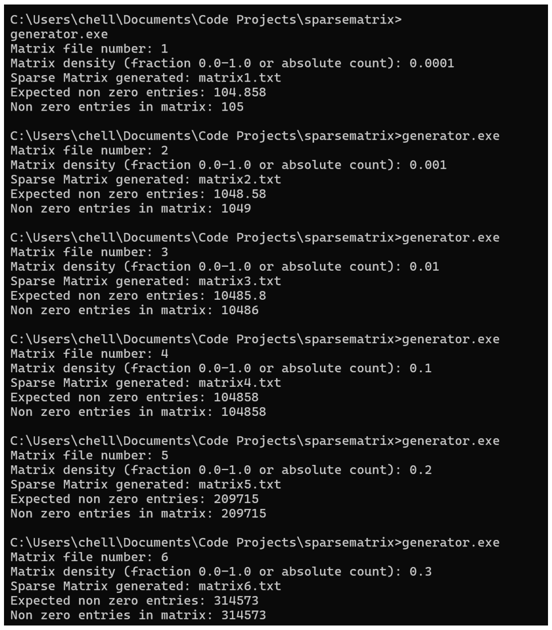
Sparse Matrix Accelerator & Systolic Array Simulation Framework

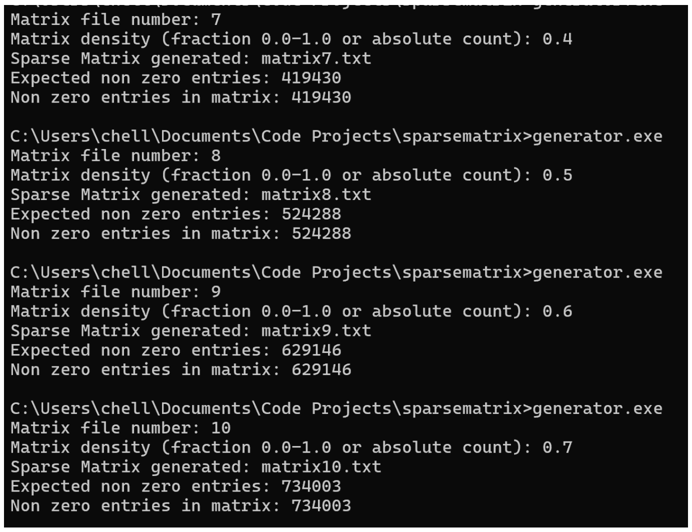
Contents:

Generator file

This program generates a random sparse matrix by treating each matrix entry as a Bernoulli trial within a binomial sampling process. Given a target density or nonzero count, it uses probabilistic selection to choose exactly that many positions without duplicates. The result is a binary matrix where the selected entries are set to 1, and all others remain 0. The output is written to a file, ensuring the generated matrix has the requested sparsity level while mimicking binomial distribution behavior in its placement of nonzeros.



```
C:\Users\chell\Documents\Code Projects\sparsematrix>
generator.exe
Matrix file number: 1
Matrix density (fraction 0.0-1.0 or absolute count): 0.0001
Sparse Matrix generated: matrix1.txt
Expected non zero entries: 104.858
Non zero entries in matrix: 105

C:\Users\chell\Documents\Code Projects\sparsematrix>generator.exe
Matrix file number: 2
Matrix density (fraction 0.0-1.0 or absolute count): 0.001
Sparse Matrix generated: matrix2.txt
Expected non zero entries: 1048.58
Non zero entries in matrix: 1049

C:\Users\chell\Documents\Code Projects\sparsematrix>generator.exe
Matrix file number: 3
Matrix density (fraction 0.0-1.0 or absolute count): 0.01
Sparse Matrix generated: matrix3.txt
Expected non zero entries: 10485.8
Non zero entries in matrix: 10486

C:\Users\chell\Documents\Code Projects\sparsematrix>generator.exe
Matrix file number: 4
Matrix density (fraction 0.0-1.0 or absolute count): 0.1
Sparse Matrix generated: matrix4.txt
Expected non zero entries: 104858
Non zero entries in matrix: 104858

C:\Users\chell\Documents\Code Projects\sparsematrix>generator.exe
Matrix file number: 5
Matrix density (fraction 0.0-1.0 or absolute count): 0.2
Sparse Matrix generated: matrix5.txt
Expected non zero entries: 209715
Non zero entries in matrix: 209715

C:\Users\chell\Documents\Code Projects\sparsematrix>generator.exe
Matrix file number: 6
Matrix density (fraction 0.0-1.0 or absolute count): 0.3
Sparse Matrix generated: matrix6.txt
Expected non zero entries: 314573
Non zero entries in matrix: 314573
```



```
Matrix file number: 7
Matrix density (fraction 0.0-1.0 or absolute count): 0.4
Sparse Matrix generated: matrix7.txt
Expected non zero entries: 419430
Non zero entries in matrix: 419430

C:\Users\chell\Documents\Code Projects\sparsematrix>generator.exe
Matrix file number: 8
Matrix density (fraction 0.0-1.0 or absolute count): 0.5
Sparse Matrix generated: matrix8.txt
Expected non zero entries: 524288
Non zero entries in matrix: 524288

C:\Users\chell\Documents\Code Projects\sparsematrix>generator.exe
Matrix file number: 9
Matrix density (fraction 0.0-1.0 or absolute count): 0.6
Sparse Matrix generated: matrix9.txt
Expected non zero entries: 629146
Non zero entries in matrix: 629146

C:\Users\chell\Documents\Code Projects\sparsematrix>generator.exe
Matrix file number: 10
Matrix density (fraction 0.0-1.0 or absolute count): 0.7
Sparse Matrix generated: matrix10.txt
Expected non zero entries: 734003
Non zero entries in matrix: 734003
```

