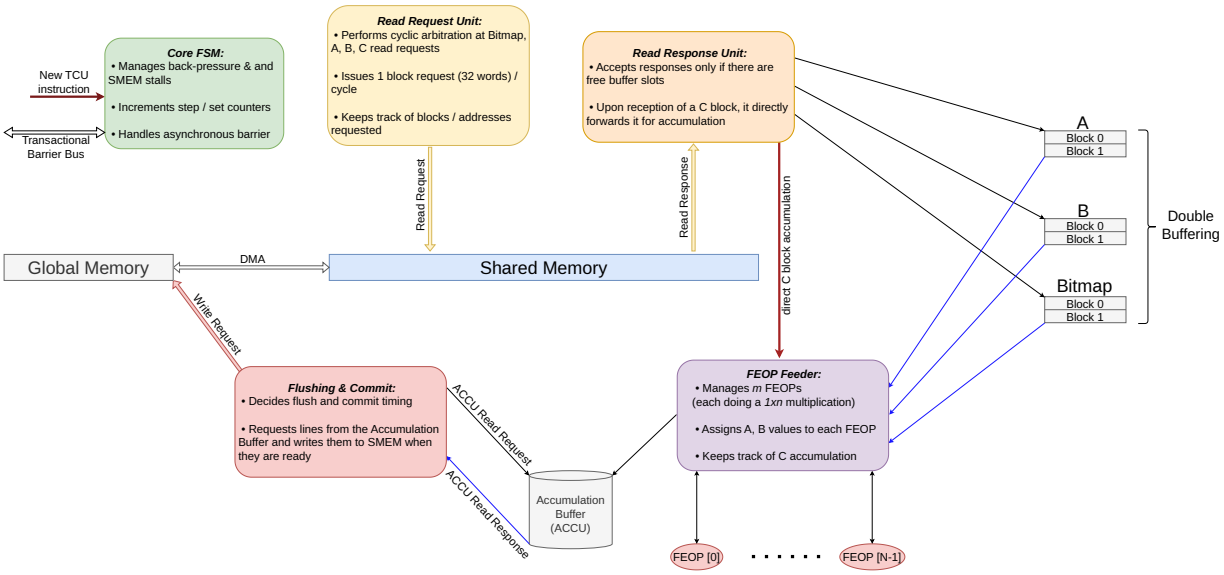


FIGURE 4.8: Overview of the proposed TCU-OP architecture and its connections to the GPU memory system. The core control logic manages step and set execution, the request and response units orchestrate shared-memory accesses for A, B, bitmap, and C blocks, and the FEOP feeder dispatches operands to the FEOP units. Partial products are accumulated in the accumulation buffer, while final results are flushed back to memory during commit.

4.5 Kernel Design

The proposed architecture is designed as a HW / SW co-design. Although the TCU-OP provides the hardware mechanisms required for direct memory access, asynchronous execution, and sparse outer-product computation, these benefits can only be fully exploited by an appropriately structured kernel. An inefficient software schedule would expose memory latency, increase synchronization overhead, and reduce the effective utilization of the Tensor Core.

The optimized kernel uses shared-memory double buffering to overlap three phases of execution: metadata generation in software, DMA-based data movement, and asynchronous TCU computation [25, 36]. As shown in Figure 4.9, the kernel prepares the metadata required for the DMA and TCU invocations while data movement and Tensor Core execution proceed concurrently. When the TCU processes tiles stored in one half of shared memory, DMA transfers the next set of tiles into the other half. This ping-pong organization allows the TCU to continue execution with minimal interruption, provided that data movement remains sufficiently overlapped with computation.

This execution flow can be orchestrated by a single warp, which computes the required metadata and launches the asynchronous DMA and TCU operations. The remaining warps are then free to execute independent work. Moreover, because

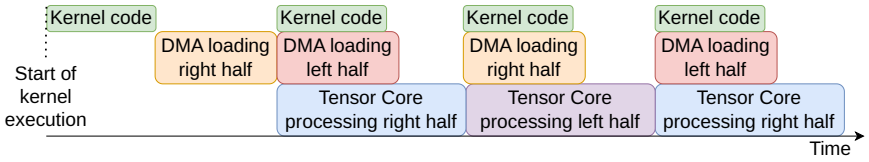


FIGURE 4.9: Kernel execution pipeline using shared-memory double buffering. Kernel-side metadata generation, DMA data transfers, and asynchronous TCU execution are overlapped.

Algorithm 1 Double-buffered TCU-OP kernel execution

Allocate two shared-memory buffers: `buf[0]` and `buf[1]`
Create DMA barriers: `dma_barrier[0]` and `dma_barrier[1]`
Create TCU barriers: `tcu_barrier[0]` and `tcu_barrier[1]`
`curr_stage` $\leftarrow 0$
`next_stage` $\leftarrow 0$

procedure LAUNCHPENDINGMMA()
    Wait on `tcu_barrier[curr_stage]`
    Wait on `dma_barrier[next_stage]`
    Issue TCU_OP using the metadata prepared for `next_stage`
    `curr_stage` $\leftarrow$ `next_stage`
    `next_stage` $\leftarrow$ `next_stage` $\oplus 1$
end procedure

for $m = 0$ to $M_{\text{tiles}} - 1$ **do**
    for $n = 0$ to $N_{\text{tiles}} - 1$ **do**
        for $k = 0$ to $K_{\text{tiles}} - 1$ **do**
            if $\neg$ `first_iter` **then**
                LAUNCHPENDINGMMA()
            end if
            Issue DMA requests for `buf[next_stage]`
            Calculate and store TCU_OP metadata for `next_stage`
        end for
    end for
end for

// Launch the final pending TCU operation
LAUNCHPENDINGMMA()
Wait on `tcu_barrier[curr_stage]`

operands are no longer staged through the register file, the register footprint of the Tensor Core invocation is significantly reduced. This avoids filling the register file with matrix fragments and reduces the risk of register spilling or additional register-file pressure. Algorithm 1 gives a high-level pseudo-code description of the double-buffered kernel execution.

The kernel organization also changes how accumulation along the K dimension is handled. Instead of repeatedly initializing and flushing the same output tile for every K-tile, TCU-OP keeps the output tile resident inside the accumulation buffer until the full K matrix dimension has been processed. The kernel traverses the M and N output tiles in the outer loops, while the K dimension is processed in the inner. As a result, initialization and flushing are performed only once per output tile, whereas the amount of computation scales with K. This amortizes the fixed overheads of accumulation-buffer initialization and write-back.

Decoupling the Tensor Core from warp register capacity also enables larger output tiles. In this work, TCU-OP computes 32×32 output tiles, improving input reuse compared to the smaller tiles supported by the register-file-based baseline. Each loaded element of A contributes to more columns of B, and vice versa. Figure 4.10 illustrates this effect. With a smaller accumulator, the same input tiles must be fetched multiple times to compute the output matrix. With a larger accumulator, the computation can be reshaped so that the same tiles are fetched once and reused for a larger number of multiplications.

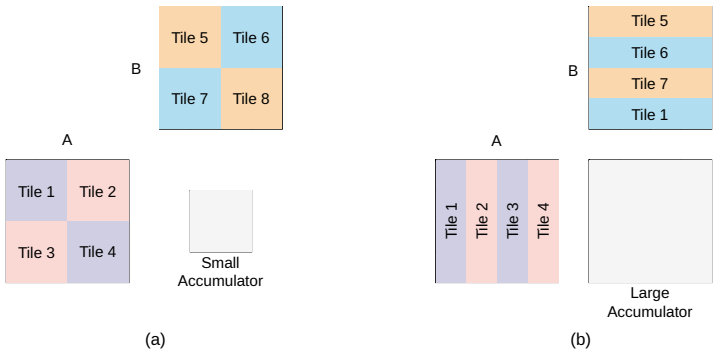


FIGURE 4.10: Effect of accumulation-buffer size on input reuse. (a) With a small accumulator, the output matrix must be computed using smaller tiles, causing input tiles to be fetched twice. (b) With a larger accumulator, the tile shape can be expanded so that each fetched input tile is reused across a larger output region, being fetched only once and doubling reuse.

4.6 Datatype Support and ISA Extension

The TCU-OP design is parameterizable with respect to input and output datatypes. The same outer-product structure can therefore support different numerical formats by modifying the multiplier, accumulator, and datapath widths, without changing the execution model of the unit. In this work, the implemented design supports FP8, FP16, FP32, and INT8 input operands, while the accumulation and output paths support FP32 and INT32 results. This allows the architecture to target both floating-point and integer GEMM workloads using the same sparse execution mechanism.

To expose the new hardware functionality to software, the ISA is extended with a new instruction that initiates an outer-product MMA operation [26]. This instruction provides all information required by the TCU-OP to locate the operands, select the execution mode, configure the datatype, and synchronize asynchronous execution.

The instruction contains the following fields:

- Base addresses of the A, B, C, and D matrix tiles, corresponding to the operation
$$D = A \times B + C.$$
- Base addresses of the A and B bitmap metadata, used in the sparse execution modes.
- Asynchronous barrier identifier, used to notify completion of the TCU operation.
- Sparsity mode, selecting among the 3 supported execution modes.
- Input and output datatype fields, according to the supported numerical formats.
- K-dimension size for the current TCU invocation.
- Initialization and flush flags. The initialization flag indicates that the accumulation buffer must be initialized with the C tile, while the flush flag indicates that the final accumulated result must be written back to memory.

Through this instruction, software can launch a complete TCU-OP operation with minimal instruction overhead. The hardware then performs operand fetching, sparse step scheduling, accumulation, and optional result flushing according to the provided configuration.

Overall, this chapter presented the complete hardware and software interface of the proposed TCU-OP design. The architecture replaces the baseline inner-product

datapath with an outer-product execution model, introduces an accumulation buffer to retain partial output tiles, supports bitmap-based sparse execution through hardware-controlled step skipping, and enables direct memory access to reduce register-file pressure and data-movement overheads. Together with the optimized kernel structure and the ISA extension described above, these mechanisms form a complete HW/SW co-designed system for dense and sparse GEMM execution. The following chapter evaluates the performance, utilization, and hardware cost of this design and quantifies the benefits of each architectural choice.

Chapter 5

Microarchitecture Optimizations and Right Sizing

5.1 Introduction

The proposed TCU-OP architecture is highly parameterizable, allowing the hardware configuration to be tuned according to the target workload and implementation constraints. Parameters such as the FEOP step dimensions, the number of parallel partial products, the accumulation-buffer organization, and the queue depth directly affect performance, area, power consumption, and timing complexity. Therefore, selecting these parameters requires balancing compute throughput against the hardware cost of the supporting structures.

This chapter examines the main microarchitectural parameters of the design and evaluates their impact on execution time and hardware efficiency. The goal is to identify a configuration that preserves most of the available performance benefit while avoiding unnecessary area and power overhead. The resulting choices are then used as the default configuration for the experimental evaluation in the following chapter.

5.2 Step Size Exploration

In the proposed design, the step size is denoted as $m \times n$ and corresponds to the region of the accumulation buffer that can be updated in one cycle. Equivalently, it determines the number of partial products generated by the FEOP datapath per cycle. This parameter directly affects the peak compute throughput of the TCU-OP, but it also determines the size of several supporting structures, including the number of multipliers, adders, queues, accumulation-buffer banks, and crossbar ports.

