Connectivity, Not Coordinates: Accelerating SpMM and High-Order Tensor Contractions via Graph Abstractions

Amir Ghazizadeh Ahsaei
University of Central Florida
Orlando, FL, USA
amir.g@ucf.edu

Fangzhou Ye
University of Central Florida
Orlando, FL, USA
fangzhou.ye@ucf.edu

Hao Zheng
University of Central Florida
Orlando, FL, USA
hao.zheng@ucf.edu

Abstract

Most sparse accelerators inherit their core abstraction from dense linear algebra. The operands are treated as matrices, the kernel as a loop nest over row and column indices, and optimization as a choice of loop order, coordinate-based tiles, and compressed formats. We posit that this abstraction is inadequate for sparse computations whose data reuse is determined by connectivity among nonzeros, a property that coordinate-based representations cannot capture. We therefore introduce a graph abstraction for sparse computation, in which tiling becomes graph partitioning, dataflow becomes graph traversal, and reuse becomes a property that can be quantified before execution rather than inferred from execution. We support this position with three sets of results. Measurements on six SuiteSparse matrices show that our proposed non-contiguous tiling reduces modeled off-chip accesses by up to 4.04× over contiguous tiling. We propose two accelerators on this abstraction. *Aquila* targets conventional SpMM. It introduces vertex decomposition and adaptive depth traversal, which cluster nonzeros from discontinuous rows and columns by connectivity rather than by matrix dimensionality. A pull-after-push dataflow then raises the reuse of both dense operands while removing the synchronization cost of output accumulation. Aquila achieves a 4.88× average speedup over a prior SpMM accelerator and exploits 72.1% of the theoretical on-chip data reuse, against 39.9% on average for prior designs. *TensorPrism* targets high-order tensor contraction, using a co-occurrence graph to capture reuse across tensor modes and achieving 1.71×–2.40× geometric-mean speedups over four baseline designs on eight benchmarks. Together, these results show how graph abstraction enables graph-based non-contiguous tiling and dataflows that exploit reuse in both matrix and high-order tensor computations.

1 Introduction

Sparse-Dense Matrix Multiplication (SpMM) is a critical computation kernel in numerous domains, including machine learning and scientific computing [1, 2]. Its importance has grown as sparsity has become a property designed into models rather than a compression step applied after training, a shift traced in Figure 2. For example, Mixture-of-Experts (MoE) routing [3, 4] activates only a small subset of expert weights for each token, while sparse attention [5–7] evaluates only a fraction of the attention score matrix, so both reduce per-token FLOPs without reducing model capacity. While sparsity eliminates arithmetic, the irregular access patterns shift the performance bottleneck onto the memory system.

Prior work has approached SpMM acceleration using techniques inherited from dense linear algebra. They first fix a loop order over the row and column indices, which yields the inner-product, outer-product, and Gustavson dataflows [8–10]. They then partition the operands into tiles along contiguous index ranges so that each tile fits within the on-chip buffer [11–13]. Some methods further improve locality by permuting rows and columns to bring nonzeros closer together [2, 14–16]. Specifically, these decisions rely solely on the row and column indices of nonzeros, which we collectively call coordinate-based [8, 13]. This works well for dense matrices, where indices fully describe the data and adjacent rows map to adjacent memory locations, but breaks down for sparse matrices.

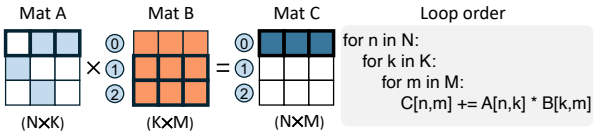


Figure 1: Row-wise Gustavson dataflow example.

However, sparsity breaks the correspondence between index and memory adjacency, as Figure 1 illustrates. For $C = A \times B$, the figure shows the Gustavson dataflow with the row-wise loop order $n \rightarrow k \rightarrow m$ used by most SpMM accelerators [10, 11, 17, 18]. While the loop order naturally captures output reuse, opportunities to reuse the input B are determined by the overlap of column indices across rows. In Figure 1, rows 0 and 2 both access $B[1, :]$, so placing them in one tile would eliminate a dense-row fetch. Yet contiguous tiling groups rows by index, which cannot guarantee that rows with overlapping column indices are placed in the same tile. This mismatch is measurable in practice, where only 39.9% of the theoretical on-chip reuse is exploited on average and up to 92% is left unused in the worst cases [19].

