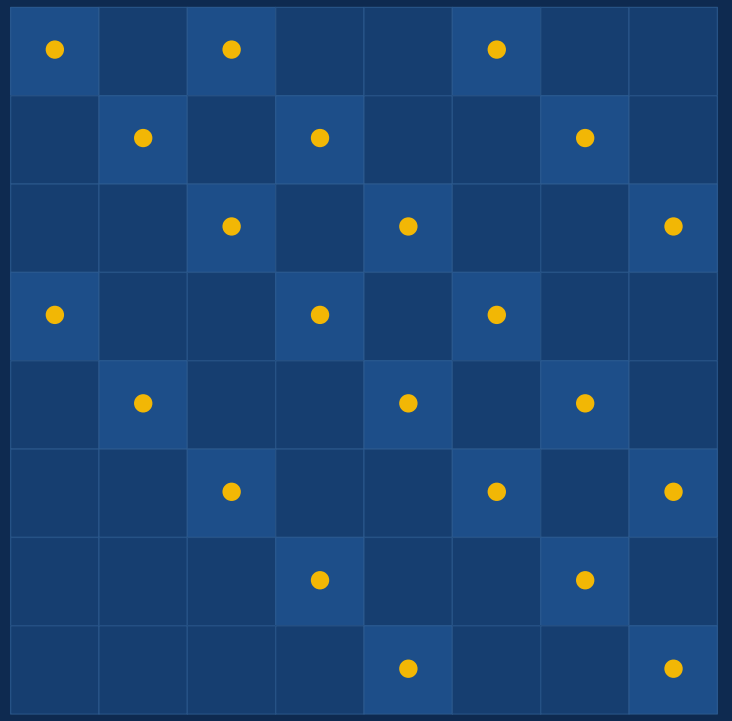
Computing with Diagonally Structured Matrices

CPU and GPU-Accelerated Matrix Multiplication

International Congress on Mathematical Software
July 20 – 23, 2026
Waterloo, Canada

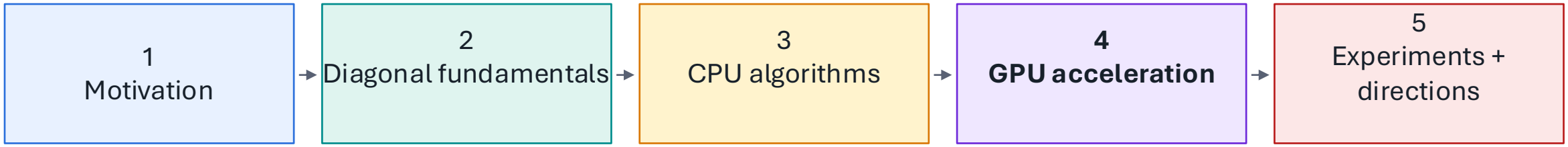
Shahadat Hossain · Department of Computer Science
University of Northern British Columbia

$C \leftarrow C + AB, \quad C \leftarrow C + A^T B, \quad C \leftarrow C + AB^T$



Make diagonals
first-class computational objects

Outline



Diagonal storage turns inner-product-style memory access into vector-like diagonal interactions that expose **locality**, **independent work**, and **inexpensive transpose access.**

Why revisit matrix multiplication?

Row/column view

Rows of A
columns of B
inner products

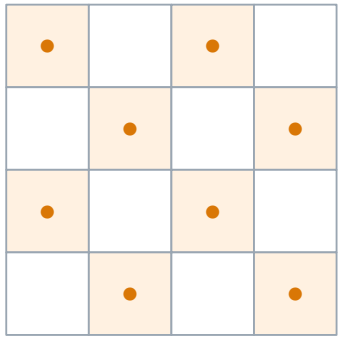
Works well for dense GEMM when the data layout and access are highly tuned; less natural for structured sparse bands and transposed access.

Diagonal view

$$C_k \leftarrow C_k + \sum_i A_i \odot B_{k-i}$$

Contiguous diagonal vectors become the units of arithmetic and parallel scheduling.

Nonzero diagonals



The arithmetic remains as in standard matrix multiplication; the data organization and computation ordering change.