Dataset
Dataset1
Matrix 1 to10
The generator generates 1024x1024 matrices with the required density of 0.0001, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7. Each non zero value is to takes the value 1.

Dataset2
<u>NOTE:</u>
The actual density of these matrices may not be captured as the matrix used in the dataset2 are only a subset cropped to 1024 x 1024. Due to device limitations the matrices were cropped out rather than remastered to fit 1024x1024.

matrix 11 - seems to be more dense on the left and weathers out towards the right, but the matrix 11 used here is a subset cropped to 1024x 1024

matrix 12 - is the polar opposite of matrix 11, it is more dense as it moves towards the right.

matrix 13 - this has a blotched square dense pattern across. The subset used as the dataset happens to be one of the dense squares itself.

matrix 14 - it has most of its nnz elements across the diagonal and the top row. The subset used seems dense as it covers the row and diagonal parts.

matrix 15 - it is sparsed with a pattern observed as we move more towards the middle. The subset used covers the top left thereby it is almost fully sparse.

matrix 16 - it appears symmetric with more density towards the middle. The selected subset covers the top left cropped out value.

matrix 17 - it is highly dense for the subset cropped out.

matrix 18 - it has a uniform pattern across. Almost like multiple diagonal lines across the matrices. The subset has captured one of the block that has the diagonal pattern.

matrix 19 - very similar to 18 but the matrix has a smaller square pattern and a larger sqaured connected to the rigth bottom corner of it wit the same multiple diagonal pattern. The subset used is from the small square matrix.

matrix 20 - it has a diagonal sparse matrix, so to mimic that matrix 20 was cropped and resized to keep its character.

Folder Structure
The code organizes outputs into a folder hierarchy under data <fileNum>. The raw folder stores the original tile (64x64) files (tile*.txt) extracted from the input matrix. The compressions folder contains the compressed representations, including per-tile compressed files (compressed*.txt) and intermediate tile-level compression logs (tile_comp*.txt). The decompressions folder holds all decompressed results, including per-tile outputs (decompressed*.txt, tile_decomp*.txt) as well as a cycle log (log_decompression_cycle.txt) that records the latency of decompression.

Main
Runs only compression and decompression functions for a chosen format.

CSR_MAIN / BCSR_MAIN / LIL_MAIN / COO_MAIN

Perform full end-to-end execution: tiling, compression, decompression, verification, and output data collection for all tiles.

