

Project A2: Dense vs Sparse Matrix Multiplication (SIMD + Multithreading)

ECSE 4320 Advanced Computer Systems

Jaithra Pagadala

1 Setup and Methodology

Machine. 12th Gen Intel(R) Core(TM) i7-1260P (x86_64), 16 logical CPUs, 12 cores, SMT enabled.

Compiler and ISA. GCC with `-O3 -march=native -std=c++17 -fopenmp`. SIMD paths use AVX2 FMA when supported by the CPU and enabled by `-march=native`. Scalar baselines use the same loop order without vector intrinsics.

Variability controls. The run script pins threads to a fixed CPU set via `taskset`. OpenMP uses `OMP_PLACES=cores` and `OMP_PROC_BIND=close`. The run script attempts to set the CPU governor to `performance` (best-effort). Each configuration is repeated ≥ 3 times. Median runtime is used for plots. p50/p95/p99 latency for repeated kernel calls is recorded in the CSV.

Reproducibility. All experiments were made using the following specs: `lscpu`:

- CPU model: 13th Gen Intel Core i9-13905H
- OS: Ubuntu 22.04.6
- Cores: 14
- GCC: 11.04
- Perf : 5.11
- Pinned Frequency: 1.5 GHz

The general layout of the experiment was through:

```
chmod +x run_a2.sh
./run_a2.sh                # runs sweeps, writes results/results_a2.csv,
                           generates plots
python3 plot_a2.py --csv results/results_a2.csv --outdir results
PERF=1 ./run_a2.sh        # enables perf stat on representative cases
```

2 Kernels Implemented

2.1 Dense GEMM (Dense $A \times$ Dense B)

Row-major matrices are used. A tiled GEMM baseline is implemented:

$$C_{m \times n} \leftarrow A_{m \times k} B_{k \times n}.$$

Tiling uses `tileM`, `tileK`, and `tileN`. The SIMD path vectorizes across the n dimension (columns of B and C).

2.2 CSR-SpMM (Sparse $A \times$ Dense B)

Sparse A is stored in CSR: `rowptr`, `colidx`, and `values`. The kernel computes:

$$C \leftarrow A_{\text{CSR}} \cdot B.$$

A column blocking knob `jblock` is used to improve locality in B and C . The SIMD path is enabled for row-major B because loads across j are contiguous.

2.3 Sparsity Patterns

The generator supports:

- **uniform**: nonzeros spread across the full matrix.
- **band**: nonzeros concentrated near a diagonal band.
- **blockdiag**: block-diagonal structure.

3 Experiments, Plots, and Analysis

All figures are generated by `plot_a2.py` and saved in `results/`.

3.1 1) Correctness and baselines

Correctness is validated using a scalar reference for each kernel. Dense GEMM uses a scalar tiled reference. CSR-SpMM uses a scalar CSR reference. Relative error is expected to be small for float (typical tolerance target: $\leq 10^{-3}$). Scalar single-thread runs define the baseline. SIMD single-thread runs isolate vectorization benefit.

3.2 2) SIMD and threading scaling

Figure 1 reports GEMM GFLOP/s vs threads (scalar vs SIMD). Figure 2 reports CSR-SpMM GFLOP/s vs threads (scalar vs SIMD).