Fundamentals of diagonal representation

Each entry a_{ij} has a diagonal offset p = j − i and position r = min(i,j) inside that diagonal.

$$d_p(A) = \{ a_{ij} \mid j - i = p \}$$
$$r(a_{ij}) = \min\{i, j\}$$

Interpretation

p>0 **Superdiagonal**

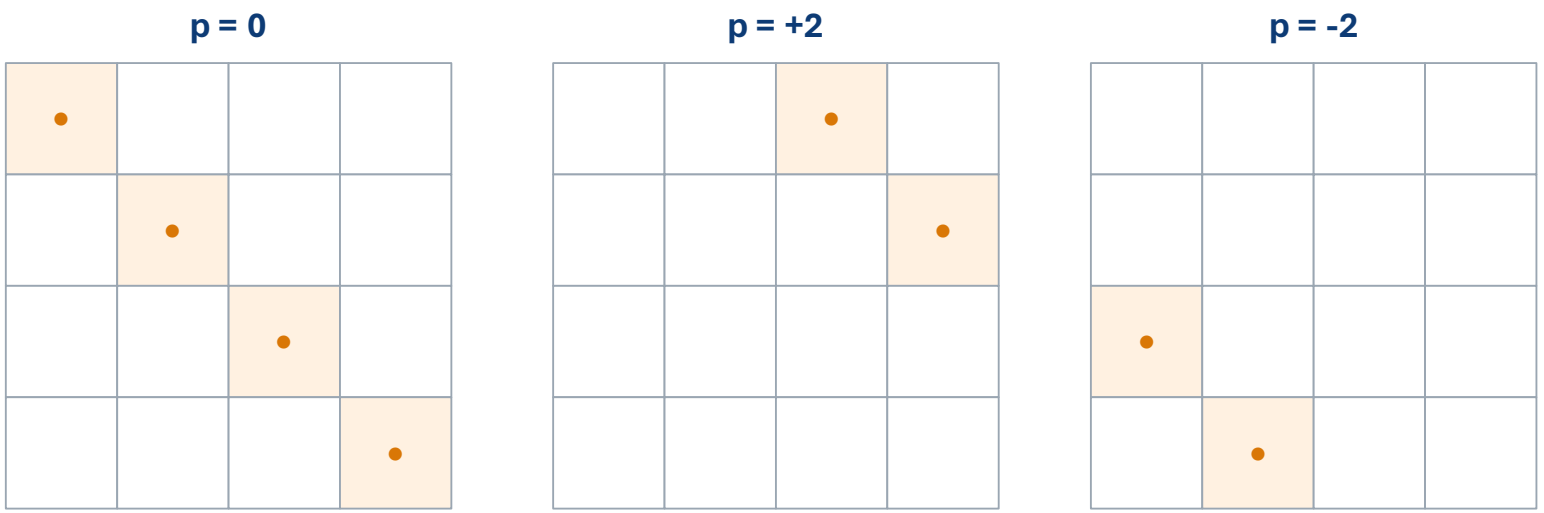
above the principal diagonal

p=0 **Principal diagonal**

entries a00, a11, …

p<0 **Subdiagonal**

below the principal diagonal



The diagonal vector order is consistent, independent of whether the original matrix would normally be stored in row-major or column-major order

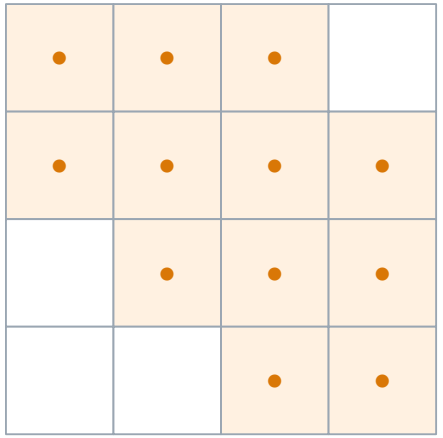
HM storage: packed diagonals without padding

A one-dimensional array stores only the diagonals that belong to the structure.

$$(A_0 \mid A_1 \mid \cdots \mid A_{k_u} \mid A_{-1} \mid \cdots \mid A_{-k_l})$$

$$N_{\mathrm{HM}} = n(k_u + k_l + 1) - \frac{k_u(k_u+1)}{2} - \frac{k_l(k_l+1)}{2}$$

Example: ku = 2, kl = 1



Consequences

1 No redundant padding

Unlike BLAS-style band storage, HM stores true nonzero diagonal entries.