Terminal execution

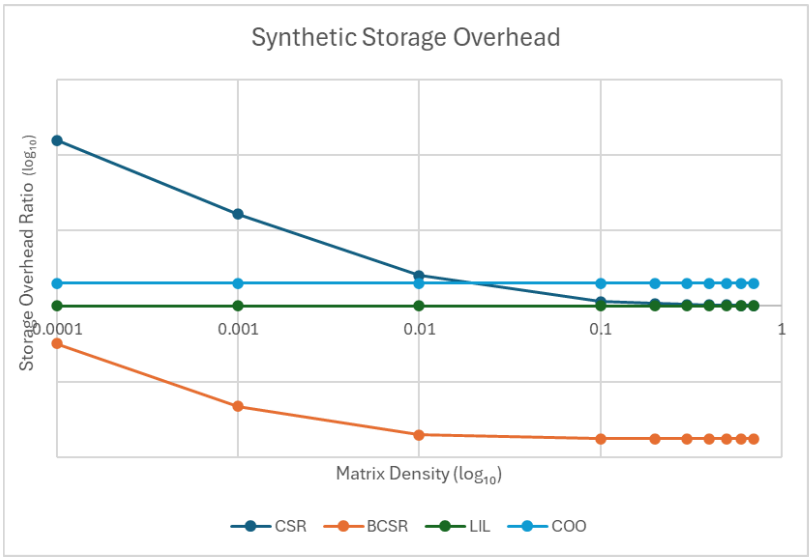
```
C/C++

g++ format_main.exe
Kindly wait a minute or two for its execution.
Input: enter the matrix dataset file number
```

Functionality:
All outputs perform a functionality check file by file to ensure compression and decompression reconstruct the expected tile and matrix. And all program execution is bound to the mathematical time cycle formula computed.

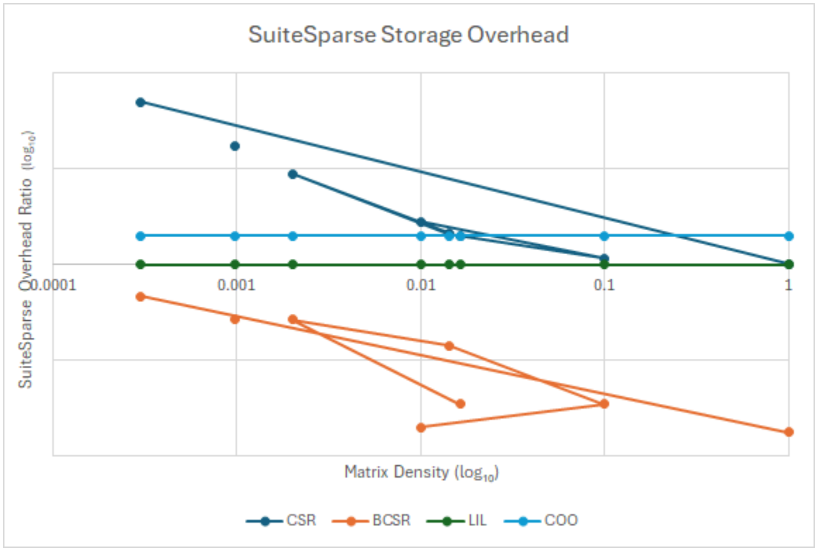
OUTPUT FIGURES:

Storage Overhead

Synthetic



SuiteSparse



Graph Elements

X-axis: Matrix Density ($\log_{10}$ scale)
Y-axis: Metadata Ratio ($\log_{10}$ scale)
Formats Compared: CSR, BCSR (block size 8), LIL, COO
Matrix Density = (number of non zero element)/(1024 x 1024)

CSR (Compressed Sparse Row):
Metadata = row pointers + column indices
At very low densities, CSR has very high storage overhead (about 157× at density 0.0001) because of the large row pointer array relative to the few nonzeros. As density increases, the ratio drops sharply and stabilizes around ~1 for dense matrices.
CSR requires storing row pointers, so at high sparsity this dominates, but when the matrix is denser the overhead is amortized.