In this paper, we argue that the fundamental limitation of existing SpMM optimizations originates **in matrix representation rather than in the techniques built upon it.** Our key idea is to formulate the kernel as a graph problem. A becomes the adjacency structure of a graph whose edges are its nonzero elements, while rows of B and C are feature vectors attached to vertices. The translation adds no information, since the coordinates already determine every edge, but it exposes shared neighborhoods directly to the tiling and traversal decisions that govern reuse. Specifically, tiling becomes graph partitioning, growing tiles by connectivity rather than index ranges. Dataflow becomes graph traversal, allowing the choice between reusing B and C to be made per neighborhood rather than for the entire kernel. Reuse becomes a graph property, computable from vertex degrees and shared neighborhoods before execution, eliminating the empirical design-space sweeps required by coordinate representations [20]. We support this position with three sets of results, presented in Section 4. A measurement study on six SuiteSparse matrices isolates representation from hardware, showing that tiles grown by shared columns reduce modeled

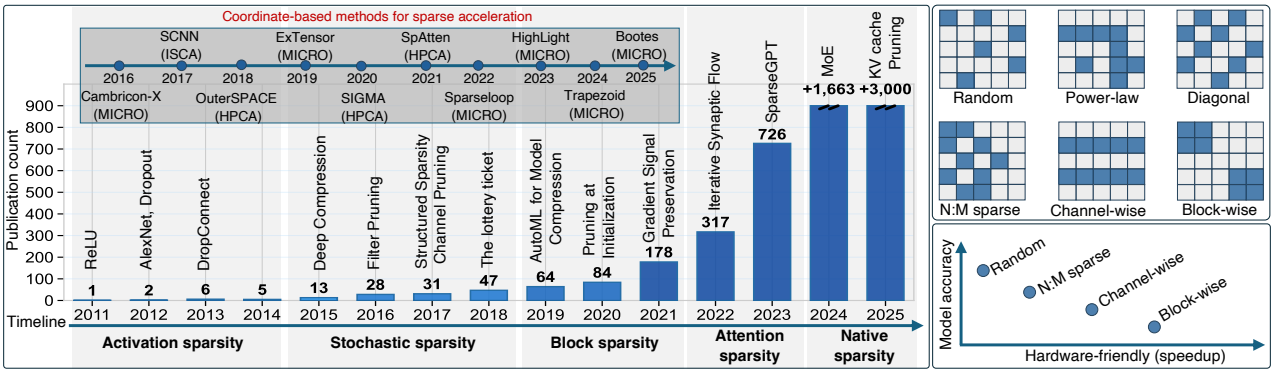


Figure 2: The growth of sparsity research and the accuracy–efficiency coupling on dense-optimized hardware.

Table 1: Design concerns under the two abstractions.

Concern	Matrix formulation	Graph formulation
Operand	coordinates + values	edges; vertices carry B/C rows
Tiling	contiguous index blocks	partition by connectivity
Load balance	nnz-count balancing	vertex decomposition
Dataflow	fixed loop order, push *or* pull	traversal, pull-after-push
Reuse metric	post-hoc, empirical sweeps	edge weights, computable a priori
Locality repair	reordering (hard, ad hoc)	PCST partitioning (linear-time heuristic)
High order	matricization (loses modes)	co-occurrence graph (keeps them)

dense-row fetches by 1.10× to 4.04× over index-based tiling and outperform both bandwidth and reuse-oriented reordering under matched budgets. We then instantiate the abstraction in two accelerators. *Aquila* applies it to SpMM, achieving a 4.88× average speedup over Sextans [12] and raising exploited reuse from a 39.9% prior-work mean to 72.1% of the theoretical maximum [19]. *TensorPrism* extends it to high-order tensor contraction, where matricization discards cross-mode reuse, achieving geometric-mean speedups of 1.71× to 2.40× over four baseline designs on eight tensors [21]. The measurement study and the reordering comparison are new to this paper, whereas all accelerator results summarize the full evaluations of the two designs.

2 Limitations of Coordinate-Based Methods

Coordinate-based methods [8–12] choose their loop order, tile boundaries, and operand permutation from the row and column indices of the nonzero elements. These methods treat each row's indices in isolation and never compare column sets across rows, so the accelerator cannot predict how many dense rows a given choice will fetch. Four limitations follow, and Table 1 contrasts each of them with the corresponding concern under a graph formulation.

Loop ordering inefficiency. SpMM dataflows are derived from loop orderings of $C = A \times B$, and each ordering fixes the direction in which operands move. Inner-product and Gustavson orderings are *pull*-style, meaning that an output row gathers the dense rows it needs. This keeps the accumulation into C local, but it re-fetches rows of B according to each sparse row's nonzero pattern [8, 10, 17]. Outer-product orderings are *push*-style, meaning that a dense row is fetched once and broadcast to every output row that needs it. This maximizes the reuse of B, but it scatters partial sums across C at a cost in buffering and synchronization [9, 22]. Adaptive designs relax this by selecting a loop order per window [11], yet each window still executes one fixed order, so the choice remains between reusing

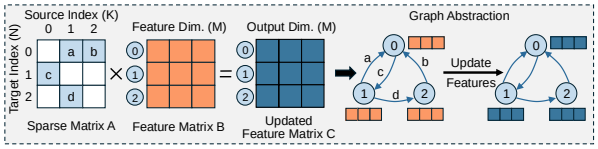


Figure 3: Example of translating SpMM as a graph.