2 Uniform structures

Dense, banded, triangular, symmetric, and structured sparse matrices can be stored by diagonals.

3 Vector traversal

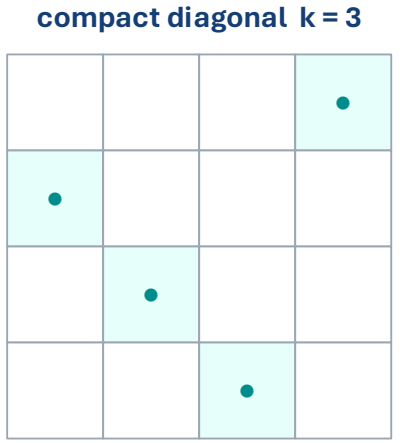
Within each diagonal, elements are contiguous and can be processed with stride-1 access.

HM is especially natural for **structured-sparse i.e.,** when nonzero diagonals are arbitrarily distributed.

CDM storage: compact paired diagonals

CDM packs compatible super- and subdiagonals into length-n compact diagonals.

$$A_k^{(c)} = (A_k\,;\, A_{-(n-k)})^{\mathrm{T}}, \qquad \widetilde{A}_k^{(c)} = (A_{-(n-k)}\,;\, A_k)^{\mathrm{T}}$$



Why compact diagonal?

For banded matrices, CDM regularizes the computation: compact diagonal rows (as a 2-d array) have uniform length and can be tiled for GPU thread blocks.

$$C_0 \leftarrow C_0 + A_0 \odot B_0 + \sum_{k=1}^{n-1} A_k^{(c)} \odot \widetilde{B}_k^{(c)}$$

For n = 4, the compact vector combines A3 with A-1.

CDM is the GPU-facing representation for banded multiplication

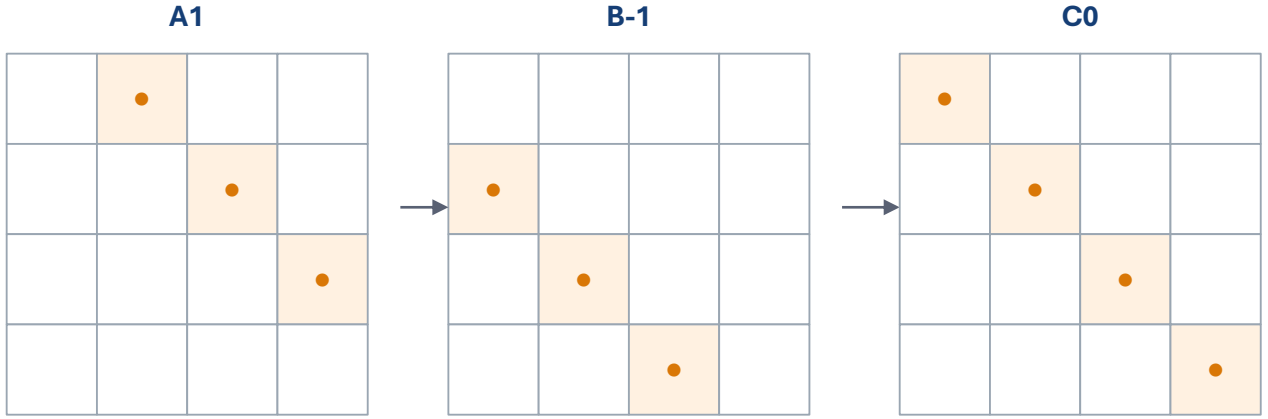
Matrix multiplication by diagonals

A product diagonal is built from the diagonal pairs whose labels are compatible.

$$C_k \leftarrow C_k + \sum_i A_i \odot B_{k-i}$$

$$A_i \odot B_j \longrightarrow C_{i+j}$$

The key operation is an elementwise multiply-add between diagonal vectors.



$$C_k^{(c)} \leftarrow C_k^{(c)} + A_k^{(c)} \odot B_0^{r_k} + \sum_{i=1}^{n-1} A_{(k+i) \bmod n}^{(c)} \odot (\widetilde{B}_i^{(c)})^{r_k}$$

For CDM, the same idea appears as **shifted compact-diagonal interactions**.