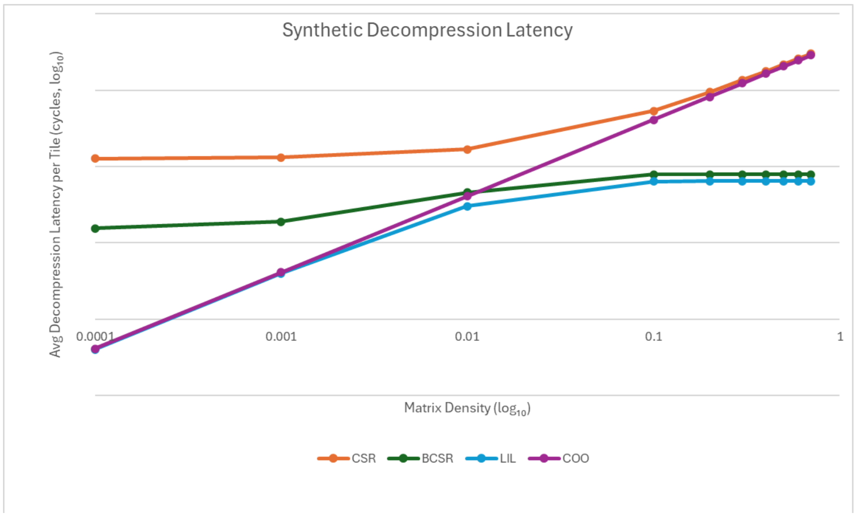
BCSR (Block CSR):
Metadata = block column indices + block row pointers.
BCSR maintains consistently low overhead across all densities (below ~0.32).
With block size = 8, storage is aligned to blocks rather than individual nonzeros, which minimizes indexing overhead. Once block utilization increases, the format is very efficient.

LIL (List of Lists):
Metadata = per-row linked structure
LIL shows a constant overhead ratio of 1 across all densities.
LIL is implemented row-wise with fixed structures for storing nonzeros, so its storage overhead is stable and independent of density.

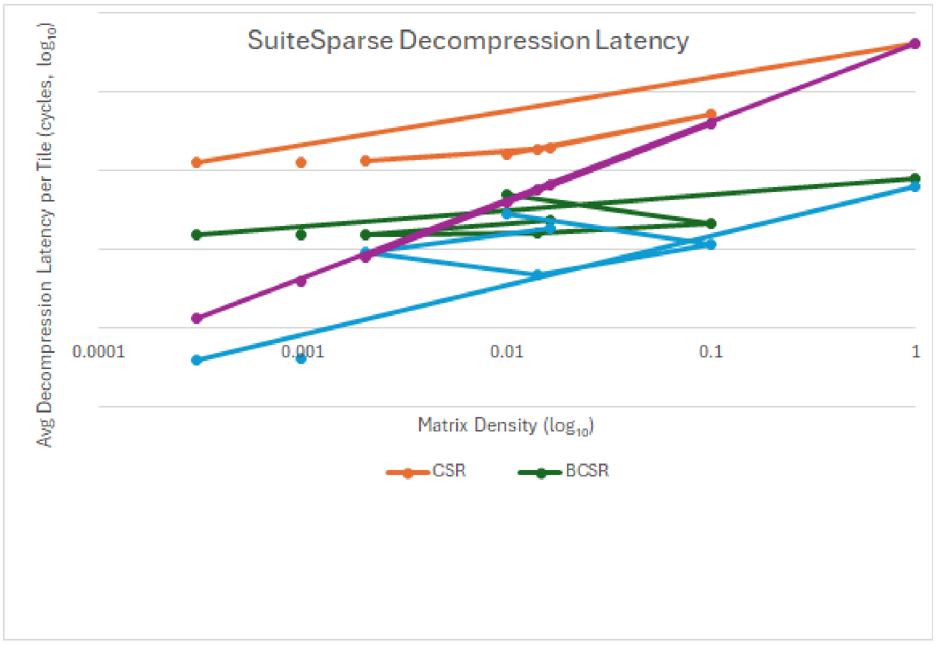
COO (Coordinate List):
Metadata = row indices + column indices for every nonzero.
COO maintains a fixed overhead ratio of 2 at all densities.
COO stores explicit row and column indices for each nonzero, so overhead depends only on the number of nonzeros (nnz), not the density distribution.

Latency Time

Synthetic



SuiteSparse



Graph Elements

X-axis: Matrix Density ($\log_{10}$ scale)
Y-axis: Average Decompression Latency per Tile (cycles, $\log_{10}$ scale)
Formats Compared: CSR, BCSR (block size 8), LIL, COO
Matrix Density = (number of non zero element)/(1024 x 1024)

CSR
Formula: cycles = 2*TILE + nnz - 1 = 127 + nnz for TILE = 64. (nnz= number of non zero elements.)
Behavior: a fixed floor of about 127 cycles per tile at very low density, then linear growth with nnz. Values rise from ~127 (density ≈ 1e-4) to ~2994 (density ≈ 0.7).