B and reusing C rather than obtaining both within a window. High-order contraction sharpens the same bind, since the number of valid traversal orders grows with dimensionality while any single fixed order captures the reuse of one mode at the expense of the others.

Tiling ignores connectivity. Tiling partitions the operands into contiguous row or column blocks sized to the on-chip buffer, but under unstructured sparsity an index-aligned tile carries no guarantee about its contents. Two tiles of identical dimensions can hold entirely different nonzero distributions, so the traffic a tile generates is not predictable from its shape. Because B and C must also conform to the tile shape of A, dense rows are fetched speculatively and frequently never reused. Adaptive tiling [11, 12] balances the nonzero count across tiles but still respects index boundaries, so it equalizes work while leaving reuse untouched. Reordering methods [2, 14, 15], most recently through spectral clustering [16], instead move the nonzeros toward the tiles. However, reordering for locality is a hard combinatorial problem in matrix form [23, 24], and each heuristic encodes assumptions about a single class of sparsity pattern. The newest work in this line reports that some patterns do not benefit from reordering at all [16].

Reuse cannot be quantified. The matrix abstraction offers no primitive for estimating how much reuse a candidate schedule exposes. Designers therefore answer the question by sweeping configurations empirically [20], and because nothing in the representation predicts the outcome in advance, the sweep must be repeated for every new sparsity regime.

Matrix representation discards reuse opportunities. For sparse tensor contraction, the standard approach unfolds the high-order tensor into a two-dimensional matrix so that SpMM machinery can be reused [8]. Unfolding inflates the dimensions by orders of magnitude, and each choice of unfolding induces a different sparsity pattern, which makes the resulting optimization mode-dependent. More damaging, nonzeros that co-occur across several tensor modes, and that would share operand fetches under any reasonable schedule, land at distant coordinates in the unfolded matrix where no coordinate-based tiler will group them. Hypergraph formulations [25, 26] preserve the multi-way structure but

encode membership rather than pairwise coordinate overlap, and their partitioning cost grows rapidly with tensor order. The 39.9% figure measured at order two is therefore a best case, and the gap widens as order increases [19].

These consequences reach beyond kernel benchmarks and into model design itself. On dense-optimized hardware, the sparsity patterns that best preserve model accuracy, unstructured and moderate, are the ones that accelerate the worst, while the structured patterns that hardware handles well constrain the model [27]. Hooker called this dynamic the hardware lottery [28], and sparse acceleration is living through it now. The accelerator band at the top of Figure 2 traces a decade of designs, from Cambricon-X and SCNN through Trapezoid [18] and Bootes [16]. Several already consult connectivity, through islandization [2], community detection [24], or spectral clustering [16], but each uses it to permute the operand and then hands a coordinate-aligned matrix to a loop nest. The substrate, rather than model quality, is selecting which sparsity patterns win.

3 Sparse Kernels as Graphs

A sparse matrix and a directed graph are the same object under a change of notation. Each nonzero $A[i, j]$ becomes an edge from source vertex j to target vertex i, with the columns of A defining source vertices and its rows defining targets. Each row of the dense B is a feature vector carried by a source vertex, and each row of C is an accumulation at a target vertex, as shown in Figure 3. Neither half of this move is new on its own. Graph and hypergraph partitioning have modeled sparse matrix operands since the 1990s [26, 29], and direction-optimizing traversal established the push and pull duality in graph processing [30]. What is new here is keeping the graph as the execution abstraction rather than using it to permute a matrix, so that shared data dependencies become shared neighbors. Two vertices with a common neighbor require the same row of B, so a partitioner can place them in one tile deliberately, whereas coordinate-based tiling groups them only by coincidence. The two dataflow directions of Section 2 also become concrete under this view. A pull is a target vertex gathering features along its incoming edges, and a push is a source vertex broadcasting its own feature along its outgoing edges.

3.1 Sparse Dense Matrix Multiplication

We realize this view for SpMM at order two in *Aquila* through three mechanisms.

Non-contiguous tiling. Tiles are grown from the graph rather than sliced from index ranges, as Figure 6 illustrates. High-degree vertices are first split into bounded-degree children, which evens out the workload and lets several tiles reuse the same dense row. An adaptive depth traversal then grows each tile by co-locating vertices that share neighbors. Rows that lie far apart in index space but reference similar column sets end up in the same tile, so storage shrinks and reuse rises for both B and C. Figure 4 shows the effect on the Cora citation graph, where splitting vertices of degree above 16 grows the matrix from 2,708 to 2,773 rows. Regrouping rows and columns by shared neighborhoods then collects the scattered nonzeros into dense effectual blocks along the diagonal. At 64×64, the share of ineffectual tiles, those holding no nonzeros and dropping out of the schedule entirely, rises from 3.5% to 53.9%.