Transpose access is a storage-level operation

This is central to matrix-transpose multiplication and BLAS-like updates.

$$D_k(A^{\mathrm{T}}) = \widetilde{D}_k(A) \quad \Longrightarrow \quad \text{transpose access without data movement}$$

Intuition

A **Before transpose**

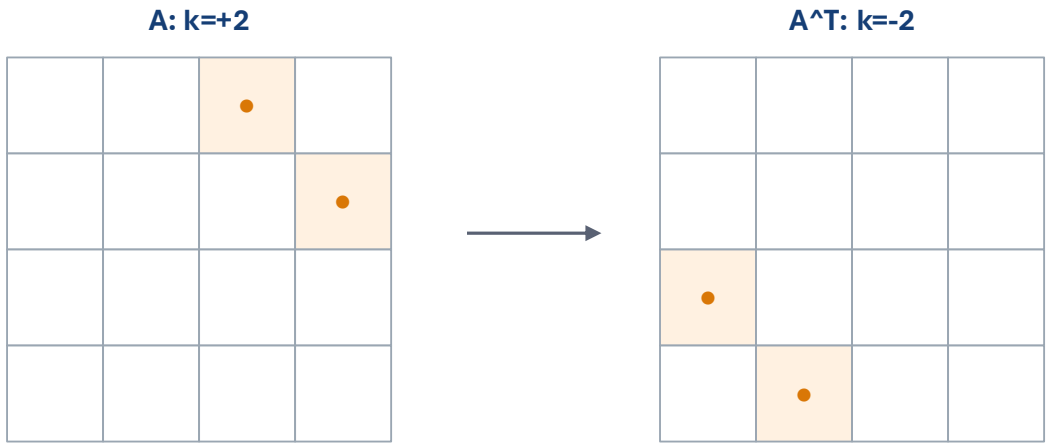
aij lies on the +k superdiagonal.

AT **After transpose**

the same value lies on the -k subdiagonal.

CDM **Compact reverse**

swap the compatible pair; no full matrix data movement is required.



Avoiding explicit transpose is a direct path to improving matrix-transpose kernels.

Sequential and shared-memory CPU view

The same diagonal formulation exposes contiguous work units on a multicore CPU.

Sequential baseline

Loop over output diagonals
Loop over compatible pairs
Vector multiply-add

Shared-memory parallelism

Independent output diagonals
OpenMP-style tasks
Minimal synchronization

Hardware benefit

Stride-1 access
Better spatial locality
Less index chasing

$$\text{speedup} = t_{\text{CPU}}/t_{\text{GPU}}$$

The CPU version is not the endpoint; it is the reference point for demonstrating how diagonal work maps to massively threaded GPUs.

GPU acceleration: what changes?

The diagonal abstraction becomes a map from matrix structure to thread-block structure.