BCSR
Formula: cycles = number of non-zero blocks + 2* number of block rows - 1.
Behavior: increases as more blocks become nonzero, then plateaus once most block rows are active. In these runs it levels near ~79 cycles from density ≈ 0.2 onward, because the 2* number of block rows - 1 dominates.

LIL
Formula: cycles = nz
Behavior: grows with the count of rows that contain at least one nonzero, then saturates at 64 once all rows are active. This makes LIL the fastest at moderate to high densities here.

COO
Formula: cycles = nnz.
Behavior: strictly linear in nnz with no fixed overhead. Best at ultra-sparse regimes, then becomes the slowest as density increases.

FORMAT RECOMMENDATION

For **extremely sparse matrices,** COO is the best choice because it only stores explicit (row, column, value) tuples for nonzeros, so both storage and latency scale directly with nnz. Formats like BCSR or LIL waste space by storing entire blocks or padding when there are only a few scattered entries.

For **moderately dense matrices**, BCSR is more effective since its block-based grouping amortizes metadata over 64 values and reconstructs one block per cycle. This yields stable latency that saturates quickly and storage overhead that becomes negligible as density increases, making it more efficient than COO or LIL when many entries are present in most regions.

CSR

The CSR implementation uses local BRAM buffers to first load a 64×64 dense tile (A_BRAM) and then build CSR arrays in BRAM (val_BRAM, col_BRAM, offset_BRAM). Each row is scanned once to append nonzeros and their column indices, while offset_BRAM[i] stores the running count of nonzeros up to the end of row i. Decompression streams these arrays back into a zero-initialized output tile by walking the offsets row by row and writing A[i][col[j]] = val[j]. The modeled latency per tile is two row passes plus one cycle per nonzero:
cycles ≈ rows + rows + nnz = 64 + 64 + nnz = 128 + nnz.

For very sparse matrices, latency stays close to the fixed base cost since there are almost no nonzeros, but storage overhead is high because the row offsets outweigh the few values stored. For moderately dense matrices, latency rises steadily with the number of nonzeros, while storage becomes efficient as the fixed offset cost is spread across many entries, bringing the metadata-to-data ratio close to one.

This format is inefficient at extremely low densities due to heavy offset overhead, but becomes highly effective at moderate densities where both storage and latency scale predictably and efficiently.

Code Logic:

Buffering and burst to BRAM

- copy each 64×64 tile A_tile into on-chip BRAM arrays: A_BRAM, offset_BRAM, val_BRAM, col_BRAM
- local low-latency access and contiguous scan, fewer external reads, easy rowwise accumulation

csr format path (csr_format)

- inputs: A[MAX_ROWS][MAX_COLS], outputs: offset[MAX_ROWS], col[MAX_NNZ], val[MAX_NNZ], *size
- burst read A → A_BRAM (row major copy)
- for each row i
    - walk columns j, read value = A_BRAM[i][j]
    - if nonzero, append into BRAM fifo arrays: val_BRAM[nnz + row_nnz] = value, col_BRAM[nnz + row_nnz] = j
    - track row_nnz for this row
    - set offset_BRAM[i] = nnz + row_nnz

    - advance nnz += row_nnz

- Burst write BRAM → output arrays
    - Copy first nnz entries into val[] and col[]
    - Copy all row offsets into offset[]
- Set *size = nnz
- Invariant check in main: size == offset[TILE-1]

Decompression path

- Clear A_tile_decomp to zeros in main
- sSate variables: load, diff_flag, loop, diff, i, j, current_row, B0
- Rebuild rows using CSR streams offset[], col[], val[]

Per-cycle behavior (load, diff, offload)

- Total cycles in main: cycles_tile = TILE + TILE + size - 1
    - first TILE cycles to load row offsets
    - next TILE cycles for control/row stepping
    - plus size cycles for writing all nnz

- in csr_decompression loop:
    - load phase (load == 1):
        - pick next row i = current_row, fetch B0 = offset[i], current_row++
        - set load = 0, diff_flag = 1
    - diff phase (diff_flag == 1):
        - compute how many nnz to emit for row i
        - if i == 0: diff = B0
          else diff = B0 - offset[i-1]
        - if diff > 0: set loop = 1, diff_flag = 0
          else no nnz for this row, set load = 1, diff_flag = 0