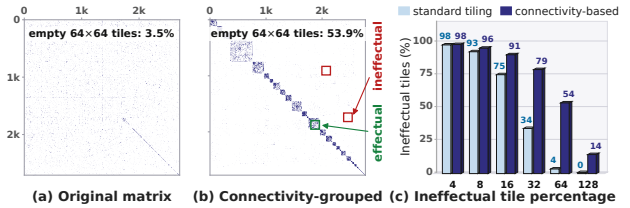


Figure 4: The connectivity transformation on the Cora graph.

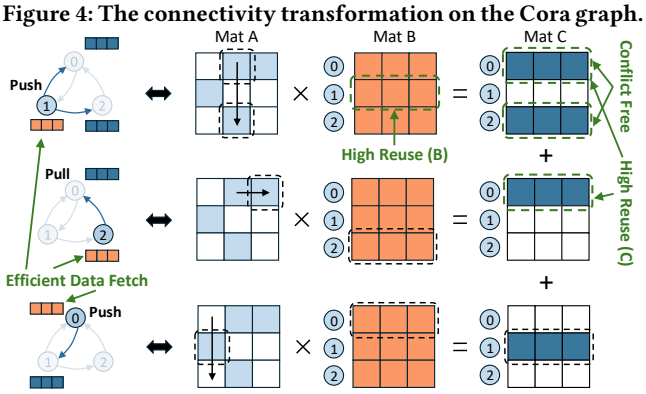


Figure 5: Proposed pull-after-push on the operand of Figure 1.

Table 2: Datasets used in the paper.

Dataset	Domain	Rows/Order	NNZ	Dataset	Domain	Rows/Order	NNZ
Reuse measurement, SuiteSparse				GAP (GAP-road)	road network	24.0M	57.7M
ca-HepPh	collaboration	12.0K	237K	MYC (mycielskian17)	graph coloring	0.10M	100M
wiki-Vote	social network	8.3K	104K	RED (Reddit)	social, GNN	0.23M	115M
memplus	circuit	17.8K	126K	PMED (PubMed)	citation, GNN	0.02M	0.09M
mac_econ_fwd500	economics	207K	1.27M	***Tensor contraction, FROSTT***			
luxembourg_osm	road network	115K	239K	uber	ride sharing	4-D	3.3M
bcsstk17	FEM, structural	11.0K	429K	nips	publication topics	4-D	3.1M
SpMM at order two				nell1 (Nell-1)	knowledge graph	3-D	144M
Cora	citation graph	2.7K	10.6K	nell2 (Nell-2)	knowledge graph	3-D	76.9M
DEL (delaunay_n24)	numerical simulation	16.8M	101M	flkr (Flickr)	social media	4-D	113M
SER (Serena)	geomechanics	1.4M	64.1M	lbnl (LBNL-network)	network traffic	5-D	1.7M
WIKI (wiki-topcats)	hyperlinks	1.8M	28.5M	chcr (Chicago-Crime)	urban records	4-D	5.3M
PAP (coPapersCiteseer)	co-authorship	0.43M	32.1M	amre (Amazon-Reviews)	recommendation	3-D	1.74B

Pull-after-push (PaP) dataflow. A loop nest must commit to one direction for the entire kernel, whereas a traversal can change direction at every vertex. PaP uses that freedom to take both directions in turn, as Figure 5 shows. It first pushes, broadcasting one fetched row of B along all outgoing edges of a source vertex, which is where the reuse of B comes from. Each of those partial products lands in the buffer of a different target vertex, so no accumulator is shared, and the synchronization cost that makes push-style dataflows expensive disappears. It then pulls, walking a target vertex's own incoming edges to finish that row of C on chip. A single schedule therefore obtains the input reuse of a push and the output reuse of a pull, which no fixed loop order can provide. A bidirectional fiber tree format interleaves row and column indices so that both directions remain cheap, in contrast to traditional row-major or column-major access.