$$T \times T \text{ thread block}, \qquad T = 2^d,\ d = 4$$

1 Thread mapping

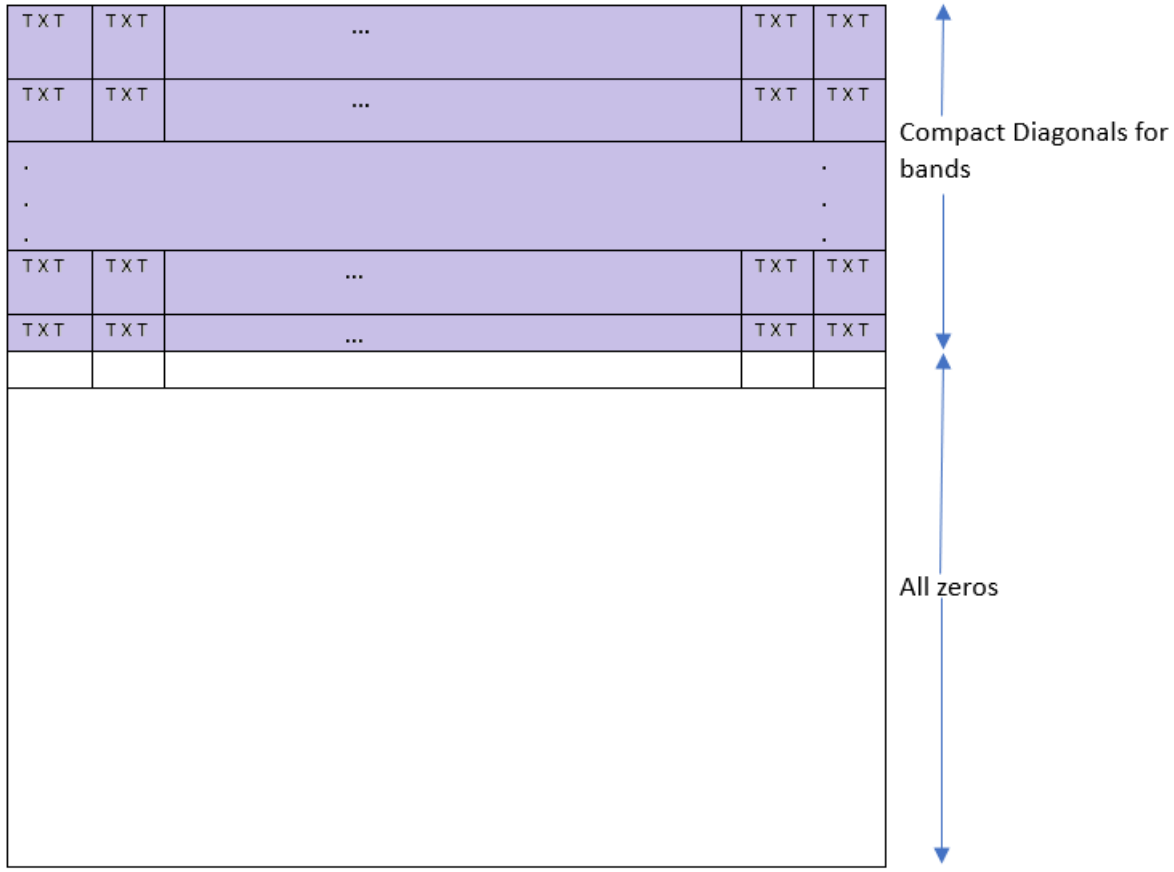
A thread computes one entry of C in the banded CDM algorithm.

2 Shared memory

A T-by-T tile of A is loaded into shared memory and reused across T multiply-adds.

3 Global memory

B is accessed from global memory because its reuse is too small to justify shared-memory traffic.



Banded matrices on GPU: CDM + tiling

CDM regularity makes the banded case a good fit for a tiled CUDA kernel.

Representation

dgX
DsX
Xlist

For a banded matrix X, CDM stores compact diagonals as rows and records the diagonal indices explicitly.

$$C_k^{(c)} \leftarrow C_k^{(c)} + A_k^{(c)} \odot B_0^{r_k} + \sum_{i=1}^{n-1} A_{(k+i) \bmod n}^{(c)} \odot (\widetilde{B}_i^{(c)})^{r_k}$$

Kernel outline

for each C tile
  load A tile into shared memory
  synchronize threads
  multiply with B entries
  write C entry

Memory for product C can be preallocated because the compact diagonal indices are known before numerical multiplication.

Structured sparse matrices on GPU: HM + atomic updates

For arbitrarily distributed nonzero diagonals, each nonzero from A drives work across B diagonals.

$$X = (d_g^X,\ D_s^X,\ X_{\text{list}},\ \text{StartDg}^X)$$

$$a_{ij} \mapsto D_{j-i}(A)[\min(i,j)], \qquad b_{jk} \mapsto D_{k-j}(B)[\min(j,k)]$$

Kernel strategy

1 One thread per nonzero of A

The number of GPU threads is nzA.

2 Loop through B diagonals

Find the matching B entry and its contribution location in C.