    - offload phase (loop == 1):
        - while diff > 0:
            - increment global nnz index j++
            - write back one element A[i][col[j]] = val[j]
            - decrement diff
        - when diff == 0: set loop = 0, load = 1
- Termination when rows exhausted and no pending offload

Pipelining choices

- Loops touch shared BRAM arrays (A_BRAM, val_BRAM, col_BRAM, offset_BRAM)
- Concurrent reads and writes to the same banks create hazards
- The commented #pragma HLS PIPELINE II=1 marks intent, but banking limits can force sequential behavior in practice

Cycle model

- in CSR you model cycles per tile as 64 + 64 + nnz - 1
    - offsets streaming and row control plus all nnz writebacks
    - matches the state machine: load rows, compute diffs, offload writes

Global tiling

- Global 1024×1024, tile 64×64, grid 16×16
- Slice tile, dump raw tile
- Run csr_format
- Write per-tile compressed dump and per-tile decompressed tile
- Accumulate cycles and log: cycles, nnz, tile id
- Reconstruct A_big_decomp, verify against original with fabs(a-b) > 1e-6
- Write full decompressed matrix and summary (metadata/data ratio and cycle stats)

Why BRAM

- On-chip storage for deterministic latency
- Row-major scans are fast from BRAM
- Random writes during decompression hit on-chip array, not external memory

Performance modeling hooks

- Throughput tied to nnz emission rate
- Efficiency limited by BRAM banking and shared access during format and offload

Edge conditions

- MAX_NNZ bounds per-tile storage
- Offsets are exclusive-prefix counts, so offset[TILE-1] == nnz

BSCR

The BCSR implementation partitions a 64×64 tile into 8×8 blocks and copies the dense tile into BRAM. The formatter scans each 8×8 region; if any entry in a region is nonzero, it records one block column index and stores the entire 8×8 block's 64 values contiguously in BRAM. Row pointers accumulate the count of blocks per block-row. The decompressor loads three BRAM arrays: B0 for row pointers, B1 for block columns, and BVAL for the 64 values per block. Reconstruction proceeds block-row by block-row; for each pointer difference, one block is emitted per cycle by writing the 8×8 values back to the correct location.

For very sparse matrices, BCSR wastes space because even a single nonzero forces storage of an entire 8×8 block, so padding dominates. Latency is low at first but rises with the number of active blocks. For moderately dense matrices, most blocks contain data, so storage becomes efficient as block metadata is shared across many values, and latency stabilizes at a predictable level once all blocks are active.

BCSR is inefficient when nonzeros are scattered, but it is highly effective for moderate densities where values cluster, giving both stable latency and compact storage.

Code Logic:

Buffering and burst to bram

- copy 64×64 tile into on-chip BRAM: A_BRAM, plus BRAM staging for row_BRAM, col_BRAM, val_BRAM
- local, contiguous scans for blocks and fast random writes during block dump

BCSR format path

- Block_size=8, values_per_block=64, BR=BC=MAX_ROWS/block_size=8
- Burst read A → A_BRAM
- for each output block-row out_row in [0, BR)
    - row_start_blocks = *size_blocks
    - for each output block-col out_col in [0, BC)
        - scan 8×8 window; if any nnz then mark has_nnz = true
        - when has_nnz
            - write block column index as starting column: col_BRAM[*size_blocks] = out_col * block_size
            - write full 8×8 payload to val_BRAM[base + t] (store all 64 values)
            - (*size_blocks)++
    - set row_BRAM[out_row + 1] = *size_blocks

- Burst write BRAM → outputs: copy col_BRAM, copy (*size_blocks)*64 vals, copy row_ptr [0..BR]
- Block-level padding. any nonzero inside an 8×8 triggers storing all 64 entries for that block

decompression path

- Inputs: row_ptr + 1 (so decompressor sees B0 starting at index 0), col, val, size_blocks
- On-chip buffers: B0[BR] row offsets, B1[MAX_BLOCKS] block columns, BVAL[values_per_block][MAX_BLOCKS] payloads
- Load B0, B1, and block payloads into BRAM buffers
- Clear A_out
- block-rows br=0..BR-1
    - read off = B0[br], compute diff = off - prev_off, update prev_off = off
    - while diff > 0
        - get block column bc_raw = B1[blk_ptr], map to block index bc
        - place 8×8 payload BVAL[blk_ptr] at (r0=br*8, c0=bc*8)
        - blk_ptr++, diff--