3.2 High-Order Tensor Contraction

Sparse high-order tensor contraction is the kernel behind attention over 4D activation tensors, volumetric vision, recommendation, and quantum simulation, and it is where coordinate methods are weakest. The difficulty at high order is deciding what to build the graph from. A hypergraph models a tensor faithfully, with one vertex per mode index and one hyperedge per nonzero, but a tiler needs coordinate overlap, meaning how often two indices co-occur across nonzeros in any mode. We therefore collapse the hypergraph into a weighted *co-occurrence graph* (CoG). Vertices v_i and v_j are connected when they co-occur in at least one nonzero, and the edge

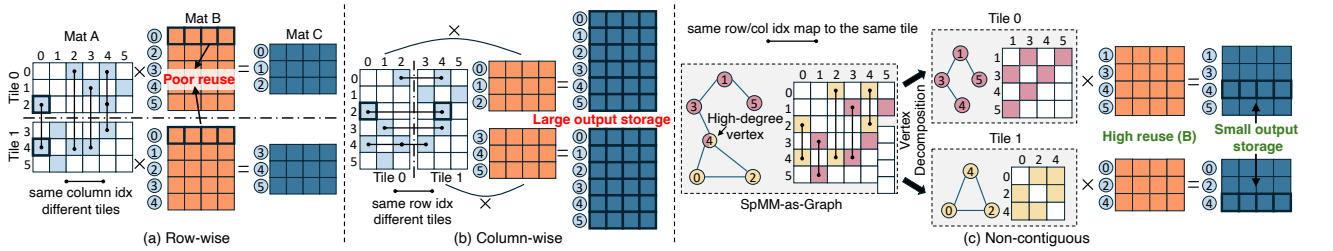


Figure 6: Coordinate-aligned (a, b) versus non-contiguous tiling (c).

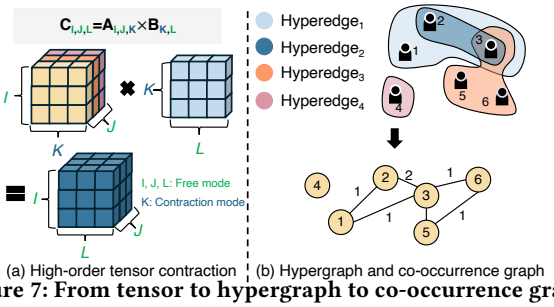


Figure 7: From tensor to hypergraph to co-occurrence graph.

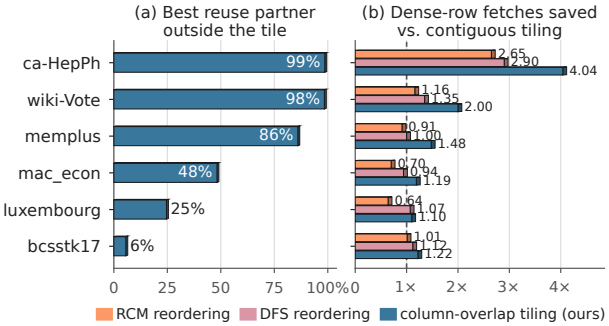


Figure 8: How much reuse lies beyond contiguous tiles (a), and how much each method recovers at matched budgets (b).

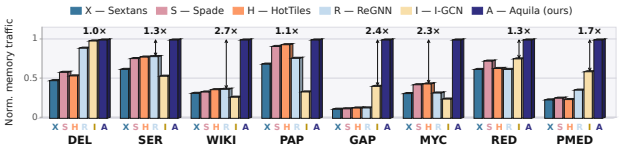


Figure 9: Off-chip traffic at order two. Bars show inverse traffic, normalized so the design moving the *least* data sits at 1, so taller is better.

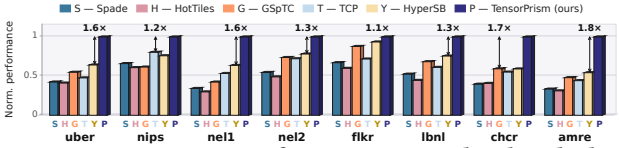


Figure 10: Contraction performance, normalized to the best design per tensor; taller is faster.

weight records the co-occurrence frequency. An edge of weight w corresponds to w operand fetches that a schedule can share, so the cross-mode reuse that matricization discards becomes explicit graph structure (Figure 7).

Tiling then becomes a Prize-Collecting Steiner Tree (PCST) problem [31] on the CoG, gathering high-weight neighborhoods into tiles subject to buffer capacity and load balance. PCST is itself NP-hard, but a primal-dual approximation [31] yields a near-linear-time heuristic on the CoG, against the rapidly growing cost of hypergraph partitioners [25, 26]. The dataflow follows by extending PaP into high dimensions over these tiles, so that reuse across all tensor modes is captured at once. Any single unfolding, by contrast, preserves only the reuse visible in its two-dimensional shadow or projection. We call the resulting design *TensorPrism*, and Section 4 reports its simulation results [21].