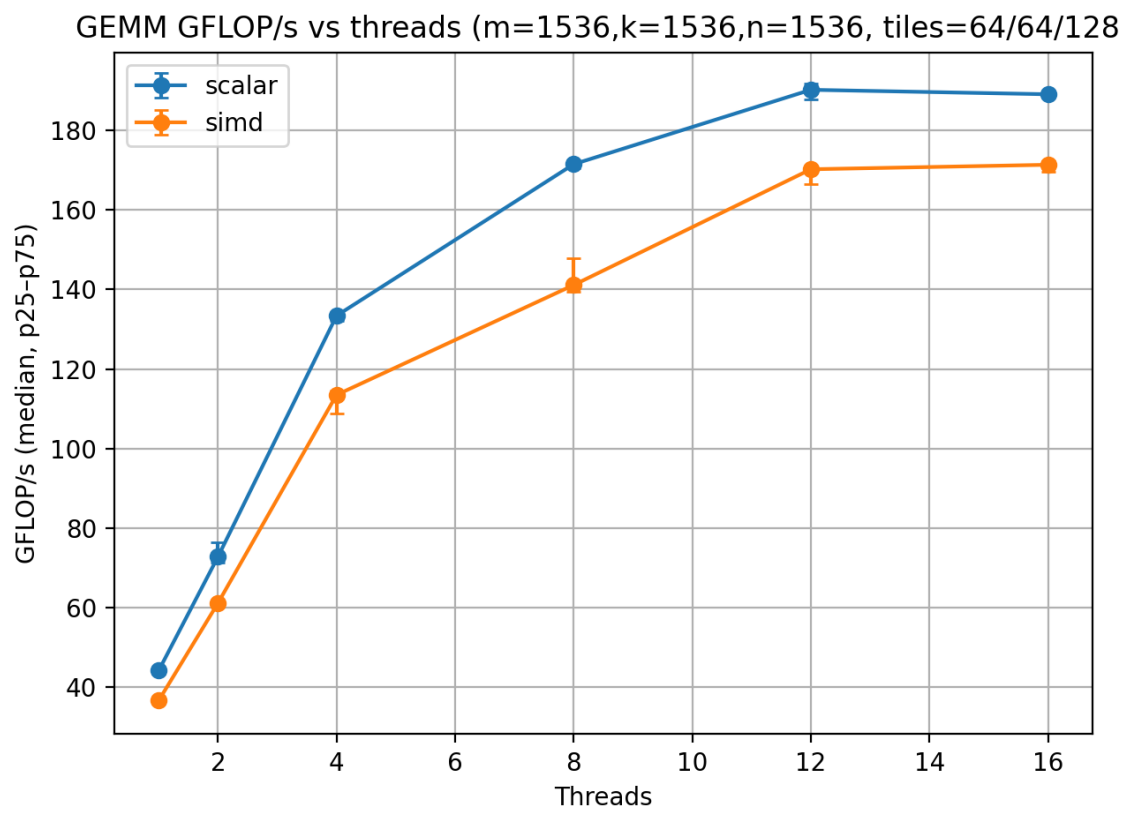


Figure 1: Dense GEMM GFLOP/s vs threads (1536^3).

MM GFLOP/s vs threads (m=2048,k=2048,n=512, density=0.01, jblock=128)

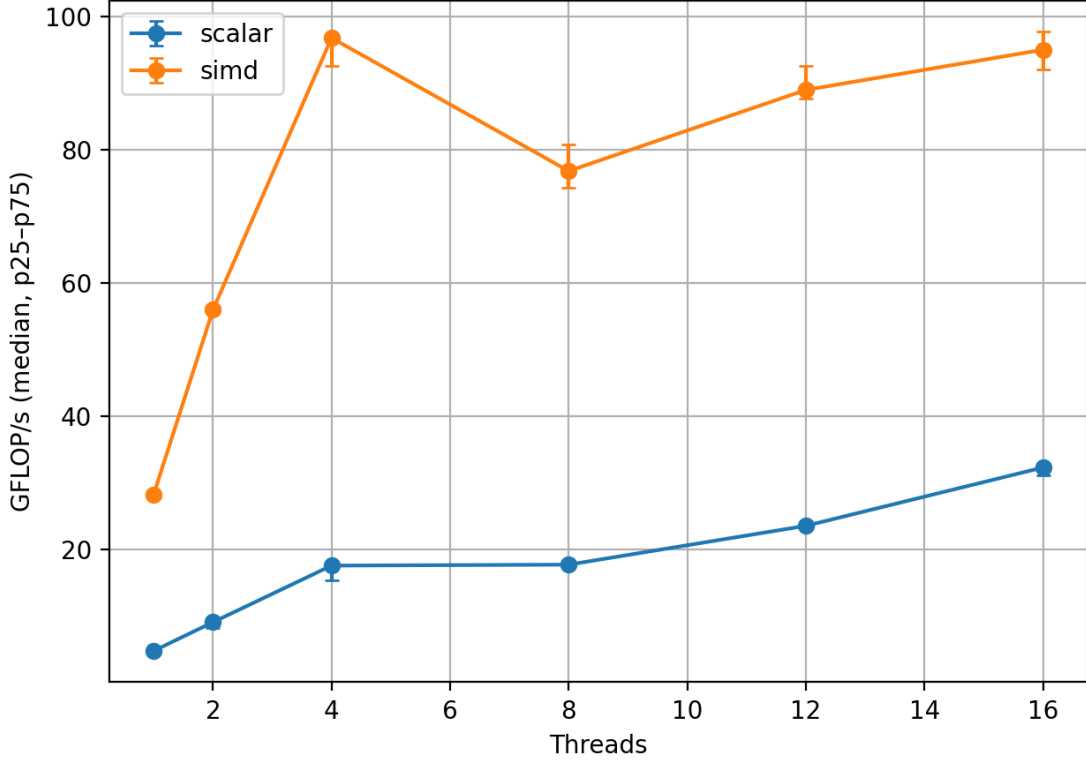


Figure 2: CSR-SpMM GFLOP/s vs threads (m=2048, k=2048, n=512, density=1%, row-major B).