The baseline Vortex TCU, in the 32-thread configuration, contains 128 multipliers and the corresponding reduction hardware, allowing it to produce 128 results per cycle. However, as shown in the evaluation, the baseline design is heavily underutilized because GEMM execution is dominated by data movement rather than

computation. Therefore, preserving the same amount of compute hardware would provide limited benefit while significantly increasing the area and power cost of the proposed sparse datapath.

For this reason, the TCU-OP is right-sized to produce 32 partial products per cycle, corresponding to

$$m \times n = 32.$$

This reduces the multiplier count to one fourth of the 32-thread Vortex TCU baseline. Since the evaluated workloads are primarily memory-bound, this reduction has negligible impact on performance, while it substantially decreases the number of multipliers, adders, queues, banks, and associated control logic.

This choice is particularly important for the crossbar. The crossbar must route FEOP outputs to accumulation-buffer banks, and its cost grows rapidly with the number of inputs and outputs. A 128-input crossbar would introduce significant area, wiring, and timing overhead, making the design considerably more expensive. By limiting the number of parallel FEOP outputs to 32, the proposed design uses a crossbar size comparable to existing crossbar structures in Vortex, making the unit more practical to synthesize and integrate. Thus, the selected step size provides a balanced trade-off between compute throughput and hardware cost.

5.3 Step Dimension Exploration

After fixing the total step size to $m \times n = 32$, we further explore the shape of each step. The possible factorizations are 1×32, 2×16, 4×8, 8×4, 16×2, and 32×1. The last three configurations are symmetric to the first three with respect to the two operands, while the 1×32 configuration provides no useful sparsity granularity along the B dimension. Therefore, the main candidates considered are 2×16 and 4×8.

Let the accumulation buffer have dimensions $M \times N$. For a step of size $m \times n$, the number of vertical steps is M/m, while the number of horizontal steps is N/n. These quantities directly determine the sparsity granularity that can be exploited from the two operands. Horizontal steps correspond to the selected row of B, while vertical steps correspond to the selected column of A. For example, if only two horizontal steps are available by design, the exploitable sparsity levels for a row of B are 0%, 50%, and 100%. Thus, a row with 80% sparsity can only be rounded down to the previous exploitable level, namely 50%, allowing only half of the horizontal steps to be skipped. If four horizontal steps are available, the exploitable levels become 0%, 25%, 50%, 75%, and 100%, allowing the same row to skip $3/4$ of the horizontal steps.

These sparsity benefits are not determined only by the global sparsity of the matrix. Since the sparsity pattern is unstructured, individual rows or columns may exhibit higher local sparsity than the average matrix sparsity. For instance, even if the overall sparsity of B is 30%, some 32-element rows may still exceed 50% sparsity and therefore allow step skipping in the 2×16 configuration. This makes the proposed mechanism fine-grained, since it does not require a structured sparsity pattern across the whole matrix.

From a sparsity-granularity perspective, the 4×8 configuration appears attractive because it provides a more balanced partitioning between the A and B dimensions. However, two additional factors must be considered. First, the accumulation-buffer banks are organized so that all elements of a dense step map to different banks and can be written in parallel. If a row of the accumulation buffer spans multiple horizontal steps, flushing the row requires multiple turns. Consequently, the 4×8 configuration incurs twice the flushing overhead of the 2×16 configuration, because it uses twice as many horizontal steps.

Second, the step dimensions directly affect crossbar stalls caused by bank conflicts. For a step of size $m \times n$, the bank assigned to an output element at dense coordinates (r, c) is computed as

$$\text{bank}(r, c) = (r \bmod m) \cdot n + (c \bmod n).$$

Let R and C denote the number of distinct row and column residue classes touched by a sparse step, respectively. For example, in a 4×8 configuration, if the selected dense rows are 3, 7, 11, and 12, their residues modulo 4 are 3, 3, 3, and 0. Therefore, the distinct row residue classes are $\{0, 3\}$, and $R = 2$. The maximum possible values of R and C are m and n, respectively.

Because each sparse step performs the cross product of selected rows and columns, every row residue class is paired with every column residue class. Thus, the number of unique banks touched by the step is

$$\text{unique_banks} = R \cdot C.$$

For a 32-element step, the expected number of bank conflicts is therefore

$$\begin{aligned} \mathbb{E}[\text{conflicts}] &= 32 - \mathbb{E}[\text{unique_banks}] \\ &= 32 - \mathbb{E}[R] \cdot \mathbb{E}[C]. \end{aligned}$$

assuming that the row and column residue distributions are independent under unstructured sparsity.

For random sparse indices selected from a 32-element vector, the expected number of occupied residue classes when choosing k indices across d residue classes is

$$\mathbb{E}[\text{occupied}(d,k)] = d\left(1 - \frac{\binom{32-32/d}{k}}{\binom{32}{k}}\right),$$

where $\binom{n}{k}$ denotes the binomial coefficient.

For the 2×16 configuration, the expected number of occupied row and column residue classes is

$$\mathbb{E}[R] = 2\left(1 - \frac{\binom{16}{2}}{\binom{32}{2}}\right) = 1.516,$$

and

$$\mathbb{E}[C] = 16\left(1 - \frac{\binom{30}{16}}{\binom{32}{16}}\right) = 12.129.$$

Thus, the expected number of bank conflicts is

$$\mathbb{E}[\text{conflicts}] = 32 - 1.516 \cdot 12.129 \approx 13.6.$$

For the 4×8 configuration, the corresponding values are

$$\mathbb{E}[R] = 4\left(1 - \frac{\binom{24}{4}}{\binom{32}{4}}\right) = 2.818,$$

and

$$\mathbb{E}[C] = 8\left(1 - \frac{\binom{28}{8}}{\binom{32}{8}}\right) = 5.636.$$

Therefore,

$$\mathbb{E}[\text{conflicts}] = 32 - 2.818 \cdot 5.636 \approx 16.1.$$

This analysis shows that the 4×8 configuration introduces approximately 2.5 additional bank conflicts per sparse step compared to the 2×16 configuration. Although the queues before the crossbar are designed to absorb short-term imbalance, this difference still affects the effective throughput of the sparse datapath and is reflected in the experimental results.

To determine the most suitable step dimensions experimentally, we measure both total kernel cycles and crossbar stall percentage for the 2×16 and 4×8 configurations. The benchmark performs an $M \times N \times K = 64 \times 64 \times 512$ FP8 GEMM, using a queue depth of $Q = 4$. We evaluate four sparsity scenarios, denoted as A/B sparsity. The crossbar stall percentage is defined as the fraction of active TCU cycles in which execution is stalled because the accumulation-buffer banks

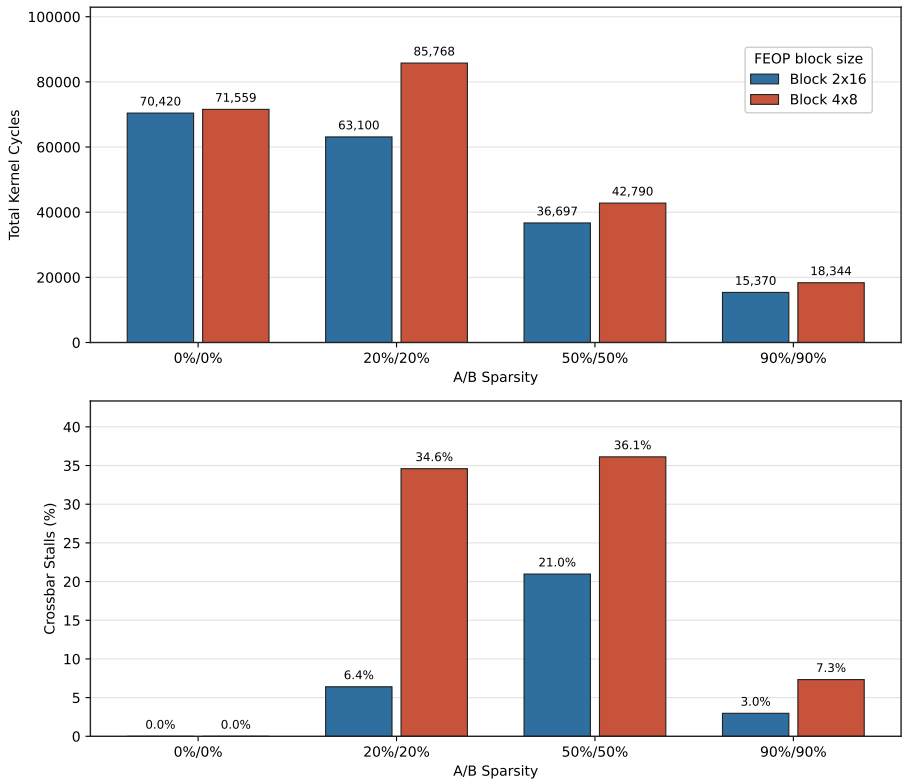


FIGURE 5.1: Effect of FEOP step dimensions on kernel cycles and crossbar stalls for a $64 \times 64 \times 512$ FP8 GEMM with queue depth $Q = 4$. The 2×16 configuration consistently achieves lower execution time than 4×8, mainly due to fewer crossbar stalls and lower flushing overhead.

cannot accept the generated partial products. The results are shown in Figure 5.1.

In the dense case, corresponding to $0\%/0\%$ sparsity, both configurations incur no crossbar stalls, as the dense access pattern maps the generated outputs to distinct banks. Nevertheless, the 4×8 configuration requires slightly more cycles than 2×16. This difference is mainly attributed to the higher flushing overhead, since 4×8 uses twice as many horizontal steps and therefore requires more turns to write back each accumulator row.

As sparsity increases, the 2×16 configuration consistently outperforms 4×8. The difference is primarily caused by crossbar stalls. At $20\%/20\%$ sparsity, the 4×8 configuration already exhibits a significantly higher stall percentage, while the stalls increase even more at $50\%/50\%$. This behavior agrees with the previous bank-conflict analysis: sparse compaction makes output addresses irregular, and the 4×8 step shape produces more bank conflicts on average. At very high sparsity, such as $90\%/90\%$, the stall percentage decreases for both configurations because fewer valid partial products are generated, allowing the queues to absorb most conflicts before back-pressure reaches the FEOP datapath.

Based on these results, the 2×16 configuration is selected for the remainder

of the evaluation. It provides lower execution time across all examined sparsity scenarios, reduces crossbar pressure, and avoids the additional flushing overhead of the 4×8 configuration.

5.4 Queue Depth Exploration

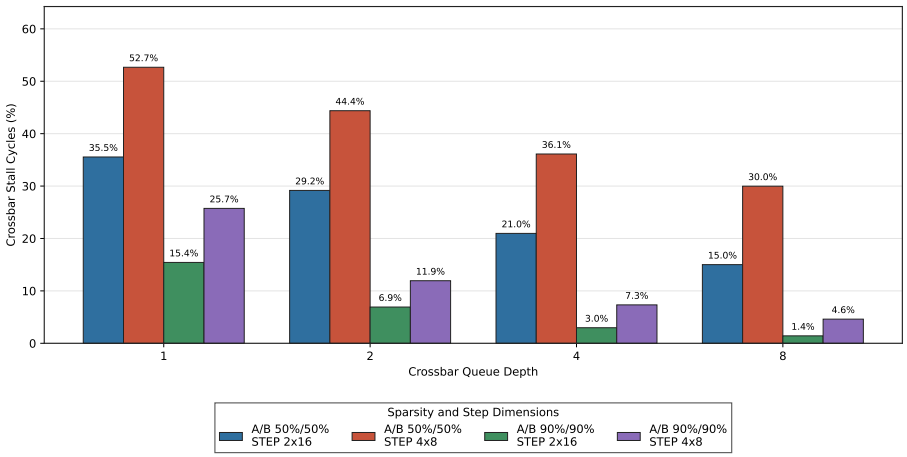


FIGURE 5.2: Effect of crossbar queue depth on stall percentage for a $64 \times 64 \times 512$ FP8 GEMM.