3 Update C safely

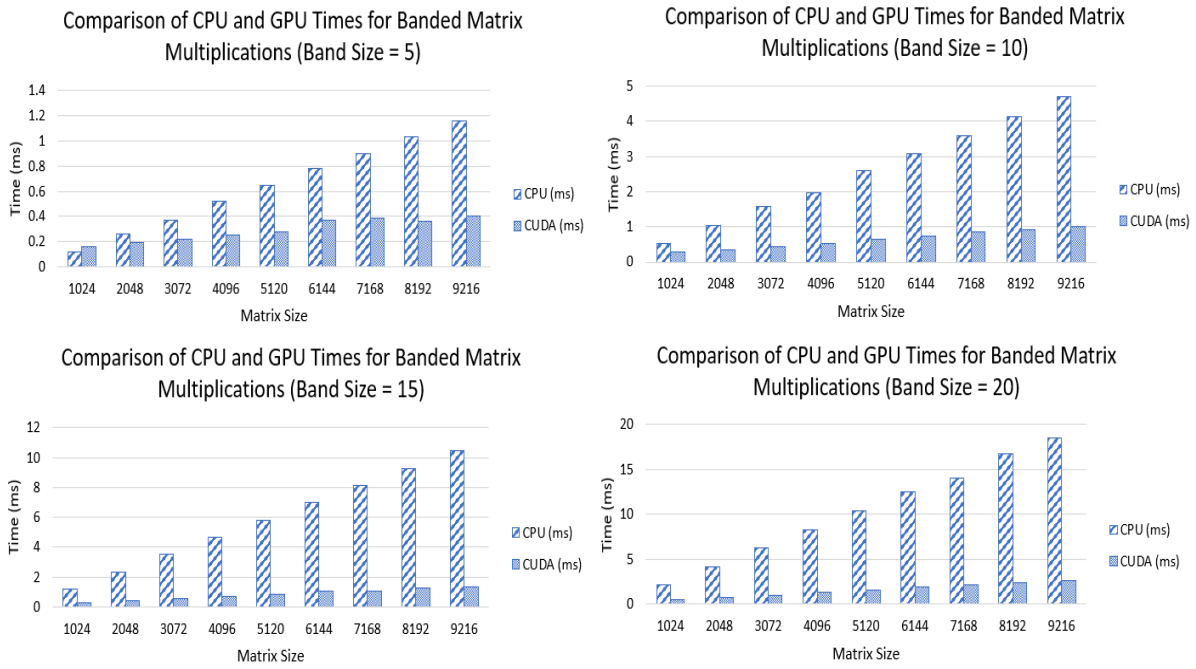
Multiple threads may contribute to the same C entry, so the update uses an atomic operation.

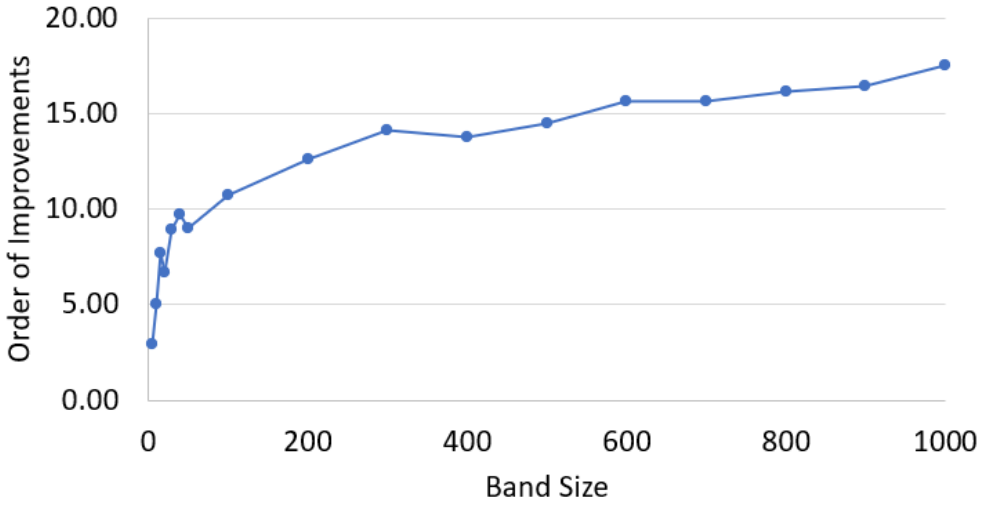
$$C_{d_C}[p_C] \mathrel{+}= A_{d_A}[p_A]\, B_{d_B}[p_B] \quad \text{(atomic add)}$$

This approach minimizes the diagonal-pair search overhead compared with assigning one thread to each possible output entry.