Architecture. Both designs share one organization, which is what makes them instances of a single approach rather than two point solutions. A scheduler builds and partitions the graph at runtime, processing elements switch traversal direction within a tile, and an aggregation unit resolves cross-tile accumulation off the critical path. In Aquila, the scheduler is a non-contiguous tiling engine that streams vertex decomposition and adaptive depth traversal. Each processing element carries a sparse and a dense buffer so that it can alternate push and pull modes, and a child-parent aggregator accumulates partial sums from decomposed children into their parents without cross-element synchronization [19]. TensorPrism keeps that structure and replaces the tiling engine with a co-occurrence graph scheduler that computes edge weights, generates the graph, and runs the partitioner. A mesh of sixteen processing elements then lets fetches issued by different tiles share the network [21]. The same organization therefore covers sparsity from extreme to moderate, at order two and above.

4 Preliminary Results

Table 2 lists the datasets used throughout the paper. Both designs are evaluated in cycle-accurate simulation paired with Ramulator [32], but on separate configurations. The order-two design uses 32 processing elements with 1K FP32 MACs at 1 GHz and a 1.25 MB buffer over HBM at 256 GB/s, matched by all baselines, while the contraction prototype uses 16 processing elements with 8K FP32 MACs at 650 MHz and a 2.5 MB global buffer over HBM2 at 307.2 GB/s. Its baselines are SPADE [33] and HotTiles [34] extended by matricization, the tensor-native GSpTC [35] and TCP [36], and HyperSB, a hypergraph-partitioning baseline built on KaHyPar [26], all scaled to matching compute and timed end-to-end, including graph construction.

Data Reuse Analysis. Rows that reference the same columns fetch the same dense rows, so they benefit from sharing a tile. For each of 2,000 sampled rows per matrix, we locate its *best reuse partner*, the row with which it shares the most columns, and test whether that partner falls inside the same 64-row contiguous tile. As Figure 8(a) shows, the partner lies outside the tile for 98.6% of rows on ca-HepPh, 98.4% on wiki-Vote, and 86.0% on memplus, while the banded FEM matrix bcsstk17 sits at 5.6% because its index locality and connectivity locality coincide. Rebuilding the tiles greedily by

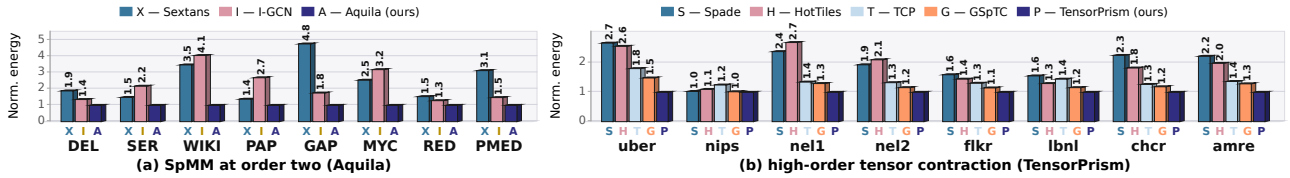


Figure 11: Total energy at order two (a) and high order (b), normalized to Aquila and TensorPrism.

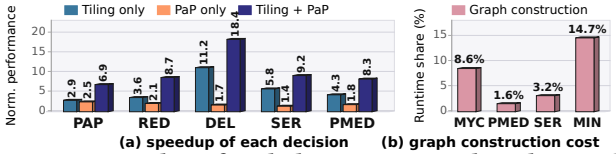


Figure 12: Speedup of each decision over index tiling with pull (a), and graph construction cost (b).

column overlap, which approximates adaptive depth traversal in software, reduces distinct dense-row fetches by 1.10× to 4.04× on every matrix, and the gain tracks panel (a) almost monotonically. RCM reordering [37] reacts inconsistently by comparison, helping ca-HepPh by 2.65× while hurting three of the six matrices, down to 0.64×, because bandwidth minimization constrains distance to the diagonal rather than shared neighborhoods. We also evaluate a reuse-oriented reordering that visits rows depth-first through shared columns, similar to islandization [2]. It improves on RCM with a geometric mean of 1.28× against 1.04×, but trails column-overlap tiling at 1.63×, because it orders rows without regard to tile boundaries and can still split rows that share columns. All three methods produce one row permutation cut into the same 64-row tiles, so buffer capacity and metadata size are held fixed.

Performance Analysis. Averaged over all datasets and feature dimensions, Aquila runs 4.88× faster than Sextans [12] and retains a margin over SPADE, HotTiles, ReGNN, and I-GCN [2, 33, 34, 38], rising to 33.5× on GAP at a feature dimension of 256 [19]. Figure 12(a) compares each decision against index-based tiling with a pull dataflow. Non-contiguous tiling alone improves performance by 5.58× on average and PaP alone by 1.91×, while the two together reach 10.30× because denser tiles give PaP more reuse to capture. At high order, Figure 10 shows TensorPrism reaching geometric-mean speedups of 2.22×, 2.40×, 1.71×, and 1.76× over SPADE, HotTiles, GSpTC, and TCP, and 1.49× over HyperSB [21]. The gap over SPADE and HotTiles is the largest because matricization destroys cross-mode locality, forcing redundant fetches that no coordinate-based tiler can recover. The gap over hypergraph partitioning is the narrowest and the most informative, since it isolates the co-occurrence graph itself, the tiler needing pairwise coordinate overlap rather than hyperedge membership.