The previous analysis showed that crossbar stalls have a significant impact on the performance of the TCU-OP. For this reason, the queues placed before the crossbar are critical for absorbing short-term irregularities in the sparse access pattern. Deeper queues can tolerate more bank conflicts before backpressure reaches the FEOP datapath, but they also increase area and power consumption. Therefore, the queue depth must be selected carefully.

To evaluate this trade-off, we measure the crossbar stall percentage for queue depths from 1 to 8, considering both candidate step dimensions, 2×16 and 4×8. The benchmark performs an $M \times N \times K = 64 \times 64 \times 512$ FP8 GEMM and evaluates representative sparsity scenarios. The results are shown in Figure 5.2.

As expected, increasing the queue depth consistently reduces crossbar stalls. The effect is more pronounced at smaller depths, where the queues have limited ability to absorb bursts of conflicting accesses. As the depth increases, the marginal benefit decreases, indicating that most short-term conflicts have already been hidden. Across all evaluated queue depths and sparsity scenarios, the 2×16 step configuration exhibits lower stall percentages than the 4×8 configuration, confirming the trend observed in the step-dimension analysis.

Based on these results, the remaining evaluation uses a queue depth of 4. This value provides most of the stall reduction achieved by deeper queues, while avoiding the additional storage, area, and power overhead of larger queue configurations.

Chapter 6

Experimental Evaluation

6.1 Introduction

In this chapter, we evaluate the runtime performance of TCU-OP against the baseline Vortex TCU implementations for both dense and sparse workloads. The proposed design is implemented at the RTL level within the Vortex GPU, enabling cycle-accurate performance measurements. This implementation-level evaluation also exposes low-level microarchitectural effects, such as buffering behavior, pipeline stalls, and memory-system interactions, which are often hidden in higher-level simulation models.

The evaluation covers a range of hardware and software parameters in order to assess the effectiveness of the proposed HW/SW co-design. All experiments measure end-to-end kernel execution, including total cycles, instruction count, hardware activity, and any initialization or finalization overheads. Unless otherwise stated, experiments use a 32-thread warp configuration with two warps per scheduler, closely matching the execution model of modern GPU Tensor Core programming. We evaluate a single GPU core, since each contains one Tensor Core, which allows us to isolate the behavior of the unit under test.

Although TCU-OP supports asynchronous execution and allows warps to perform independent work while the Tensor Core is active, this potential overlap is not included in the current measurements. The reported results therefore focus on the direct impact of the proposed datapath, memory interface, sparse execution mechanism, and kernel organization. Quantifying the additional benefit of overlapping independent warp execution with asynchronous TCU operations is left for future work.

6.2 Benchmarks and Baselines

The evaluation uses GEMM as the main workload, since matrix multiplication is the dominant computational kernel in many machine-learning and artificial-intelligence

applications. We consider the operation

$$D = A \times B + C,$$

where A has dimensions $M \times K$, B has dimensions $K \times N$, and C and D have dimensions $M \times N$. Throughout the evaluation, this workload is denoted as an $M \times N \times K$ matrix multiplication.

To evaluate performance and hardware overhead, we compare the proposed implementation against two Vortex TCU baselines. The first baseline is SGEMM, which corresponds to the dense Vortex Tensor Core implementation described in the background chapter. This kernel does not support sparsity and follows the baseline register-file-based dataflow: operands are loaded into the register file, the Tensor Core consumes register-resident fragments, and the results are written back through the register file before being stored to global memory.

The second baseline is SGEMM_SP, the Vortex implementation of a $2{:}4$ structured sparsity mechanism similar to the one used in NVIDIA Sparse Tensor Cores. This kernel uses a dense A matrix and a sparse B matrix following the $2{:}4$ pattern, together with the required metadata. Since exactly half of the B operands are removed in each group of four values, the theoretical maximum speedup over dense SGEMM is $2\times$, assuming all other overheads are hidden.

The proposed kernel, denoted as SGEMM_OP, uses the TCU-OP unit introduced in this thesis. It is evaluated with both dense and sparse inputs and compared against the two baselines in terms of runtime performance, instruction behavior, hardware activity, and implementation overhead. This allows us to isolate the benefits of the outer-product datapath, direct memory access, accumulation-buffer reuse, and dual-side sparse execution.

6.3 Useful Metrics

To evaluate the proposed design across different parameters and runtime conditions, we collect several metrics that capture performance, memory behavior, hardware activity, and implementation cost.

The primary performance metric is the total number of kernel cycles, measured from the first instruction of the kernel to its completion. This provides an end-to-end view of each approach, including initialization, synchronization, computation, and finalization overheads. As a result, the reported cycle counts reflect the actual cost of executing the complete kernel rather than only the isolated Tensor Core operation.

We also measure the total number of executed kernel instructions. This metric indicates how much pressure each approach places on the scheduler, scoreboard,

and GPU pipeline. A lower instruction count generally suggests reduced frontend activity, lower control overhead, and greater opportunity for warps to execute independent work while asynchronous mechanisms are active.

Memory traffic is another important metric, since global memory is shared across cores and can become a major system bottleneck. Therefore, we measure the total number of global-memory read and write requests. For SGEMM_OP, we also report DMA transactions separately, since DMA is the primary mechanism used to move operands from global memory to shared memory before they are consumed by TCU-OP.

To quantify how effectively the Tensor Core hardware is used, we measure Tensor Core multiplier utilization. This metric is defined as the percentage of total kernel cycles during which the multipliers produce valid outputs. Higher utilization indicates that a larger fraction of execution time is spent performing useful arithmetic work, rather than waiting for memory, synchronization, or pipeline resources.

6.4 Performance Comparison

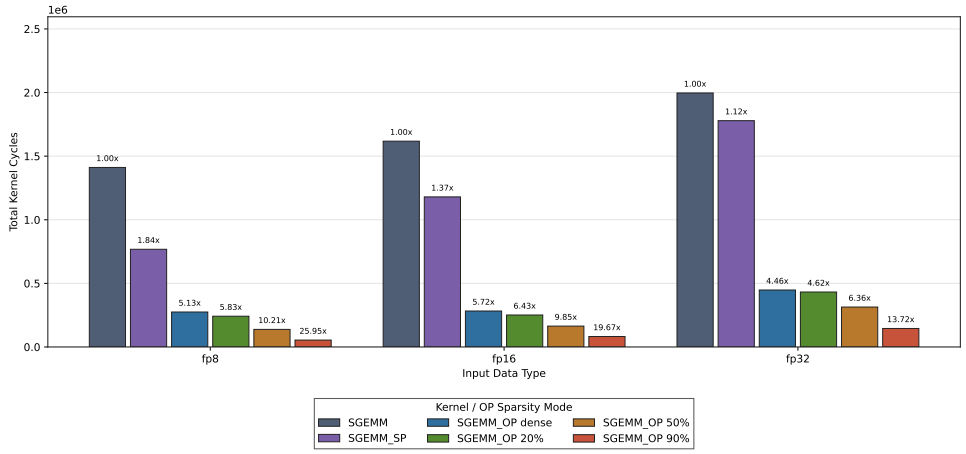


FIGURE 6.1: Kernel execution cycles for SGEMM, SGEMM_SP, and SGEMM_OP across FP8, FP16, and FP32 input datatypes. The evaluated GEMM size is $M \times N \times K = 128 \times 128 \times 512$.

We first evaluate the runtime performance of the proposed SGEMM_OP kernel against the dense SGEMM baseline and the structured-sparse SGEMM_SP baseline. For SGEMM_OP, we consider four unstructured sparsity scenarios, denoted as A/B sparsity: $0\%/0\%$, $20\%/20\%$, $50\%/50\%$, and $90\%/90\%$. In contrast, SGEMM_SP exploits structured $2{:}4$ sparsity only on the B operand, corresponding to a fixed 50% weight sparsity pattern. Therefore, while SGEMM_SP can theoretically approach a $2\times$ speedup over dense execution, it relies on a constrained sparsity pattern that

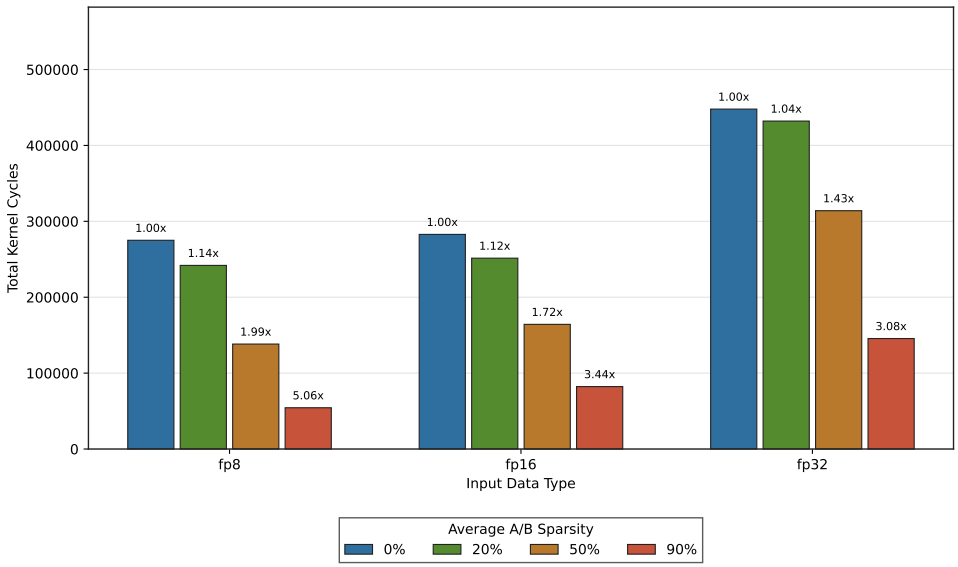


FIGURE 6.2: Kernel execution cycles of SGEMM_OP across input datatypes and unstructured A/B sparsity levels. The isolated view highlights the performance gains obtained from sparse step skipping in the proposed TCU-OP design.

may require accuracy-sensitive pruning. The sparsity evaluated for SGEMM_OP is unstructured and randomly distributed, without imposing such a pattern constraint or accuracy loss.

The evaluated GEMM size is

$$M \times N \times K = 128 \times 128 \times 512.$$

This workload size is large enough to capture the steady-state behavior of the kernel, while keeping RTL simulation time manageable. Figure 6.1 reports the total kernel cycles for FP8, FP16, and FP32 input datatypes across all evaluated kernels. Figure 6.2 isolates the SGEMM_OP results, making the effect of sparsity on the proposed design clearer.

The results show that SGEMM_OP outperforms both Vortex baselines across all evaluated datatypes. The SGEMM_SP baseline achieves speedups of $1.12\times$, $1.37\times$, and $1.84\times$ for FP32, FP16, and FP8, respectively, approaching the theoretical $2\times$ limit only for the smallest datatype. Even in the dense case, where no sparse steps can be skipped, SGEMM_OP achieves speedups of $4.46\times$, $5.72\times$, and $5.13\times$ over dense SGEMM for FP32, FP16, and FP8, respectively. Compared to SGEMM_SP, the dense SGEMM_OP kernel is still faster by $3.98\times$, $4.18\times$, and $2.79\times$, respectively.