Analysis. Dense GEMM has high arithmetic intensity due to reuse of A and B within tiles. SIMD improves throughput by computing multiple columns per instruction. Thread scaling is typically strong until shared cache and memory bandwidth limits appear. CSR-SpMM has lower arithmetic intensity. Each nonzero multiplies one value from A with a full row segment of B , then accumulates into C . Under uniform sparsity, reuse of a given B row can be limited and memory traffic becomes a stronger limiter. Scaling can saturate earlier than GEMM. SIMD provides benefit mainly when B is row-major and the inner loop over j is contiguous.

3.3 3) Density break-even (uniform random)

Figure 3 compares dense GEMM runtime (fixed cost for the same m, k, n) against CSR-SpMM runtime as density is swept. The break-even density is the smallest density where CSR-SpMM is faster than dense GEMM.

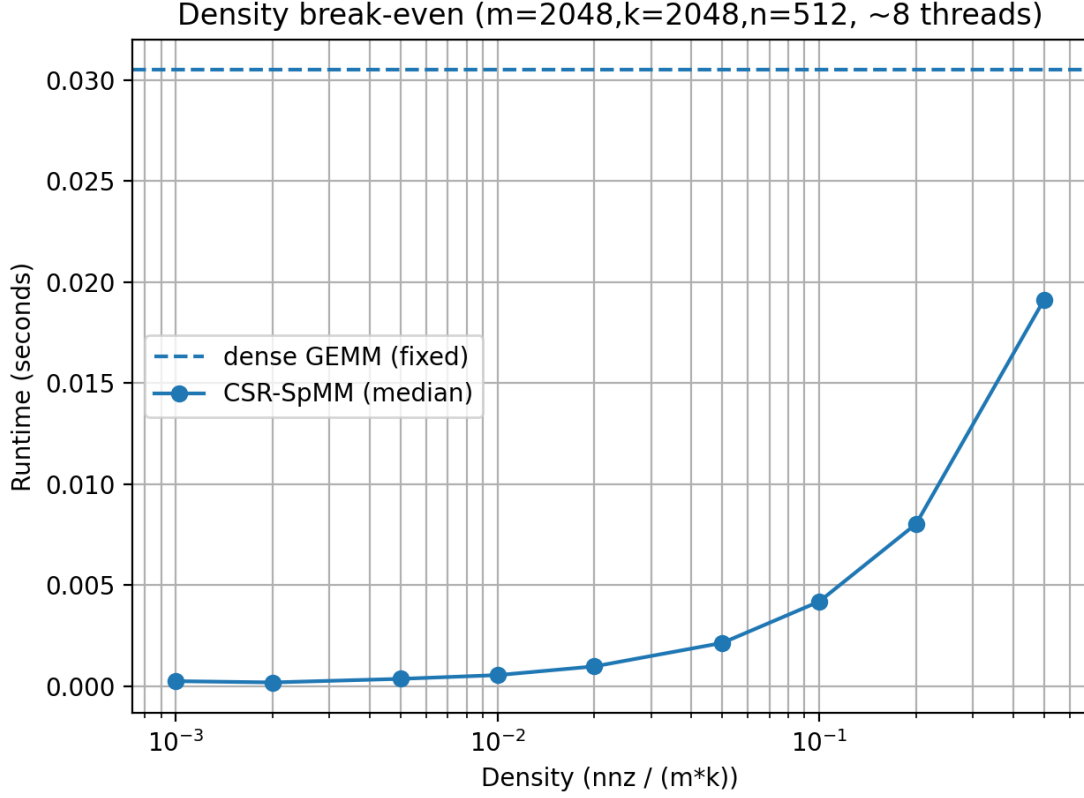


Figure 3: Density sweep and break-even point ($m=2048, k=2048, n=512$, SIMD, ≈ 8 threads).