Numerical evidence I: banded matrices

GPU acceleration scales with matrix size and band size.



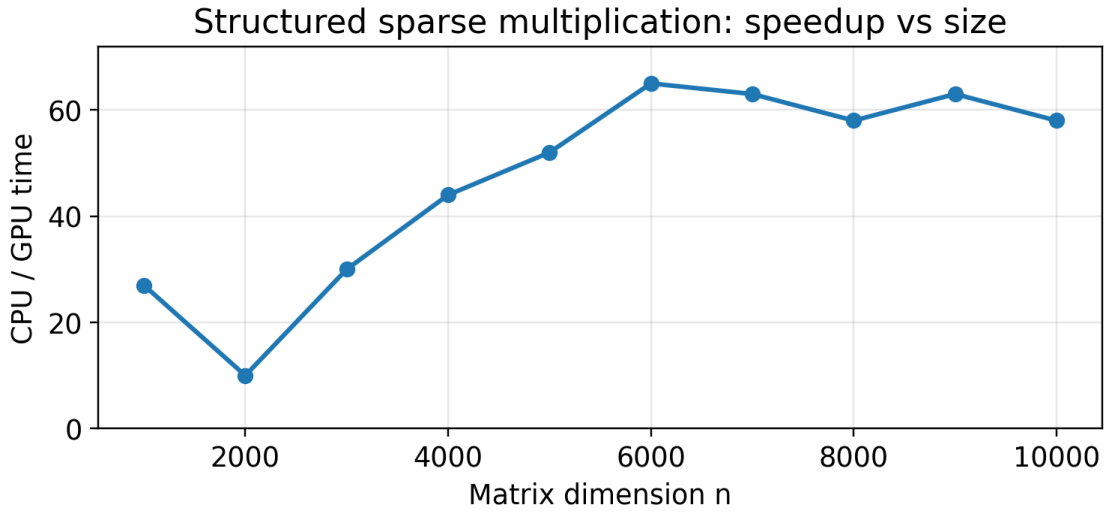


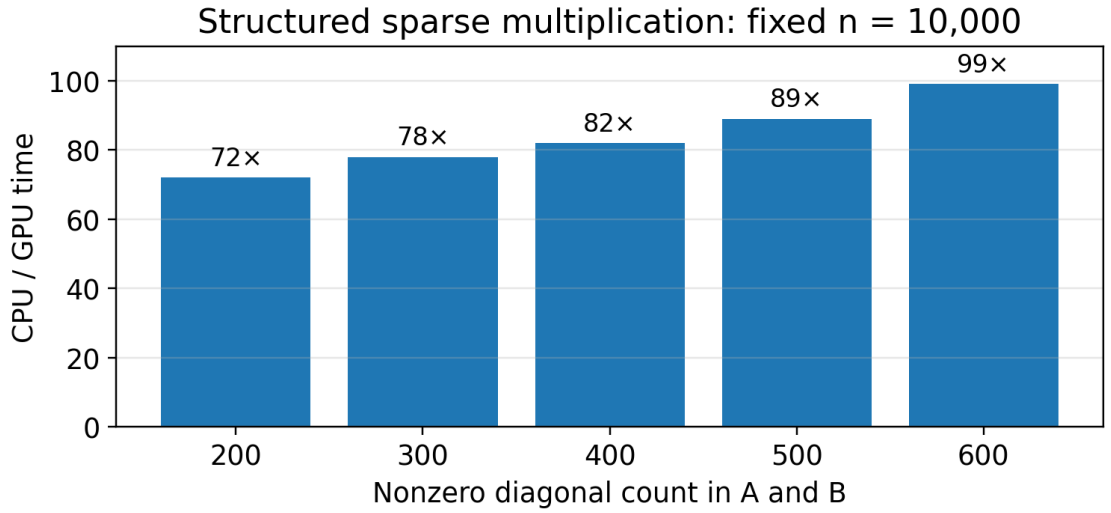
Banded experiments used n = 1024 to 9216 and bandwidths 5, 10, 15, 20; the fixed-size study varied band size at n = 10,240.