As sparsity increases, the performance of SGEMM_OP improves further because fewer outer-product steps are executed and less sparse data is transported through the datapath. At $50\%/50\%$ sparsity, the speedup over dense SGEMM reaches $6.36\times$

for FP32, $9.85\times$ for FP16, and $10.21\times$ for FP8. At $90\%/90\%$ sparsity, the gains increase to $13.72\times$, $19.67\times$, and $25.95\times$, respectively. Relative to the structured-sparse SGEMM_SP baseline, the $90\%/90\%$ SGEMM_OP case achieves speedups of approximately $12.25\times$ for FP32, $14.36\times$ for FP16, and $14.10\times$ for FP8.

The isolated SGEMM_OP results further show the scaling effect of unstructured sparsity. Compared to its own dense execution, SGEMM_OP improves by $1.14\times$, $1.99\times$, and $5.06\times$ for FP8 at 20%, 50%, and 90% sparsity, respectively. The corresponding gains are $1.12\times$, $1.72\times$, and $3.44\times$ for FP16, and $1.04\times$, $1.43\times$, and $3.08\times$ for FP32.

The results also highlight the effect of datatype width. FP8 achieves the lowest execution time, followed by FP16 and then FP32. This behavior is expected, since narrower datatypes allow more values to be transferred per memory block and reduce the effective memory traffic for the same matrix dimensions. Consequently, the advantage of the proposed memory-oriented TCU-OP design is most pronounced for low-precision GEMM workloads, which are common in modern inference-oriented machine-learning applications.

6.5 Total Instruction Count and Front-End Pressure

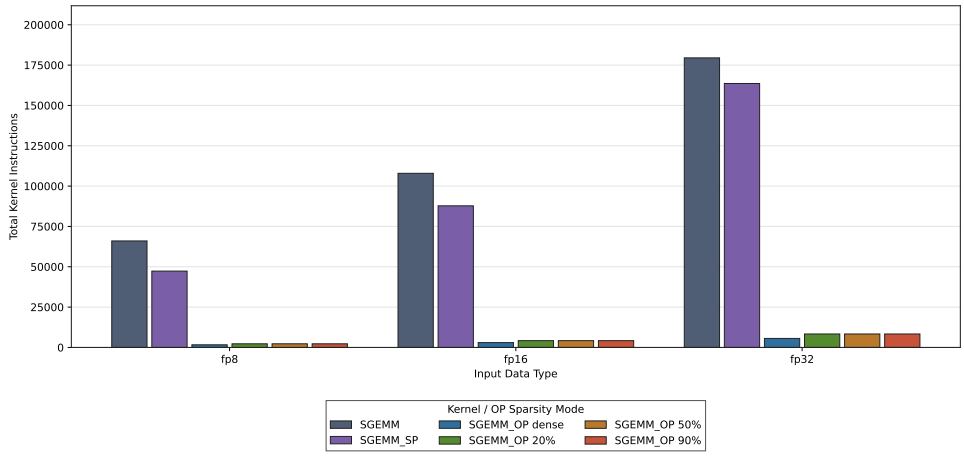


FIGURE 6.3: Total kernel instruction count for SGEMM, SGEMM_SP, and SGEMM_OP across input datatypes and sparsity modes. SGEMM_OP substantially reduces instruction count by replacing explicit register-file staging with asynchronous DMA and TCU execution.

The second metric we evaluate is the total number of kernel instructions executed from invocation to completion. Figure 6.3 reports the instruction count across all kernels, while Figure 6.4 isolates the behavior of SGEMM_OP across different sparsity levels.

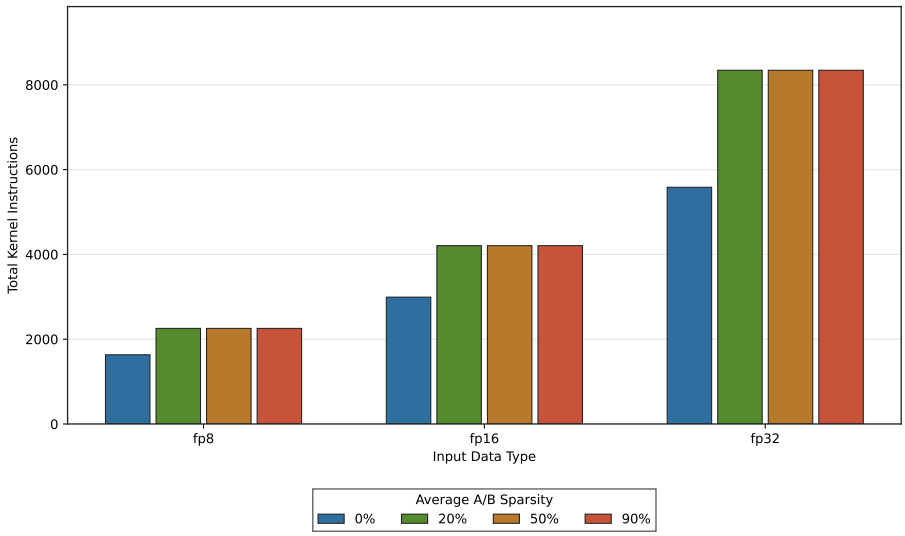


FIGURE 6.4: Instruction count of SGEMM_OP across input datatypes and sparsity levels. Sparse execution introduces a small fixed overhead due to bitmap address generation and additional DMA launches.

Across all benchmarks, the instruction count increases with datatype width. This trend is expected, since wider datatypes require more memory operations to move the same matrix dimensions, increasing the number of address-generation and data-movement instructions.

The proposed SGEMM_OP kernel executes significantly fewer instructions than both Vortex baselines. In the baseline implementations, operands are explicitly loaded through the register file, requiring pointer arithmetic, per-thread loads, and register staging for matrix fragments. In addition, each Tensor Core instruction is decomposed into multiple micro-operations, increasing pressure on the scheduler, scoreboard, and pipeline. In contrast, SGEMM_OP performs data movement and Tensor Core execution asynchronously. Each TCU invocation receives compact metadata through a single instruction, while the hardware internally performs operand fetching, scheduling, and accumulation. As a result, across the evaluated sparsity cases and datatypes, SGEMM_OP reduces the instruction count by more than an order of magnitude compared to both baselines.

Figure 6.4 shows that sparse execution introduces a nearly constant instruction overhead compared to dense SGEMM_OP. This overhead is expected, since the sparse kernel must compute bitmap source addresses in both global and shared memory and issue additional DMA transfers for the A and B bitmap metadata. Importantly, this overhead does not scale significantly with the sparsity level itself; the 20%, 50%, and 90% sparse cases have identical instruction counts. Therefore, the sparse performance gains observed earlier come primarily from hardware step skipping and reduced useful work inside the TCU, rather than from changes in the software instruction stream.

6.6 Memory Accesses

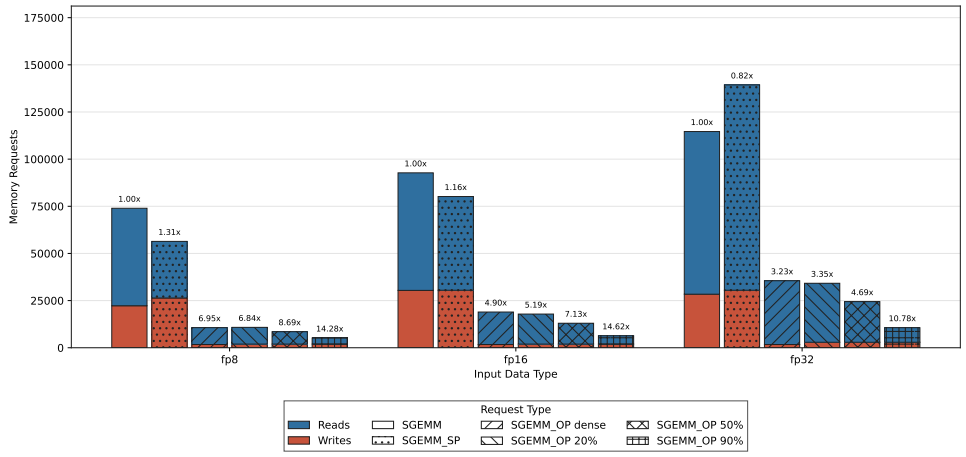


FIGURE 6.5: Total off-chip memory requests for SGEMM, SGEMM_SP, and SGEMM_OP across input datatypes and sparsity modes. Each bar is split into read and write requests.

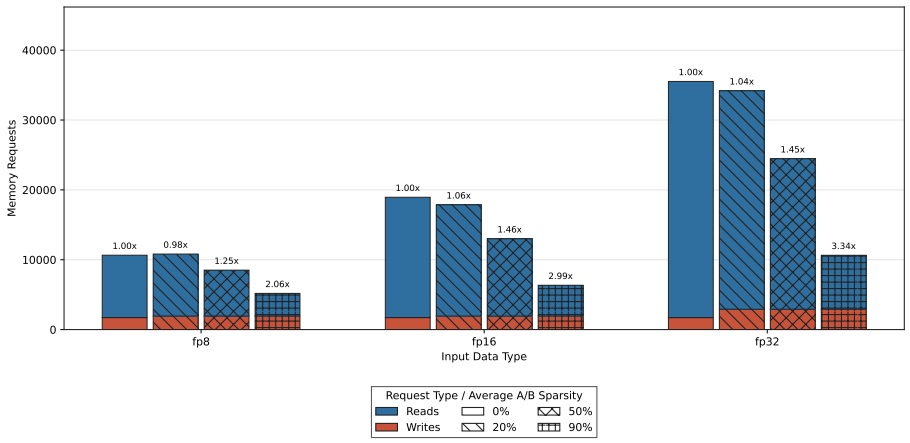


FIGURE 6.6: Off-chip memory requests of SGEMM_OP across input datatypes and unstructured sparsity levels. Higher sparsity reduces the number of transferred nonzero operand values, despite the additional bitmap traffic.

This section evaluates the memory-access behavior of the proposed design. Figures 6.5 and 6.6 report the total number of requests issued to off-chip DRAM memory, where the input operands are initially stored. Each bar is divided into read and write requests, allowing the contribution of operand loading and result write-back to be examined separately.

As expected, wider input datatypes require more memory requests for the same matrix dimensions, since fewer values fit in each memory line. For example, with a 64-byte memory line, a single access can transfer 64 FP8 values, but only 16 FP32

values. Therefore, FP32 kernels generate more memory traffic than FP16 and FP8 kernels.

The proposed SGEMM_OP kernel substantially reduces off-chip memory traffic compared to both Vortex baselines. This reduction comes from two main sources. First, the larger output tile enabled by the accumulation buffer increases reuse of the A and B operands, reducing the number of times they must be fetched from memory. Second, the proposed kernel relies on contiguous DMA transfers to shared memory, whereas the baseline implementations perform per-lane fragment loads and stores through the register file. In the baseline dataflow, values must be loaded in a layout that matches the exact register-level fragmentation required by the Tensor Core instruction. As a result, many memory lines are fetched even when only part of each line is useful. In contrast, SGEMM_OP transfers contiguous operand blocks, maximizing the efficiency of memory-line utilization.