Off-Chip Memory Access. Every reuse opportunity taken on chip is a dense-row fetch that never reaches DRAM. As Figure 9 shows, Aquila reduces off-chip accesses by 3.23×, 2.85×, and 2.81× over Sextans, SPADE, and HotTiles, and by 2.67× and 2.38× over the Graph Neural Network (GNN) accelerators ReGNN and I-GCN, whose islandization is itself a reuse-oriented reordering [2]. The per-dataset pattern is the diagnostic part, since I-GCN produces twice the traffic of Sextans on PAP yet the least among the baselines on GAP, a heuristic tuned to one sparsity structure reacting inconsistently to another. Aquila's margin over the nearest baseline runs from parity on DEL to 2.7× on WIKI, and it never trails.

Energy Efficiency. Compute energy is nearly identical across designs that execute the same operations, so the traffic reductions above appear directly as energy savings. As Figure 11 shows, Aquila reduces energy at order two by an average of 60.5% over Sextans and 55.7% over I-GCN. At high order, TensorPrism reduces energy by 44.6%, 42.3%, 26.9%, and 15.7% over SPADE, HotTiles, TCP, and GSpTC. The simulated breakdowns attribute the difference to off-chip access, where transfers account for 79.1% of total energy in TCP against 47.4% under the co-occurrence graph. We present the accelerator results as simulation-only and preliminary.

5 Where Coordinate Methods Suffice

Our claim has clear boundaries. Coordinate methods remain effective when the sparsity structure is already captured by coordinates. The banded FEM matrix is one such case, with only 5.6% of its reuse invisible to coordinates, yielding just a 1.22× reduction in modeled dense-row fetches (Figure 8). Structured N:M and block-sparse patterns are coordinate-friendly by construction, so when such regularity preserves model accuracy, dense-style hardware is the right choice. Our claim instead targets unstructured and moderate sparsity, where useful reuse is less exposed by coordinates [27].

A second boundary is the cost of building the graph. Both designs construct the graph in hardware for every kernel invocation, and all reported results include this cost. As Figure 12(b) shows, graph construction accounts for 7.3% of Aquila's runtime on average and rises to 14.7% on an attention map from MInference (MIN) [6] at 10% density. In TensorPrism, it accounts for 13.5% of the runtime on nel1, compared with 19.1% for hypergraph partitioning. Static sparsity, such as a GNN adjacency matrix or the system matrix of an iterative solver, pays this cost once and amortizes it across iterations, whereas per-query attention maps pay it on every call.

Both boundaries reduce to how much of the available reuse coordinate order already exposes, which the invisible-reuse fraction of Section 4 estimates cheaply by sampling. Our claim is therefore not that coordinates always fail, but that tiling and traversal decisions must consult connectivity when coordinate order hides the reuse.

6 Conclusion

Coordinate-based sparse acceleration leaves roughly 60% of the available on-chip reuse unexploited, because the row and column indices of an operand record where its nonzeros are stored rather than which of them can share a fetch. Formulating the kernel as a graph makes that structure explicit, turning tiling into partitioning, dataflow into traversal, and reuse into a quantity computable before execution. As sparsity becomes central to efficient foundation models, the hardware beneath it needs an abstraction that can represent it.

References

[1] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *TMS*, 38(1):1–25, 2011.

[2] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *Proc. of MICRO*, pages 1051–1063. IEEE, 2021.

[3] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *Proc. of ICLR*, 2017.

[4] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *JMLR*, 23(120):1–39, 2022.

[5] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. In *Proc. of NeurIPS*, volume 33, pages 17283–17297, 2020.

[6] Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, et al. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. In *Proc. of NeurIPS*, volume 37, pages 52481–52515, 2024.

[7] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proc. of ACL*, 2025.

[8] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proc. of MICRO*, 2019.

[9] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *Proc. of HPCA*, pages 724–736. IEEE, 2018.

[10] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proc. of ASPLOS*, pages 687–701. ACM, 2021.

[11] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. Spada: Accelerating sparse matrix multiplication with adaptive dataflow. In *Proc. of ASPLOS*, pages 747–761. ACM, 2023.

[12] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proc. of FPGA*, pages 65–77. ACM, 2022.

[13] Zi Yu Xue, Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. Tailors: Accelerating sparse tensor algebra by overbooking buffer capacity. In *Proc. of MICRO*, pages 1347–1363, 2023.