Analysis. Dense GEMM performs $2mkn$ floating-point operations and amortizes memory traffic through tiling and reuse. CSR-SpMM performs $2 \cdot nnz \cdot n$ operations, so compute cost decreases linearly with density. However, CSR-SpMM introduces overhead from indices and irregular access patterns, plus reduced reuse for uniform randomness. As density increases, CSR-SpMM approaches dense work but keeps CSR overhead, so dense GEMM becomes preferable. As density decreases, sparse wins because the reduced FLOP count dominates. Structured patterns (band/block-diagonal) can improve reuse and shift break-even upward.

3.4 4) Working-set transitions and bandwidth

Figure 4 sweeps problem size to cross L1/L2/LLC/DRAM working sets and reports GFLOP/s. Figure 5 shows estimated bandwidth and includes a STREAM triad bandwidth reference measured in the same binary.

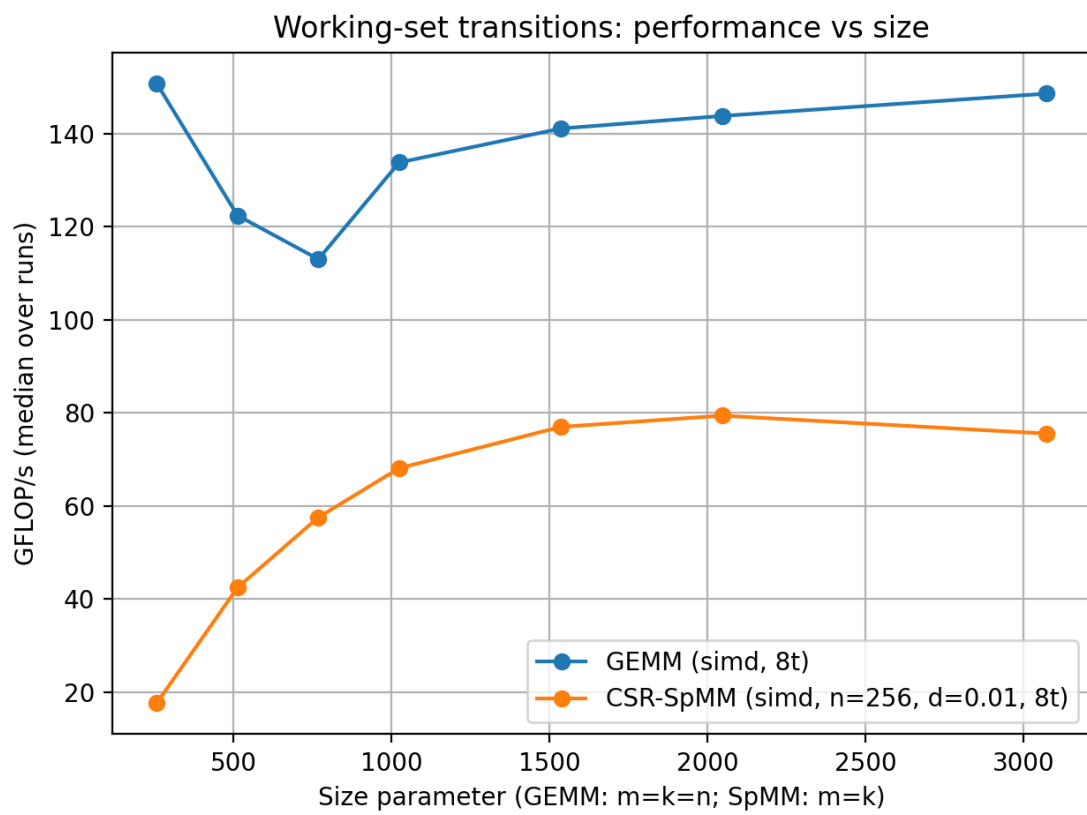


Figure 4: Working-set transitions: GFLOP/s vs size.