This reduction in DRAM traffic is important because off-chip memory is a shared resource across GPU cores. Excessive memory traffic from one core can reduce the bandwidth available to other cores, even if they execute different workloads. Therefore, reducing memory requests improves not only the local efficiency of the Tensor Core kernel, but also the potential system-level behavior of the GPU.

Figure 6.6 focuses only on SGEMM_OP and shows how memory traffic changes with sparsity. For a fixed datatype, memory requests generally decrease as sparsity increases. Although the sparse kernel must also load bitmap metadata, the operand values are stored and transferred in compressed form, so only nonzero values are moved to shared memory. Consequently, higher sparsity reduces the total number of transferred operand values and lowers off-chip traffic.

The previous figures capture requests to off-chip memory. However, the Vortex memory hierarchy also includes several cache levels before external DRAM:

- an L1 instruction cache and L1 data cache at the socket level,
- an optional L2 cache shared within a cluster,
- an optional L3 cache shared across clusters,
- external memory.

To further analyze the memory behavior, Figures 6.7 and 6.8 report L1 data-cache activity. Each bar is divided into hits, read misses, and write misses.

The D-cache results show a different trend from the off-chip memory requests. While SGEMM_OP significantly reduces DRAM traffic, its D-cache accesses are proportionally even smaller compared to the baselines. This is expected because the proposed kernel elevates the role of the DMA mechanism, which transfers operands directly to shared memory instead of relying on regular cacheable load instructions.

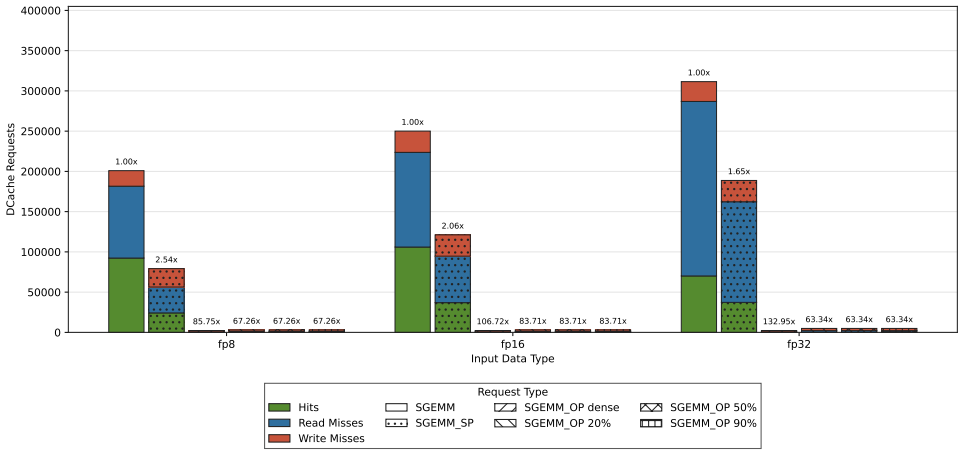


FIGURE 6.7: L1 data-cache requests for SGEMM, SGEMM_SP, and SGEMM_OP across input datatypes and sparsity modes. Each bar is split into hits, read misses, and write misses.

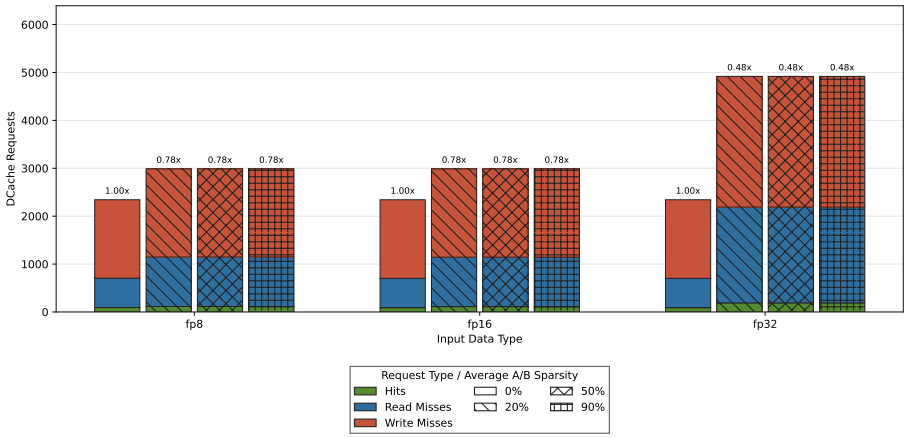


FIGURE 6.8: L1 data-cache requests of SGEMM_OP across input datatypes and sparsity levels. The requests remain constant across sparsity levels, since operand movement is performed through DMA rather than regular data-cache accesses.

As a result, operand movement does not pollute the L1 data cache or evict potentially useful data.

Figure 6.8 further shows that for SGEMM_OP D-cache activity remains nearly constant across sparsity levels. This is because all sparsity-dependent operand movement is handled by DMA transactions. The remaining D-cache accesses mainly correspond to kernel bookkeeping, metadata generation, stack accesses, and other control-related operations, which do not change significantly with the sparsity ratio.

6.7 Tensor Core Multiplier Utilization

The final metric used to compare the proposed design with the Vortex baselines is Tensor Core multiplier utilization, as defined in Section 6.3. This metric measures the fraction of kernel cycles in which the Tensor Core multipliers produce valid outputs. It therefore provides insight into how often the arithmetic datapath is active and how much execution time is lost to memory movement, synchronization, pipeline stalls, or other overheads.

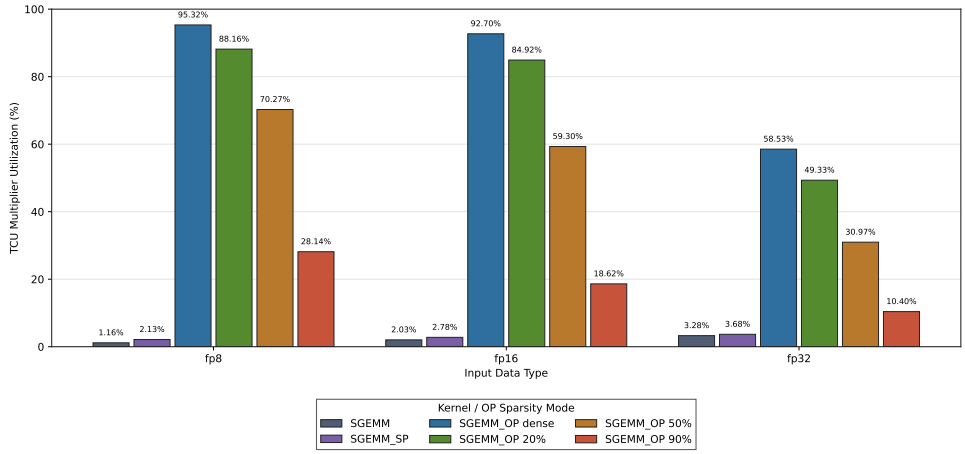


FIGURE 6.9: Tensor Core multiplier utilization across input datatypes and evaluated kernels. The baseline Vortex implementations exhibit very low utilization, while SGEMM_OP substantially improves multiplier activity by reducing operand-staging overhead and feeding the Tensor Core more efficiently.

Figure 6.9 shows the utilization of all evaluated kernels. The first two groups correspond to the Vortex dense and structured-sparse baselines, while the remaining groups correspond to SGEMM_OP under different sparsity levels.

The Vortex baselines are severely underutilized. Across the evaluated datatypes, dense SGEMM reaches a maximum utilization of only 3.28%, while SGEMM_SP reaches 3.68%. These values confirm that the baseline execution is dominated by

data movement and instruction overhead rather than by Tensor Core arithmetic. In contrast, SGEMM_OP reaches up to 95.32% utilization in the dense FP8 case, indicating that the proposed memory interface, accumulation-buffer reuse, and reduced register-file dependence allow the Tensor Core to operate close to its peak activity. A key contributor to this high utilization is the double-buffered kernel schedule described in Section 4.5, which overlaps DMA transfers, metadata preparation, and asynchronous TCU execution.

As sparsity increases, the utilization of SGEMM_OP decreases. This does not indicate that sparse execution is ineffective; rather, it reflects the fact that the Tensor Core finishes its useful arithmetic work faster. At high sparsity, many zero-only outer-product steps are skipped, reducing the number of cycles in which multipliers are active. Consequently, fixed kernel overheads, such as metadata generation and synchronization, become more visible and are no longer fully hidden by asynchronous TCU execution.

A second reason for the utilization decrease is that computation and data movement scale differently with sparsity. On average, the amount of useful outer-product computation scales with the product of the two input densities, while the amount of operand data that must be transferred scales closer to the average density of the two inputs. For example, if both A and B are 50% sparse, only $50\% \times 50\% = 25\%$ of the dense outer-product work remains, corresponding to an ideal $4\times$ reduction in computation. In contrast, the values transferred for each operand are reduced by roughly $2\times$, and the actual data-movement reduction is smaller because bitmap metadata must also be loaded. Therefore, at high sparsity, the Tensor Core arithmetic may shrink faster than the operand-fetching cost, causing the execution to become memory-bound again and reducing multiplier utilization.

Finally, utilization decreases for wider datatypes. FP32 requires more memory transfers than FP16 or FP8 for the same number of matrix elements, because fewer values fit in each memory block. This increases operand-fetching time and makes the kernel more memory-bound. The utilization trend therefore reinforces the main motivation of the proposed design: reducing data movement is essential for maintaining high Tensor Core activity.

6.8 DMA Transactions

After comparing the proposed design against the Vortex baselines, we further analyze TCU-OP-specific behavior. One important metric is the number of DMA transactions, since DMA is the mechanism used by SGEMM_OP to transfer operands from global memory to shared memory. Figure 6.10 reports the total number of

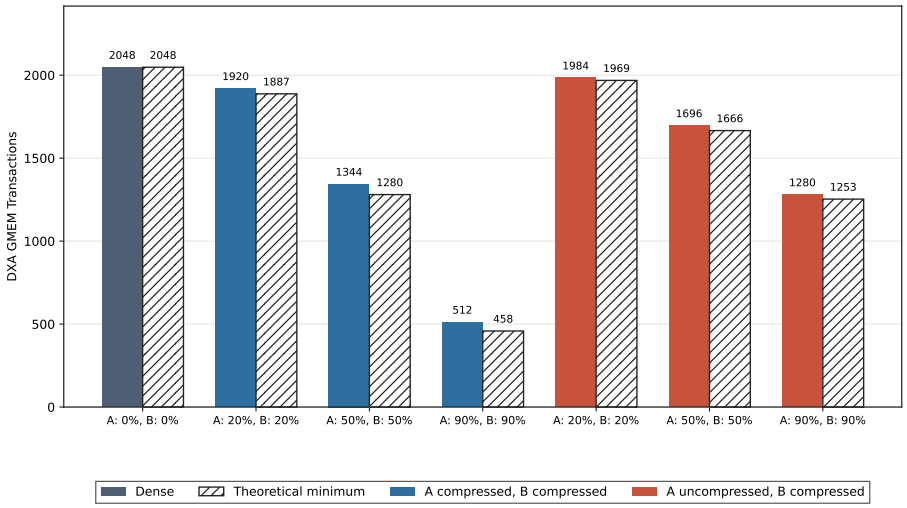


FIGURE 6.10: DMA transactions from global memory to shared memory for different sparsity modes. The dense case loads uncompressed operands, the compressed–compressed mode loads compressed A and B values together with bitmap metadata, and the uncompressed–compressed mode loads uncompressed A values together with compressed B values and bitmap metadata.