Per-cycle behavior

- One block reconstructed per cycle in the offload loop
- Total cycles per tile in main: cycles_tile = size_blocks + 2*BR - 1
    - BR to stream row offsets (B0)
    - BR for row control/compute diffs
    - size_blocks for offloading each 8×8 block
- Row_ptr passed as row_ptr + 1, so B0 aligns with block-row 0

Pipelining choices

- Shared BRAM for A_BRAM, B0, B1, BVAL creates common-memory hazards
- Scanning blocks and writing 64 values per found block hit the same banks
- Full II=1 across nested loops is not safe everywhere; sequentialization avoids stalls

Cycle model

- nnz_blocks + 2*BR - 1 from code and logged to flog
- Throughput tied to block count, not element count; dense-inside-block padding reduces compression but keeps deterministic cycles

Global tiling

- 1024×1024 global, 64×64 tiles, 16×16 grid

- for each tile: dump raw, run bcsr_format, update accounting
    - metadata per tile = size_blocks + BR (block cols + row_ptr entries except the first passed by +1)
    - data per tile = size_blocks * values_per_block
- Write compressed dump, run decompress_bcsr, log cycles, dump decompressed tile
- Reconstruct A_big_decomp, verify with fabs(a-b) > 1e-6, write full output and summary

Why BRAM

- local row_ptr, col, and block payloads allow steady streams
- block placement performs 64 contiguous writes from BRAM without external latency
- enables simple control: load, diff, offload

Padding

- Block detection is any-nnz; storage writes the full 8×8 payload
- Zeros inside a stored block are kept as zeros in val_BRAM
- Trade-off: more data than CSR on very sparse blocks, but regular write pattern and fixed-size moves

Performance modeling hooks

- Roofline view: arithmetic intensity is low; bandwidth to move blocks dominates
- Efficiency depends on block sparsity and BRAM banking for the 64-value writes
- Architectural trade-off: fewer indices and regular placement vs extra payload from block padding

Edge conditions

- Size_blocks capped by MAX_BLOCKS; nnz_blocks is clamped before loads
- bc_raw is interpreted as column start

COO

The COO implementation uses local BRAM buffers to first load the dense matrix and then build a COO tuple array of (row, col, val) triples stored entirely in BRAM. The decompression stage iterates through these tuples one per cycle, writing values back to the output matrix. Latency is proportional to the number of nonzeros (nnz), since decompression processes one tuple per cycle.

For very sparse matrices, COO is efficient because only the nonzeros are stored, so latency stays very low and storage is far smaller than the dense form, even though metadata per value is relatively heavy. For moderately dense matrices, latency grows in step with the number of nonzeros, and storage eventually becomes worse than dense once a third of the entries are filled. COO works best for highly sparse data where storage and speed scale with actual nonzeros, but it loses efficiency as matrices become moderately dense.

Code Logic:

Buffering and burst to bram

- Copy each 64×64 tile A_tile into on-chip BRAM (A_BRAM) in a burst
- Local low-latency access, fewer external reads, contiguous block for linear scan

Coo format path

- Inputs: A[MAX_ROWS][MAX_COLS] (64×64), outputs: tuple_A[MAX_NNZ][3], *size
- Burst read A → A_BRAM
- Nested scan i,j
  v = A_BRAM[i][j]
  if v != 0.0f push tuple (row=i, col=j, val=(int)v) into tuple_A_BRAM
   if (nnz >= MAX_NNZ) break
- Set *size = nnz
- Burst write tuples from BRAM to tuple_A
- Value to int in tuples, decompressed values are integer-valued floats

Decompression path (coo_decompression)

- Clear A_tile_decomp to zeros
- for i in [0, *size) read tuple (row, col, val) and write A[row][col] = val
- *size is the nnz for this tile

Pipelining choices

- Loops touch shared BRAM arrays
- Concurrent reads and writes to the same banks create hazards
- Loop cannot keep II=1 in these loops because of common memory access
- Some loops remain sequential to avoid stalls

Cycle model

- Reconstruction cycles per tile = nnz for that tile
- cycles_tile = size and sum across tiles
- Average cycles per tile = total cycles / tile_count
- Aligns with one tuple update per cycle model