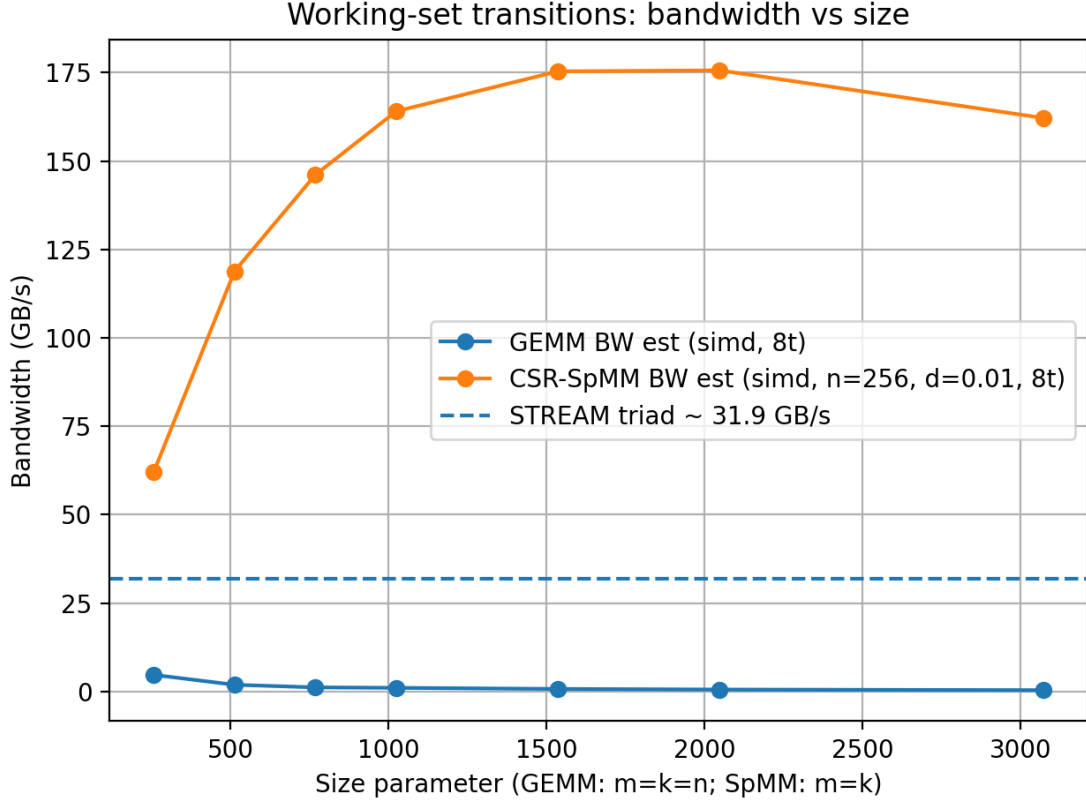


Figure 5: Estimated bandwidth vs size with STREAM triad reference.

Analysis. As sizes grow, cache reuse decreases and time becomes increasingly dominated by memory movement. Dense GEMM can maintain performance longer because tiling increases data reuse inside cache. Eventually, large matrices stress LLC and DRAM and performance drops. CSR-SpMM often becomes bandwidth-sensitive at smaller sizes because the sparse structure reduces spatial locality. Under uniform sparsity, accesses to B can behave like streaming reads with weak reuse, and the kernel trends toward the bandwidth roof earlier.

3.5 5) Roofline interpretation

Figure 6 plots achieved GFLOP/s against a simple roofline model using measured STREAM bandwidth and a peak GFLOP/s placeholder. Arithmetic intensity (AI) is computed from FLOP count divided by an estimated byte model written in the CSV.