DXA transactions from global memory to shared memory across different sparsity modes.

It is important to note that DMA is used only for loading the A, B, and C matrix tiles, together with bitmap metadata when required. The final result tile is not written back through DMA. Instead, the TCU writes the contents of the accumulation buffer directly to the destination memory address specified by the instruction.

The shaded bars represent the theoretical minimum number of transactions for each configuration, taking into account the compressed operand values and the additional bitmap traffic in sparse modes. The measured results closely follow these theoretical values. This is mainly because the DMA transfers are contiguous and aligned, allowing the memory system to utilize each transaction efficiently.

The dense case requires the largest number of DMA transactions, since both operands are transferred in uncompressed form. As sparsity increases in the compressed–compressed mode, both A and B contain fewer stored values, so the number of DMA transactions decreases. At $90\%/90\%$ sparsity, the transaction count drops from 2048 in the dense case to 512, which corresponds to only 25% of the dense traffic, or a 75% reduction.

The last three bars correspond to the uncompressed–compressed mode introduced in this work. In this mode, B is stored and transferred in compressed form together with its bitmap, while A is transferred uncompressed. Therefore, the DMA traffic is lower than in the fully dense case, because the sparse B operand benefits from compression, but higher than in the compressed–compressed mode, because

zero values of A are still transferred. This mode provides a middle point between the fully uncompressed and fully compressed approaches: it captures part of the memory-traffic benefit of sparsity without requiring software preprocessing for the dynamic A operand.

Overall, the results confirm that the proposed memory interface can closely approach the minimum expected number of DMA transactions for each representation. They also illustrate the trade-off between preprocessing complexity and memory-traffic reduction across the supported sparsity modes.

6.9 K-Dimension Size Sensitivity

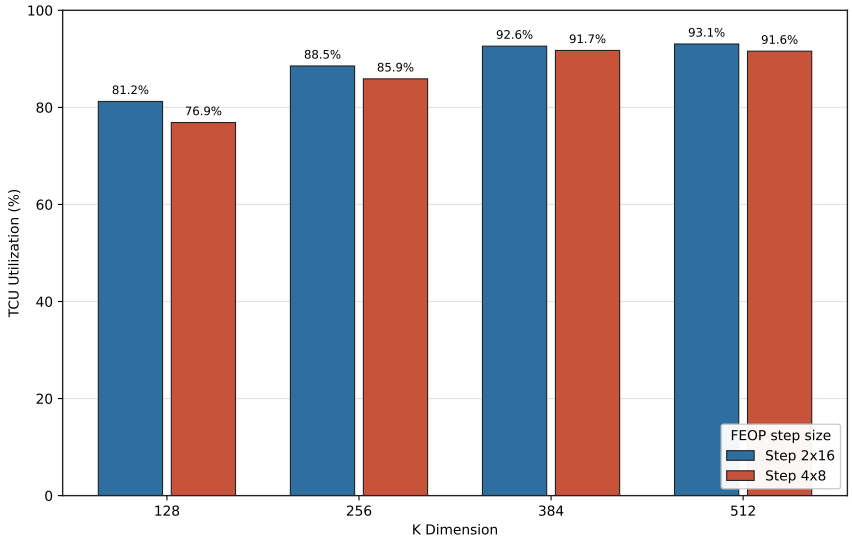


FIGURE 6.11: TCU multiplier utilization as a function of the K dimension for the 2×16 and 4×8 FEOP step configurations. The number of output tiles is kept constant, so increasing K amortizes the fixed per-output-tile overheads.

Many Tensor Core designs are sensitive to the size of the K dimension, because each output tile incurs fixed overheads for initialization, synchronization, and final write-back. These costs are amortized more effectively when the K dimension is sufficiently large, since more computation is performed for the same output tile.

Figure 6.11 shows the Tensor Core utilization in dense workloads as the K dimension varies. In this experiment, the number of output tiles is kept constant at four, while only K is swept. Therefore, the amount of per-output-tile overhead remains constant, while the computational work increases with K.

For the selected 2×16 step configuration, utilization is already 81.2% at $K = 128$. As K increases, utilization improves further and reaches a plateau of approximately 93% for $K \geq 512$. This trend confirms that the fixed overheads of kernel

code, accumulation-buffer initialization, synchronization, and flushing are increasingly amortized as the inner dimension grows.

The utilization eventually saturates instead of reaching 100%. This is mainly due to software-side synchronization and barrier handling, which introduce a fixed overhead per Tensor Core invocation. Since this overhead is associated with each invocation rather than each output tile, it limits the maximum attainable utilization even for larger K values.

The 4×8 configuration follows the same increasing trend, but remains consistently below the 2×16 configuration. This agrees with the previous analysis: although both step shapes benefit from larger K, the 4×8 configuration suffers from higher crossbar pressure and flushing overhead, leading to lower effective utilization.

The 4×8 configuration follows the same increasing trend, but remains consistently below the 2×16 configuration. This agrees with the previous analysis: although both step shapes benefit from larger K and in the dense case there are no crossbar stalls, the 4×8 configuration suffers from a higher flushing overhead, leading to lower effective utilization.

6.10 Performance Gains Across Various Sparsities

The final and most important evaluation metric of the proposed sparsity design is its end-to-end performance gain under sparse inputs relative to dense execution.

6.10.1 Compressed–Compressed Mode

We first examine the performance of the fully compressed mode, in which both A and B are stored in compressed form and are accompanied by their corresponding bitmap metadata. Although this mode represents the most aggressive form of sparse optimization, it typically requires software-side preprocessing of both operands. The preprocessing cost is not included in the present evaluation and is left as future work.

Figure 6.12 shows the total kernel cycles relative to dense execution, across a two-dimensional sparsity sweep. The horizontal axis corresponds to increasing sparsity in A, while each individual curve corresponds to a different sparsity level in B. Together, these curves form a grid of sparsity combinations, extending up to $99\%/99\%$.

The figure provides clear insight into the behavior of the proposed mechanism. Performance improves substantially as the sparsity of both operands increases, especially up to approximately $70\% - 80\%$ sparsity. In this region, the workload remains primarily compute-bound, so the reduction in useful outer-product steps

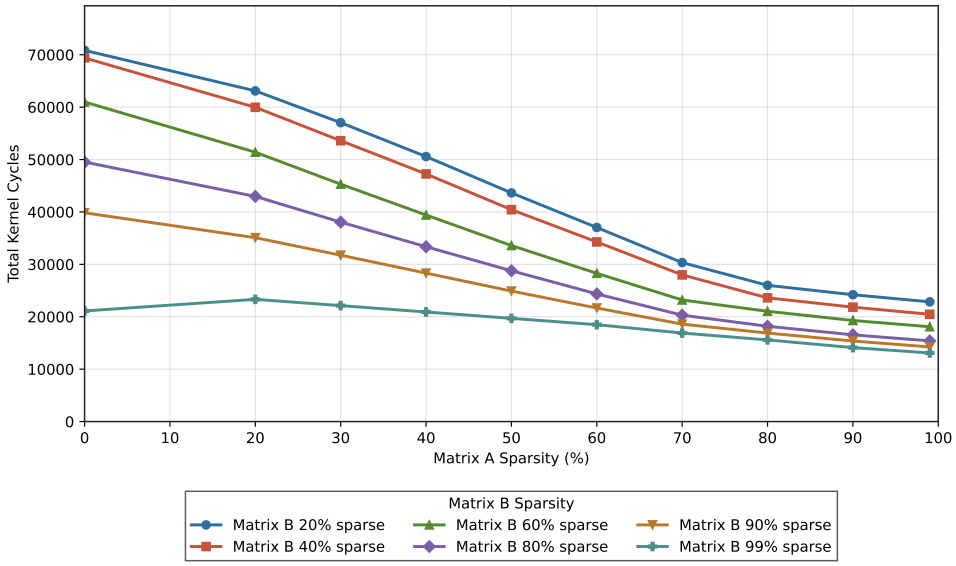


FIGURE 6.12: End-to-end kernel cycles for the compressed–compressed sparsity mode across different A and B sparsity levels. Both operands are stored in compressed form and accompanied by bitmap metadata.

translates directly into lower execution time. Beyond this point, however, the rate of improvement diminishes. As the sparsity increases further, the kernel gradually becomes memory-bound, because the reduction in computation scales faster than the reduction in data movement. Even in this regime, performance continues to improve, since compressed operands still reduce the number of memory transactions, but the gains are less dramatic than in the compute-bound region.

For a high but realistic sparsity point of $90\%/90\%$, the measured speedup is $4.58\times$. At the more extreme $99\%/99\%$ point, the speedup reaches $5.38\times$.

6.10.2 Uncompressed–Compressed Mode

The proposed design also supports the more practical uncompressed–compressed mode, in which only B is stored in compressed form, while A remains uncompressed. This mode targets the common neural-network scenario where B corresponds to static weights, while A corresponds to dynamic activations whose compression in software may be costly or impractical.

Figure 6.13 evaluates this mode across the same sparsity sweep. The solid curves correspond to the compressed–compressed results discussed previously, while the dotted curves correspond to the uncompressed–compressed case.

As expected, the uncompressed–compressed mode closely follows the compressed–compressed mode for A sparsities up to approximately 60% to 70%. In this region, the workload is still compute-bound, and therefore the benefit of compressing A is relatively small. Once the sparsity of A exceeds this range, however, the

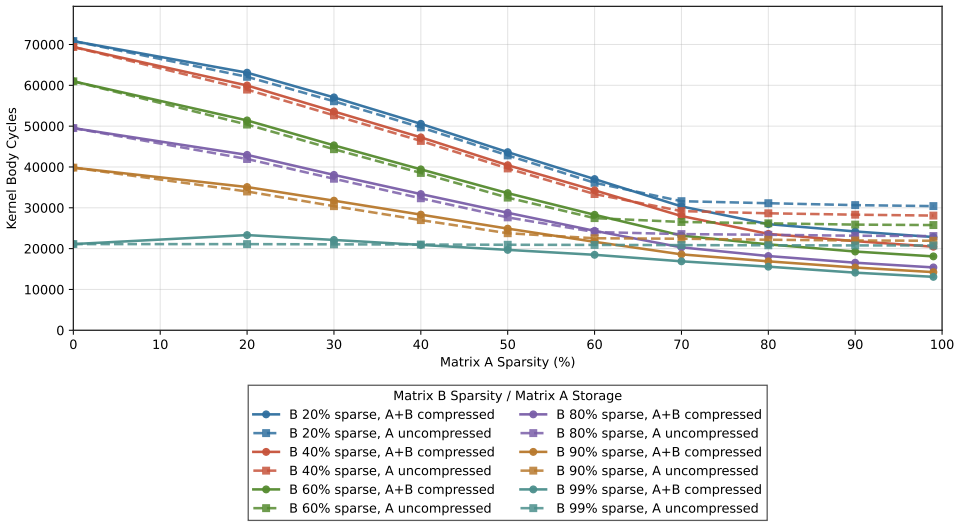


FIGURE 6.13: End-to-end kernel cycles for the uncompressed–compressed sparsity mode compared with the compressed–compressed mode. Solid lines correspond to compressed A and compressed B, while dashed lines correspond to uncompressed A and compressed B.

two modes begin to diverge. At that point, the workload becomes increasingly memory-bound, and the compressed–compressed mode benefits from lower memory traffic because only the nonzero values of A are transferred. In contrast, the uncompressed–compressed mode continues to fetch the full A tile, including zeros, and therefore its performance tends to plateau.