Takeaway: the larger the structured diagonal workload, the more the GPU kernel amortizes overhead and exploits parallelism.

Numerical evidence II: structured sparse matrices

The HM GPU kernel gives large speedups on arbitrarily distributed nonzero diagonals.





dgA,nzA	dgB,nzB	nzC	CPU ms	CUDA ms	Speedup
200, 1.75M	200, 1.73M	69.69M	4812.40	66.62	72×
300, 2.57M	300, 2.60M	76.32M	11031.72	141.86	78×
400, 3.46M	200, 3.50M	75.46M	19733.96	241.23	82×
500, 4.36M	500, 4.37M	75.50M	31817.91	358.39	89×
600, 5.24M	600, 5.23M	75.28M	46549.84	468.83	99×

Top reported speedup: 99x for n = 10,000 with 600 nonzero diagonals in both A and B.

Why diagonal structure helps

The speedups reflect several reinforcing effects rather than one isolated trick.

1 Arithmetic follows structure

Only compatible diagonal pairs are visited; work is naturally partitioned by diagonals.

2 Memory access is regular

Contiguous diagonal vectors reduce pointer chasing and preserve stride-1 traversal.

3 Transpose is cheap

Transpose access is obtained by switching diagonal orientation instead of moving O(n^2) data.

4 GPU maps work to tiles

CDM provides T-by-T blocks for banded matrices; HM assigns nonzero-driven work for structured sparse matrices.

HM/CDM/CSD → diagonal BLAS kernels → auto−tuned CPU/GPU libraries

The more the application preserves diagonal structure end-to-end, the more the data structure can pay off.

Promising research directions

The diagonal framework opens implementation and theory questions.

Diagonal BLAS-like kernels

Formalize HM/CDM/CSD operations for matrix-vector, matrix-matrix, and matrix-transpose kernels.

Auto-tuned CPU/GPU selection

Choose HM, CDM, CSD, CSR, CSC, or hybrid kernels based on diagonal density and bandwidth.

General sparse diagonalization

Extend compressed sparse diagonal ideas to general SpGEMM and graph kernels.

Applications

PDE discretizations, graph analytics, matrix-transpose products, and banded/symmetric updates.

Performance modeling

Predict memory traffic, atomic-update contention, and tile occupancy under diagonal sparsity.

Software integration

Bridge with existing BLAS, CUDA, and sparse matrix libraries.

Closing message

Diagonally structured linear algebra is a storage idea and an algorithmic scheduling idea.

$$D_k(A) = \{a_{ij} \mid j - i = k\}, \qquad p = \min(i, j) \longrightarrow C_k \leftarrow C_k + \sum_i A_i \odot B_{k-i}$$

Fundamentals

HM and CDM make diagonals first-class objects.

Acceleration

CPU and GPU implementations exploit locality and independent diagonal work.

Evidence

Up to two orders of magnitude GPU speedups.

Questions?

For more information

1. S. Hossain and M. S. Mahmud, “On Computing with Diagonally Structured Matrices,” HPEC 2019.

2. N. Aimaiti, S. Hossain, and M. S. Mahmud, “Computational Experience with Diagonally Structured Linear Algebra in Java,” HP3C 2020.

3. N. Choudhury, S. A. Haque, and S. Hossain, “Matrix Multiplication with Diagonals: Structured Sparse Matrices and Beyond,” HP3C 2023.

4. S. A. Haque, M. T. Parvez, and S. Hossain, “GPU Algorithms for Structured Sparse Matrix Multiplication with Diagonal Storage Schemes,” Algorithms 2024, 17(1), 31.

5. S. A. Haque, M. T. Parvez, and S. Hossain, “Multiplication of Sparse Matrices and their Transpose using Compressed Sparse Diagonals,” HPEC 2024.