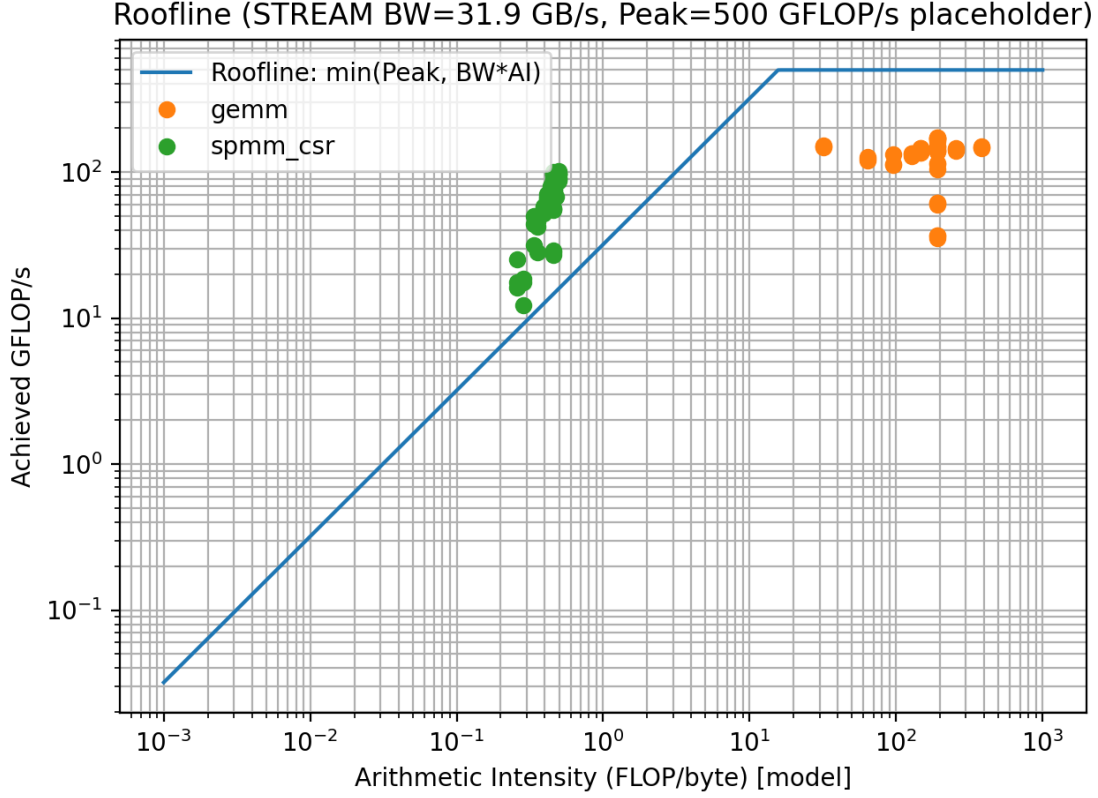


Figure 6: Roofline: achieved performance vs arithmetic intensity.

Analysis. Dense GEMM should appear at higher AI and can approach the compute roof when tiling is effective. CSR-SpMM typically appears at lower AI and is closer to the bandwidth roof, especially for uniform sparsity. When points lie near the bandwidth line, additional threads and SIMD provide diminishing returns unless bandwidth or locality improves.

3.6 Perf counters and latency tails

When `PERF=1` is enabled, `perf stat` collects cycles, instructions, cache misses, LLC misses, and dTLB load misses. These counters support explanations about cache locality and TLB pressure. The CSV also records p50/p95/p99 latency over repeated kernel calls. Tail latency increases are expected when working sets exceed caches or when access patterns become less predictable.

3.7 Conversion cost and amortization

The CSV includes `conv_seconds` as the CSR build cost (dense→CSR proxy). This cost is paid once if the same sparse matrix is reused across many SpMM calls. For small numbers of multiplies, conversion can dominate total time and make dense GEMM preferable even at low densities.

4 Limitations

- Float arithmetic is used; results can differ slightly due to associativity and vectorized reductions.
- The byte model for AI and bandwidth is a simplification. It can overestimate traffic due to cache reuse or underestimate traffic due to write-allocate effects.

- Column-major B disables the SIMD SpMM path in this implementation because inner-loop loads are not contiguous.
- Hybrid-core behavior can affect scaling and perf counter coverage depending on OS and perf permissions.

5 Conclusion

Dense GEMM benefits strongly from cache tiling, SIMD, and multithreading because arithmetic intensity is high and reuse is strong. Performance degrades mainly when working sets exceed caches and memory bandwidth becomes the limiter. CSR-SpMM is faster at low densities because work scales with nnz , but it becomes bandwidth-bound earlier and can scale less with threads under uniform sparsity. The density break-even point marks when sparse overhead and reduced locality outweigh the savings in FLOPs. Structured sparsity (banded or block-diagonal) improves locality and can shift the break-even density upward. Dense→CSR conversion cost matters for short workloads and is amortized when the sparse structure is reused many times.

A Key output files

- `results/results_a2.csv`: raw measurements (median time, GFLOP/s, AI, bandwidth estimate, p50/p95/p99, perf counters when enabled).
- `results/fig_gemm_gflops_threads.png`
- `results/fig_spmm_gflops_threads.png`
- `results/fig_breakeven_density_runtime.png`
- `results/fig_workingset_gflops.png`
- `results/fig_workingset_bandwidth.png`
- `results/fig_roofline.png`