Even so, this plateau still corresponds to a meaningful speedup over dense execution, reaching $3.20\times$ at $90\%/90\%$ sparsity and $3.39\times$ at $99\%/99\%$ sparsity. This result shows that the uncompressed–compressed mode provides a promising middle ground: it captures a substantial fraction of the sparsity benefit while avoiding the preprocessing overhead and software complexity associated with compressing the dynamic A operand.

Chapter 7

Related Work

7.1 Introduction

This chapter positions the proposed TCU-OP architecture with respect to prior work on GPU Tensor Cores, sparse matrix multiplication, and hardware support for deep-learning acceleration. The discussion focuses on the design choices most closely related to this thesis: matrix-multiply datapaths, structured and unstructured sparsity, RTL implementation, operand delivery mechanisms, and sparse accumulation support. Prior GPU Tensor Core work established the importance of specialized matrix units for dense and mixed-precision GEMM [1, 2, 3], while later architectures added structured sparsity and more advanced data-movement support [7, 14, 27]. The DSTC mechanism [8, 9] has already been analyzed throughout the thesis as the main architectural inspiration for the proposed sparse execution model. At the end of the chapter, a summary table compares TCU-OP with the most relevant related designs and highlights the key differences of this work.

7.2 VEGETA

VEGETA proposes ISA and microarchitectural extensions for accelerating sparse-dense GEMM tile operations on CPUs [21]. It extends a CPU matrix engine with new tile instructions, metadata registers, and systolic-array modifications to support several (N:M) structured sparsity formats, including $1{:}4$, $2{:}4$, $4{:}4$, and row-wise $N{:}4$ sparsity. Therefore, VEGETA provides sparsity support, but for one sparse operand and within predefined structured sparsity modes. Although it evaluates transformed unstructured sparse layers, the computation is ultimately mapped to structured formats rather than executed as native unstructured sparsity.

TCU-OP targets a different setting. VEGETA is designed for CPUs, while TCU-OP is implemented as an RTL Tensor Core extension inside a GPU. Unlike VEGETA, TCU-OP supports dual-side sparsity and natively exploits bitmap-encoded unstructured sparsity without converting the input to an $N{:}M$ format. In addition, TCU-OP

addresses GPU-specific bottlenecks through direct shared-memory access, register-file bypass, asynchronous execution, and an on-TCU accumulation buffer. Thus, while VEGETA provides structured sparse acceleration for CPU matrix engines, TCU-OP targets synthesizable GPU Tensor Core execution with dual-side unstructured sparsity and reduced operand-delivery overhead.

7.3 Vortex Structured 2:4 Sparse Tensor Core

Another closely related baseline is the Vortex implementation of structured 2:4 sparse Tensor Core execution, following the sparsity model popularized by NVIDIA Ampere Tensor Cores [7, 6]. Vortex itself provides an open RISC-V GPGPU substrate for this kind of architectural experimentation [19]. In this format, exactly two out of every four contiguous elements along the reduction dimension are nonzero. This provides sparsity support with low hardware overhead, since the sparse operand can be compressed by $2\times$ and decoded using lightweight metadata-controlled selection logic.

Compared with TCU-OP, this approach is efficient but more restrictive. It supports sparsity only in one operand, does not target arbitrary unstructured sparsity, and keeps the baseline register-file-based Tensor Core dataflow without direct shared-memory operand access or register-file bypass. In contrast, TCU-OP supports dual-side bitmap-encoded unstructured sparsity, uses an on-TCU accumulation buffer, and directly fetches operands from shared memory. Thus, while the Vortex 2:4 Tensor Core is a synthesizable and low-overhead structured-sparse baseline, TCU-OP targets a more general sparse GEMM setting with broader sparsity support and reduced operand-delivery overhead.

7.4 Cooperative Warp Execution in Tensor Core for RISC-V GPGPU

Cooperative Warp Execution also studies Tensor Core integration in the Vortex RISC-V GPGPU [22]. It identifies register-file bandwidth, warp scheduling, dependency stalls, and pipeline overheads as important limits to Tensor Core utilization. To address them, it introduces a cooperative execution model where multiple warps operate on the same GEMM tile, partition the K dimension, and share an internal Tensor Core tile buffer. This provides a synthesizable Vortex-based design that improves dense GEMM utilization by reducing redundant register-file accesses and amortizing accumulation storage.

TCU-OP addresses a complementary problem. Cooperative Warp Execution does not target sparse GEMM and therefore does not support structured or unstructured sparsity, dual-side sparsity, bitmap decoding, nonzero compaction, or sparse step skipping. It also remains focused on improving the existing dense Tensor Core execution model, whereas TCU-OP introduces an outer-product datapath, an on-TCU accumulation buffer, direct shared-memory operand access, and register-file bypass. As a result, TCU-OP reduces both data movement and arithmetic work by exploiting bitmap-encoded unstructured sparsity in one or both operands.

7.5 Virgo

Virgo is closely related to this thesis because it also identifies the tight coupling between matrix units and SIMT cores as a major limitation [23]. It proposes a cluster-level matrix unit that is physically disaggregated from the SIMT cores, allowing operands and accumulators to bypass the register file and instead use shared memory, DMA transfers, and a dedicated accumulator memory. Therefore, Virgo provides direct shared-memory access, register-file bypass, and on-accelerator accumulation in a synthesizable GPU design, improving operation granularity, reducing instruction-processing overhead, and increasing energy efficiency for dense GEMM workloads.

TCU-OP shares the goal of reducing register-file pressure and decoupling matrix execution from the standard warp datapath, but targets sparse GEMM rather than dense GEMM scalability. While Virgo uses a systolic-array-based matrix unit and does not provide sparsity support, TCU-OP introduces an outer-product FEOP datapath and sparsity support to exploit dual-side unstructured sparsity. Therefore, Virgo is complementary to this work: it demonstrates the benefits of matrix-unit disaggregation, whereas TCU-OP addresses the additional challenges of sparse Tensor Core execution in RTL.

Table 7.1 summarizes the key differences between TCU-OP and the related designs discussed above.

TABLE 7.1: Comparison of our work with related Tensor Core designs.

Design	Sparsity Support	Dual-Side Sparsity	Unstructured Sparsity	Direct SMEM Access	On-TCU Accumulator	Register-File Bypass	Synthesizable Design
VEGETA [21]	✓	×	×	×	×	×	×
CWE in TC [22]	×	×	×	×	✓	×	✓
Virgo [23]	×	×	×	✓	✓	✓	✓
Vortex baseline TC [19, 7]	✓	×	×	×	×	×	✓
DSTC [8, 9]	✓	✓	✓	×	✓	×	×
TCU-OP (Ours)	✓	✓	✓	✓	✓	✓	✓

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis presented TCU-OP, an outer-product Tensor Core architecture for dense and sparse GEMM execution, implemented at the RTL level inside the Vortex RISC-V GPGPU. The motivation behind this work was the observation that high Tensor Core peak throughput alone is not sufficient for efficient matrix multiplication. In the baseline Vortex Tensor Core, execution is largely limited by data movement, register-file pressure, instruction overhead, and memory-system bottlenecks. These limitations become even more important for sparse workloads, where computation skipping is only useful if the hardware can also deliver operands and accumulate results efficiently.

The proposed design replaces the baseline inner-product datapath with an outer-product execution model. In this formulation, a column of A and a row of B generate partial products that update the output tile. This structure is well suited to unstructured sparsity, because nonzero values from both operands can be compacted into shorter dense vectors, while zero-only steps can be skipped. To support this execution model, TCU-OP introduces a parameterized FEOP datapath, a banked accumulation buffer, bitmap-based metadata processing, Tile Index Aligners, sparse step scheduling in hardware, and queues with a crossbar for sparse accumulation.

A central contribution of the work is the decoupling of Tensor Core execution from the register-file-based baseline dataflow. Instead of requiring each warp to load matrix fragments into registers before issuing a synchronous MMA instruction, TCU-OP reads operands directly from shared memory and executes asynchronously. Operand tiles and bitmap metadata are transferred to shared memory using DMA, while the kernel overlaps metadata generation, DMA transfers, and Tensor Core computation through double buffering. This reduces register-file pressure, lowers instruction overhead, and allows the accumulation buffer to retain the output tile across the K dimension.

The architecture supports three execution modes. The uncompressed–uncompressed mode provides dense execution, the compressed–compressed mode maximizes sparse

acceleration by using compressed operands and bitmap metadata for both A and B, and the uncompressed–compressed mode targets the practical case where B is statically compressed while A remains uncompressed and is handled dynamically inside the TCU. These modes expose a trade-off between preprocessing overhead, memory-traffic reduction, and sparse computation skipping.

The microarchitectural exploration showed that careful right sizing is essential. Since the baseline Tensor Core is heavily underutilized, TCU-OP reduces the parallel step size to $m \times n = 32$, lowering the cost of multipliers, adders, queues, banks, and crossbar without significantly affecting performance. The explored configurations showed that the 2×16 step shape and a queue depth of 4 provide a good balance between sparsity granularity, crossbar stalls, flushing overhead, and hardware cost.

The experimental evaluation demonstrated that TCU-OP improves both dense and sparse GEMM execution compared to the Vortex baselines. Even in dense mode, the proposed memory interface and accumulation-buffer reuse reduce execution time by $5\times$ compared to the baseline. Under sparse inputs, the compressed–compressed mode reaches up to $27\times$ speedup, while the uncompressed–compressed mode provides a lighter-weight alternative with $17\times$ maximum speedup without requiring software compression of the dynamic A operand. The design also reduces the total instruction count by $29.3\times$, lowers memory traffic by $30.2\times$, and increases Tensor Core multiplier utilization from below 4% in the baseline to up to 95.3% in the proposed design.

Overall, this work shows that effective sparse Tensor Core acceleration requires more than simply skipping zero multiplications. The full dataflow must be redesigned to reduce operand movement, avoid unnecessary register-file traffic, manage sparse accumulation, and expose enough parallelism to keep the hardware active. TCU-OP combines these elements into a complete HW/SW co-designed system, demonstrating that dual-side unstructured sparse GEMM can be supported in an RTL GPU Tensor Core with significant performance and efficiency benefits.

8.2 Future Work

Although TCU-OP demonstrates significant benefits for dense and sparse GEMM execution, several directions remain open for future work.

A first direction is the evaluation of preprocessing overheads for compressed operands. In this thesis, compressed matrices are assumed to be already available in memory, and the cost of generating the compacted values and bitmap metadata is not included in the main performance results. This is a reasonable assumption for static weights, which can be compressed offline, but not always for activations,

whose values change dynamically at runtime. Future work could extend the software stack with optimized compression kernels and evaluate the full end-to-end cost of sparse execution, including compression, metadata generation, Tensor Core execution, and decompression or write-back when required.

A second direction is the exploration of alternative sparse encodings. This work uses bitmap encoding because it provides simple metadata access, low decoding complexity, and good efficiency for the sparsity levels targeted by DNN workloads. However, other formats such as COO, CSR, or hybrid bitmap–CSR encodings may become more efficient at extremely high sparsity levels or for irregular HPC matrices. Supporting multiple encodings could allow the hardware or compiler to select the most appropriate representation depending on the sparsity distribution, matrix size, and target workload.