Global tiling

- Global 1024×1024, tile 64×64, grid 16×16
- Slice tile, write raw tile dump
- Run coo_format, track total_meta_elems += 2*size and total_data_elems += size
- Append tile tuples to compressed files
- Run coo_decompression, dump decompressed tile
- Copy back into full A_big_decomp
- Full-matrix check with fabs(a-b) > 1e-6
- Write decompressed full matrix
- Write summary with metadata/data ratio and cycle stats

Why BRAM

- On-chip storage reduces latency. Supports burst load and burst write of tuples
- Predictable access for scan and for random writes during decompression

Performance modeling hooks

- Throughput tied to nnz because each tuple is one update
- Efficiency limited by shared memory banking on BRAM
- Connects to pipeline analysis, roofline thinking, and sparse-matrix decompression trade-offs

Edge conditions

- MAX_NNZ = 4096 caps tuples per tile. Integer cast in tuples quantizes values
- Verification tolerance 1e-6 guards floating read/write formatting

LIL

The LIL implementation first builds two lists for every column: one list of row indices and one list of corresponding values. Nonzeros are pushed into these lists, while zeros are skipped. During decompression, the design stages this data into BRAM, with the upper half storing row indices and the lower half storing values. Each column is padded with a sentinel index of −1 and value 0 to provide uniform memory layout and easy termination conditions.

For very sparse matrices, LIL is efficient because only the rows that actually contain nonzeros are processed, so latency stays very low and storage remains predictable with one index per value. As density increases, more rows become active and latency grows until all rows are touched, after which it levels off. Storage remains consistent across densities since the metadata per value is fixed.

LIL offers stable storage costs and efficient latency scaling, making it well suited for sparse data and still reliable when matrices become moderately dense.

Code Logic:

Buffering and burst to bram

- Copy a 64×64 tile into vectors, then pack into on-chip BRAM-like BRAM buffer
- Layout in BRAM: top SIZE rows are row indices, bottom SIZE rows are values, column-major

lil format path

- Scan column j = 0..SIZE-1
- For each row i, if A[i][j] != 0 push i into row_out[j] and A[i][j] into val_out[j]
- Result: per-column lists of (row,value) with variable length

Decompression path

- Clear A
- Allocate BRAM as 2*SIZE × SIZE
- Load compressed columns into BRAM with padding:
    - indices plane BRAM[k][j] = row_in[j][k] or -1 if padded
    - values plane BRAM[k+SIZE][j] = val_in[j][k] or 0 if padded
- Build nz_rows: first time a real value appears for row r, mark seen[r]=1 and record

- Cycles = nz_rows.size()
- For each cycle t, take target row r_target = nz_rows[t] and fill it by scanning all columns: find r_target in that column's index plane and write the value from the values plane

BRAMlayout and padding

- 2 planes per column: indices then values
- Fixed height SIZE per column allows uniform burst loads and aligned accesses
- Padding uses -1 for "no row" and 0 for "no value" so loops can break early per column

Per-cycle behavior

- One "cycle" reconstructs one full dense row
- Each cycle scans all columns, at most SIZE index probes per column, stops early on -1

Cycle model

- Total cycles per tile returned by lil_decompress as cycles = number of distinct non-zero rows
- Main logs cycles_tile and accumulates throughput stats across tiles

Pipelining choices

- Loading into BRAM is regular and can be burst; inner loops over k are uniform due to padding
- Row reconstruction touches all columns and probes a shared BRAM column each time; common memory and variable early-exit points limit safe II=1 pipelining across nested loops
- Keeping it sequential avoids bank conflicts on the two planes for the same column

Global tiling and accounting

- 1024×1024 global, tiles of SIZE×SIZE (64×64), 16×16 grid

- Per tile: dump raw, lil_format, write padded "rows/values" per column
- Metadata per nz = one row index; data per nz = one value
  total_meta_elems += nnz_tile, total_data_elems += nnz_tile
- lil_decompress, log cycles, dump decomp tile, stitch A_big_decomp, verify equality

Why bram

- 2×SIZE column slabs enable simple burst load/store

- pattern stays on-chip; no off-chip latency while scanning padded slots
- planes let index checks and value reads be co-located per column

Performance modeling hooks

- Roofline: intensity low; bytes moved for padded columns dominate
- Efficiency scales with nz-row count; more distinct nz rows increase cycles
- Trade-offs: variable-length lists stored simply vs fixed padded column height for regular memory traffic and simpler control