[14] Chris Mueller. Sparse matrix reordering algorithms for cluster identification. *MLB*, 23, 2004.

[15] Yi-Jou Hsiao, Chin-Fu Nien, and Hsiang-Yun Cheng. Respar: Reordering algorithm for reram-based sparse matrix-vector multiplication accelerator. In *Proc. of ICCD*, pages 260–268. IEEE, 2021.

[16] Sanjali Yadav and Bahar Asgari. Bootes: Boosting the efficiency of sparse accelerators using spectral clustering. In *Proc. of MICRO*, 2025.

[17] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *Proc. of MICRO*, pages 766–780. IEEE, 2020.

[18] Yifan Yang, Joel S Emer, and Daniel Sanchez. Trapezoid: A versatile accelerator for dense and sparse matrix multiplications. In *Proc. of ISCA*, pages 931–945, 2024.

[19] Amir Ghazizadeh Ahsaei, Lingxiang Yin, Shilin Tian, Fangzhou Ye, Fan Yao, and Hao Zheng. Rethinking tiling and dataflow for spmm acceleration: A graph transformation framework. In *Proc. of MICRO*, 2025.

[20] Namjoo Amirmahdi Yadav, Sanjali and Bahar Asgari. Misam: Machine learning assisted dataflow selection in accelerators for sparse matrix multiplication. In *Proc. of MICRO*, 2025.

[21] Fangzhou Ye, Shilin Tian, Amir Ghazizadeh Ahsaei, and Hao Zheng. Tensorprism: Rethinking sparse high-order tensor acceleration via co-occurrence graph. In *Proc. of ISCA*, pages 291–306. IEEE, 2026.

[22] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *Proc. of HPCA*, pages 261–274. IEEE, 2020.

[23] Paolo Sylos Labini, Massimo Bernaschi, Werner Nutt, Francesco Silvestri, and Flavio Vella. Blocking sparse matrices to leverage dense-specific multiplication. In *Proc. of IA3 Workshop*, pages 19–24. IEEE, 2022.

[24] Vignesh Balaji, Neal C Crago, Aamer Jaleel, and Stephen W Keckler. Community-based matrix reordering for sparse linear algebra optimization. In *Proc. of ISPASS*, pages 214–223. IEEE, 2023.

[25] Gunduz Vehbi Demirci, Cagatay Dikici, and Tim Atherton. Accelerating sparse deep learning via multi-layer tensor reordering and partitioning. In *Proc. of HPEC*, pages 1–8, 2025.

[26] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *JEA*, 27:1–39, 2023.

[27] Jinhao Li, Jiaming Xu, Shan Huang, Yonghua Chen, Wen Li, Jun Liu, Yaoxiu Lian, Jiayi Pan, Li Ding, Hao Zhou, Yu Wang, and Guohao Dai. Large language model inference acceleration: A comprehensive hardware perspective. *arXiv preprint arXiv:2410.04466*, 2024.

[28] Sara Hooker. The hardware lottery. *Commun. ACM*, 2021.

[29] Ümit Veysel Çatalyürek. *Hypergraph models for sparse matrix partitioning and reordering*. PhD thesis, Bilkent Universitesi, 1999.

[30] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[31] David S Johnson, Maria Minkoff, and Steven Phillips. The prize collecting steiner tree problem: theory and practice. In *Proc. of SODA*, pages 760–769, 2000.

[32] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *CAL*, 15(1):45–49, 2015.

[33] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. Spade: A flexible and scalable accelerator for spmm and sddmm. In *Proc. of ISCA*, 2023.

[34] Gerasimos Gerogiannis, Sriram Aananthakrishnan, Josep Torrellas, and Ibrahim Hur. Hottiles: Accelerating spmm with heterogeneous accelerator architectures. In *Proc. of HPCA*, pages 1012–1028. IEEE, 2024.

[35] Guoqing Xiao, Chuanghui Yin, Yuedan Chen, Mingxing Duan, and Kenli Li. Efficient utilization of multi-threading parallelism on heterogeneous systems for sparse tensor contraction. *TPDS*, 35(6):1044–1055, 2024.

[36] Hanjoon Kim, Younggeun Choi, Junyoung Park, Byeongwook Bae, Hyunmin Jeong, Sang Min Lee, Jeseung Yeon, Minho Kim, Changjae Park, Boncheol Gu, et al. Tcp: A tensor contraction processor for ai workloads industrial product. In *Proc. of ISCA*, pages 890–902. IEEE, 2024.

[37] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. of ACM'69*, pages 157–172, 1969.

[38] Cen Chen, Kenli Li, Yangfan Li, and Xiaofeng Zou. Regnn: A redundancy-eliminated graph neural networks accelerator. In *Proc. of HPCA*, pages 429–443. IEEE, 2022.