Another important direction is the optimization of the sparse accumulation network. The current design uses queues and a crossbar to route sparse partial products to the correct accumulation-buffer banks. While this approach is flexible and supports arbitrary unstructured sparsity, the crossbar remains one of the most expensive and timing-sensitive components of the datapath. Future designs could explore lower-cost routing networks, hierarchical crossbars, conflict-aware scheduling, or alternative bank mappings that reduce contention.

Another promising extension is to evaluate TCU-OP on larger and more realistic workloads beyond standalone GEMM. Many modern neural-network kernels, such as attention, fused MLP layers, and convolution lowered to GEMM, combine matrix multiplication with additional operations such as normalization, activation functions, and softmax. Since TCU-OP executes asynchronously and leaves the issuing warp free to perform independent work, future kernels could overlap Tensor Core execution with these surrounding operations. This would quantify the additional benefit of asynchronous execution that was not fully captured in the current evaluation.

Finally, further physical-design exploration would help refine the area, power, and timing trade-offs of the architecture. Although this thesis evaluates the RTL implementation and identifies the main hardware costs, future work could explore more aggressive clock targets, alternative SRAM implementations for the accumulation buffer, clock gating, power gating, and layout-aware crossbar optimization. Such analysis would provide a clearer view of the feasibility of scaling TCU-OP to larger tile sizes, wider datapaths, or multiple Tensor Core instances per GPU.

Appendix A

Source Code

A.1 Source code repository

The source code of Vortex TCU_OP can be found at:
https://github.com/GiorgosTriantafyllou324/vortex/tree/tcu-op-thesis.

Bibliography

[1] Stefano Markidis et al. “NVIDIA Tensor Core Programmability, Performance and Precision”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*. 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091. URL: https://arxiv.org/abs/1803.04014.

[2] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (visited on 06/06/2026).

[3] Md A. Raihan, Negar Goli, and Tor M. Aamodt. “Modeling Deep Learning Accelerator Enabled GPUs”. In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software*. 2019, pp. 79–92. DOI: 10.1109/ISPASS.2019.00018.

[4] Da Yan, Wei Wang, and Xiaowen Chu. “Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply”. In: *2020 IEEE International Parallel and Distributed Processing Symposium*. 2020, pp. 634–643. DOI: 10.1109/IPDPS47924.2020.00072.

[5] Wei Sun et al. “Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors”. In: *IEEE Transactions on Parallel and Distributed Systems* 34.1 (2023), pp. 246–261. DOI: 10.1109/TPDS.2022.3217824.

[6] Asit Mishra et al. *Accelerating Sparse Deep Neural Networks*. 2021. arXiv: 2104.08378 [cs.LG]. URL: https://arxiv.org/abs/2104.08378.

[7] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf (visited on 06/06/2026).

[8] Yang Wang et al. “Dual-Side Sparse Tensor Core”. In: *Proceedings of the 48th Annual International Symposium on Computer Architecture*. 2021, pp. 1083–1095. DOI: 10.1109/ISCA52012.2021.00088.

[9] Chen Zhang et al. “DSTC: Dual-Side Sparse Tensor Core for DNNs Acceleration on Modern GPU Architectures”. In: *IEEE Transactions on Computers* 74.2 (2025), pp. 341–355. DOI: 10.1109/TC.2024.3475814. URL: https://doi.org/10.1109/TC.2024.3475814.

[10] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017. URL: https://arxiv.org/abs/1706.03762.

[11] Kim Hazelwood et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture*. 2018, pp. 620–629. DOI: 10.1109/HPCA.2018.00059.

[12] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 1–12. DOI: 10.1145/3079856.3080246. URL: https://arxiv.org/abs/1704.04760.

[13] Jack Choquette, Olivier Giroux, and Denis Foley. "Volta: Performance and Programmability". In: *IEEE Micro* 38.2 (2018), pp. 42–52. DOI: 10.1109/MM.2018.022071134.

[14] NVIDIA Corporation. *NVIDIA H100 Tensor Core GPU Architecture*. 2022. URL: https://resources.nvidia.com/en-us-tensor-core/nvidia-h100-datasheet (visited on 06/06/2026).

[15] NVIDIA Corporation. *NVIDIA Blackwell Architecture Technical Brief*. 2024. URL: https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/ (visited on 06/06/2026).

[16] Song Han et al. "Learning Both Weights and Connections for Efficient Neural Network". In: *Advances in Neural Information Processing Systems*. Vol. 28. 2015. URL: https://arxiv.org/abs/1506.02626.

[17] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on Machine Learning*. 2010, pp. 807–814. URL: https://icml.cc/Conferences/2010/papers/432.pdf.

[18] Mingxing Zhu et al. "Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-Wise Sparse Neural Networks on Modern GPUs". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 359–371. DOI: 10.1145/3352460.3358299.

[19] Blaise Tine et al. *Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics*. 2021. arXiv: 2110.10857 [cs.AR]. URL: https://arxiv.org/abs/2110.10857.

[20] Mahmoud Khairy et al. "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. 2020, pp. 473–486. DOI: 10.1109/ISCA45697.2020.00047. URL: https://engineering.purdue.edu/tgrogers/publication/khairy-isca-2020/.

[21] Geonhwa Jeong et al. "VEGETA: Vertically-Integrated Extensions for Sparse/-Dense GEMM Tile Acceleration on CPUs". In: *2023 IEEE International Symposium on High-Performance Computer Architecture*. 2023, pp. 259–272. DOI: 10.1109/HPCA56546.2023.10071058. URL: https://arxiv.org/abs/2302.08687.

[22] Abubakr Nada, Giuseppe Maria Sarda, and Erwan Lenormand. "Cooperative Warp Execution in Tensor Core for RISC-V GPGPU". In: *2025 IEEE International Symposium on High Performance Computer Architecture*. 2025. DOI: 10.1109/HPCA61900.2025.00107. URL: https://doi.org/10.1109/HPCA61900.2025.00107.

[23] Hansung Kim et al. "Virgo: Cluster-level Matrix Unit Integration in GPUs for Scalability and Energy Efficiency". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2025. URL: https://arxiv.org/abs/2408.12073.

[24] Sharan Chetlur et al. *cuDNN: Efficient Primitives for Deep Learning*. 2014. arXiv: 1410.0759 [cs.NE]. URL: https://arxiv.org/abs/1410.0759.

[25] Andrew Kerr et al. *CUTLASS: Fast Linear Algebra in CUDA C++*. 2017. URL: https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/ (visited on 06/06/2026).

[26] NVIDIA Corporation. *Parallel Thread Execution ISA Version 8.5*. 2024. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html (visited on 06/06/2026).

[27] Jack Choquette. "NVIDIA Hopper H100 GPU: Scaling Performance". In: *IEEE Micro* 43.3 (2023), pp. 9–17. DOI: 10.1109/MM.2023.3256796.

[28] Weile Luo et al. "Benchmarking and Dissecting the NVIDIA Hopper GPU Architecture". In: *2024 IEEE International Parallel and Distributed Processing Symposium*. 2024, pp. 656–667. DOI: 10.1109/IPDPS57955.2024.00062.

[29] Jorge Albericio et al. “Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing”. In: *Proceedings of the 43rd Annual International Symposium on Computer Architecture*. 2016, pp. 1–13. DOI: 10.1109/ISCA.2016.11.

[30] Shijie Cao et al. “SeerNet: Predicting Convolutional Neural Network Feature-Map Sparsity Through Low-Bit Quantization”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 11216–11225. DOI: 10.1109/CVPR.2019.01148.

[31] Mengye Ren et al. “SBNet: Sparse Blocks Network for Fast Inference”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 8711–8720. DOI: 10.1109/CVPR.2018.00908.

[32] Kyeongsoon Kim and Won W. Ro. “WIR: Warp Instruction Reuse to Minimize Repeated Computations in GPUs”. In: *2018 IEEE International Symposium on High Performance Computer Architecture*. 2018, pp. 389–402. DOI: 10.1109/HPCA.2018.00041.

[33] Hyeonjin Kim et al. “Duplo: Lifting Redundant Memory Accesses of Deep Neural Networks for GPU Tensor Cores”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 2020, pp. 725–737. DOI: 10.1109/MICRO50266.2020.00064.

[34] Jae Seok Kwak et al. “INTERPRET: Inter-Warp Register Reuse for GPU Tensor Core”. In: *2023 32nd International Conference on Parallel Architectures and Compilation Techniques*. 2023, pp. 309–319. DOI: 10.1109/PACT58117.2023.00031.

[35] James E. Smith. “Decoupled Access/Execute Computer Architectures”. In: *ACM SIGARCH Computer Architecture News* 10.3 (1982), pp. 112–119. DOI: 10.1145/800046.801649.

[36] NVIDIA Corporation. *NVIDIA Ampere GPU Architecture Tuning Guide*. 2024. URL: https://docs.nvidia.com/cuda/pdf/Ampere_Tuning_Guide.pdf (visited on 06/06/2026).

[37] Angshuman Parashar et al. “SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 27–40. DOI: 10.1145/3079856.3080254. URL: https://arxiv.org/abs/1708.04485.

[38] Yu-Hsin Chen et al. "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308. DOI: 10.1109/JETCAS.2019.2910232. URL: https://arxiv.org/abs/1807.07928.

[39] Eric Qin et al. "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training". In: *2020 IEEE International Symposium on High Performance Computer Architecture*. 2020, pp. 58–70. DOI: 10.1109/HPCA47549.2020.00015. URL: https://doi.org/10.1109/HPCA47549.2020.00015.

[40] Song Han et al. "EIE: Efficient Inference Engine on Compressed Deep Neural Network". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. 2016, pp. 243–254. DOI: 10.1109/ISCA.2016.30. URL: https://arxiv.org/abs/1602.01528.

[41] Shijin Zhang et al. "Cambricon-X: An Accelerator for Sparse Neural Networks". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783723. URL: https://doi.org/10.1109/MICRO.2016.7783723.

[42] Ashish Gondimalla et al. "SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 151–165. DOI: 10.1145/3352460.3358291. URL: https://doi.org/10.1145/3352460.3358291.

[43] Trevor Gale et al. "Sparse GPU Kernels for Deep Learning". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020, pp. 1–14. DOI: 10.1109/SC41405.2020.00021. URL: https://arxiv.org/abs/2006.10901.

[44] Subhankar Pal et al. "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator". In: *2018 IEEE International Symposium on High Performance Computer Architecture*. 2018, pp. 724–736. DOI: 10.1109/HPCA.2018.00067.

[45] Zhekai Zhang et al. "SpArch: Efficient Architecture for Sparse Matrix Multiplication". In: *2020 IEEE International Symposium on High Performance Computer Architecture*. 2020, pp. 261–274. DOI: 10.1109/HPCA47549.2020.00030.

[46] Nitish Srivastava et al. “MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 2020, pp. 766–780. DOI: 10.1109/MICRO50266.2020.00067.

[47] Kartik Hegde et al. “ExTensor: An Accelerator for Sparse Tensor Algebra”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 319–333. DOI: 10.1145/3352460.3358275.

[48] Ruibo Fan, Wei Wang, and Xiaowen Chu. “DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores”. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2024. DOI: 10.1145/3620665.3640371.

[49] Guyue Huang et al. “RM-STC: Row-Merge Dataflow Inspired GPU Sparse Tensor Core for Energy-Efficient Sparse Acceleration”. In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 2023, pp. 1107–1122. DOI: 10.1145/3613424.3614261.